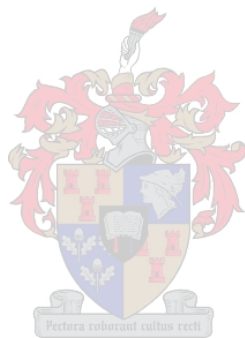


# The crossing number of a graph in the plane

Wynand Winterbach



Thesis presented in partial fulfilment of the requirements for the degree  
Master of Science at the Department of Applied Mathematics  
of the University of Stellenbosch, South Africa

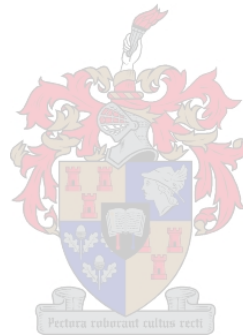


# Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: \_\_\_\_\_

Date: \_\_\_\_\_



# Abstract

Heuristics for obtaining upper bounds on crossing numbers of small graphs in the plane, and a heuristic for obtaining lower bounds to the crossing numbers of large graphs in the plane are presented in this thesis. It is shown that the two–page book layout framework is effective for deriving general upper bounds, and that it may also be used to obtain exact results for the crossing numbers of graphs.

The upper bound algorithm is based on the well–known optimization heuristics of tabu search, genetic algorithms and neural networks for obtaining two–page book layouts with few resultant edge crossings.

The lower bound algorithm is based on the notion of embedding a graph into another graph, and, to the best knowledge of the author, it is the first known lower bound algorithm for the crossing number of a graph. It utilizes Dijkstra’s shortest paths algorithm to embed one graph into another, in such a fashion as to minimize edge and vertex congestion values.

The upper bound algorithms that were developed in this thesis were applied to all non–planar complete multipartite graphs of orders 6–13. A catalogue of drawings of these graphs with low numbers of crossings is provided in the thesis. Lower bounds on the crossing numbers of these graphs were also computed, using lower bounds that are known for a number of complete multipartite graphs, as well as the fact that lower bounds on the crossing numbers of the subgraphs of a graph  $\mathcal{G}$ , are lower bounds on the crossing number of  $\mathcal{G}$ .

A reference implementation of the Garey–Johnson algorithm is supplied, and finally, it is shown that Székely’s algorithm for computing the independent–odd crossing number may be converted into a heuristic algorithm for deriving upper bounds on the plane crossing number of a graph.

This thesis also provides a thorough survey of results known for the crossing number of a graph in the plane. The survey especially focuses on algorithmic issues that have been proposed by researchers in the field of crossing number research.

# Opsomming

Bogrensheuristieke vir die benadering van kruisingsgetalle van klein grafieke in die vlak, en 'n heuristiek vir die benadering van ondergrense vir groot grafieke in die vlak word in hierdie tesis ontwikkel. Daar word getoon dat tweebled boekuitlegte 'n doeltreffende raamwerk bied vir die afleiding van algemene bogrense, en dat dit ook gebruik kan word om eksakte kruisingsgetalwaardes van grafieke te vind.

Die bogrensalgoritmes berus op die welbekende optimeringsheuristieke uit die metodologieë van tabu-soektogte, genetiese algoritmes en neurale netwerke vir die verkryging van tweebled boekuitlegte wat min kruisings realiseer.

Die ondergrensalgoritme berus op die begrip van die inbedding van een grafiek in 'n ander grafiek, en is, na die beste wete van die outeur, die eerste ondergrens algoritme vir die kruisingsgetal van 'n grafiek. Die algoritme maak gebruik van Dijkstra se kortste pad metode om 'n grafiek in 'n ander in te bed sodat die lyn- en puntbelading geminimeer word.

Die bogrensalgoritmes wat in hierdie tesis ontwikkel is, is toegepas op alle nie-planêre volledige veelledige grafieke van ordes 6–13. 'n Katalogus van tekeninge van hierdie grafieke met 'n lae aantal kruisings word in die tesis verskaf. Ondergrense vir die kruisingsgetalle van hierdie grafieke is ook bereken met behulp van ondergrense wat bekend is vir sommige volledige veelledige grafieke asook die feit dat ondergrense vir die kruisingsgetalle van subgrafieke van 'n grafiek  $\mathcal{G}$  dien as ondergrense vir die kruisingsgetal van  $\mathcal{G}$ .

'n Verwysingsimplementasie van die Garey–Johnson algoritme word ook verskaf, en laastens word daar getoon dat die algoritme van Székely vir die bepaling van die onafhanklike-onewe kruisingsgetal omgeskakel kan word na 'n heuristiese algoritme wat bogrense vir die kruisingsgetal van 'n grafiek in die vlak benader.

'n Omvattende literatuurstudie oor bekende resultate vir die kruisingsgetal van 'n grafiek in die vlak word ook in die tesis verskaf. Die literatuurstudie fokus spesifiek op algoritmiese kwessies wat deur navorsers in die kruisingsgetalveld bestudeer is.

# Terms of Reference

The study of the crossing number problem had its genesis in a brick factory in Budapest during the Second World War. In the factory, there were a number of kilns that were connected to several storage yards by means of railway tracks, and bricks were transported via small trucks that ran on the rails. Paul Turán [Tur77], who worked in the factory, noticed that the trucks were often derailed where two rails crossed one another. This led him to consider the problem of minimizing the number of rail crossings in the system. In the brick factory, every kiln was joined by rail to every storage yard. A bipartite graph may be used to model this configuration, and Turán's problem is then equivalent to asking what the minimum number of pairwise edge crossings is over all drawings of the graph.

Turán postponed his study of the problem until after the war. In 1952 he met Zarankiewicz in Poland, to whom he relayed the problem. Two years later, Zarankiewicz published a paper, "On a problem of P. Turán concerning graphs" [Zar54], in which he gave a solution to the problem for the class of bipartite graphs. Unfortunately a flaw in his proof was discovered, as described by Guy [Guy69].

Much of the subsequent research into the problem has focussed on establishing bounds — and in the cases of smaller graphs, exact results — for particular classes of graphs. The problem of determining acceptable upper bounds for the crossing numbers of large graphs was addressed by Leighton [Lei83].

Leighton's methods, however, do not guarantee good bounds for small graphs. Research done by Nicholson [Nic68] and Cimikowski [Cim02] dealt with the problem of finding upper bounds for the crossing numbers of small graphs. Unfortunately, Nicholson's heuristic does not always produce solutions of high quality, and Cimikowski only addressed part of the problem in the sense that he largely ignored the issue of the placement of vertices relative to one another. Finally, the problem of determining lower bounds algorithmically has largely been ignored.

A thorough survey of the results known for the crossing number of a graph in the plane is given in this thesis. The topic of the thesis was proposed by Jan van Vuuren, who was the supervisor of the study, whilst Paul Grobler acted as advisor at various stages of the study. Work for this thesis commenced in February 2002 and was completed in December 2004.

# Acknowledgements

During the course of this thesis, the author has learnt immeasurably more than he could have hoped. He wishes personally to express his gratitude towards

- the Department of Applied Mathematics of the University of Stellenbosch for the use of their computing facilities,
- Prof JH van Vuuren, for his diligent and patient guidance,
- Dr PJP Grobler, for his pragmatic advice,
- Peter van der Merwe, for his friendship and intellectual stimulation,
- Amelia Kühn, for being my other half, so patiently,
- My family, for making it possible for me to study without financial difficulties,
- Werner Gründlingh for sending me articles on the crossing number problem that he stumbled upon,
- Stephen Benecke for help with the typesetting of the thesis,
- The open source movement, especially the Linux team, the Free Software Foundation, and the Python Foundation – thank you for dedicating yourselves to providing the world with so much free and open knowledge and such excellent tools; without your work, the task of writing the numerous programs required for this thesis, as well as its typesetting would have been much more arduous, to say the least.

This thesis is based upon work supported by the National Research Foundation (NRF) under grant number GUN 2053755. Any opinion, findings and conclusions or recommendations expressed in this thesis are those of the authors and do not necessarily reflect the views of the NRF. Financial assistance contributing towards this research project was also granted by the post-graduate bursary office and Research Sub-Committee B of the University of Stellenbosch, and the Harry Crossley Foundation. Additionally, the financial assistance of the Department of Labour (DoL) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the DoL.





# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The brick factory problem . . . . .	1
1.2	A short history of the crossing number problem . . . . .	2
1.3	Scope and objectives of this thesis . . . . .	3
1.4	Thesis overview . . . . .	4
<b>2</b>	<b>Prerequisites</b>	<b>5</b>
2.1	Basic graph theoretic concepts . . . . .	5
2.1.1	Neighbourhoods . . . . .	6
2.1.2	Graph complements, isomorphisms, subgraphs, cliques and minors . . . . .	6
2.1.3	Connectedness . . . . .	8
2.1.4	Special graphs . . . . .	8
2.1.5	Subdivision, planarity and thickness . . . . .	10
2.1.6	Directed graphs . . . . .	14
2.2	Basic concepts in topology . . . . .	15
2.3	Basic concepts in abstract algebra . . . . .	17
2.4	Basic concepts in complexity theory . . . . .	19
2.5	Chapter summary . . . . .	24
<b>3</b>	<b>Crossing Numbers</b>	<b>25</b>
3.1	Drawings and embeddings . . . . .	25
3.1.1	From vertices to points, and edges to curves . . . . .	25
3.1.2	Definitions of graph drawings . . . . .	26
3.1.3	Important variants of drawings . . . . .	31
3.1.3.1	Book drawings . . . . .	31
3.1.3.2	Circular drawings . . . . .	33
3.1.3.3	Embeddings . . . . .	34
3.1.3.4	Other types of drawings . . . . .	34

3.2	Defining the crossing number of a graph . . . . .	35
3.2.1	Crossing numbers based on other types of drawings . . . . .	35
3.2.2	The book crossing number . . . . .	38
3.2.3	Variants of crossing number parameters . . . . .	39
3.3	Chapter summary . . . . .	40
<b>4</b>	<b>Literature review</b>	<b>41</b>
4.1	Parameters related to the crossing number . . . . .	41
4.1.1	The maximum induced planar subgraph problem . . . . .	41
4.1.2	The skewness of a graph . . . . .	42
4.1.3	The vertex splitting number . . . . .	42
4.1.4	The genus of a graph . . . . .	43
4.1.5	The thickness of a graph . . . . .	44
4.1.6	The page number of a graph . . . . .	44
4.1.7	The coarseness of a graph . . . . .	45
4.2	Analytical results for the crossing number problem . . . . .	45
4.2.1	Tutte’s algebraic formulation of crossings in graph drawings . . . . .	45
4.2.1.1	The algebraic structure — crossing chains . . . . .	46
4.2.1.2	Transformations on drawings by cross-coboundaries . . . . .	47
4.2.1.3	A lower bound to the crossing number . . . . .	48
4.2.1.4	Tutte’s main result . . . . .	48
4.2.2	Bounding techniques . . . . .	48
4.2.2.1	Standard counting method . . . . .	49
4.2.2.2	Graph-to-graph embedding . . . . .	50
4.2.2.3	Using graph minors . . . . .	54
4.2.2.4	The method of graph bisection . . . . .	55
4.2.2.5	Edge set partitioning . . . . .	56
4.2.2.6	A sparse graph technique: bounding $\nu(\mathcal{C}_m \times \mathcal{C}_n)$ from below . . .	58
4.2.2.7	Obtaining bounds for other sparse graphs . . . . .	59
4.2.2.8	A typical proof technique demonstrated for the result $\nu(\mathcal{C}_m \times \mathcal{C}_n) = m(n - 2)$ . . . . .	60
4.2.3	Crossing number bounds . . . . .	60
4.2.3.1	General crossing number bounds . . . . .	60
4.2.3.2	Multipartite graphs . . . . .	61
4.2.3.3	Results for $\mathcal{K}_n$ . . . . .	63
4.2.3.4	Progress on crossing numbers of products of graphs . . . . .	64

4.2.3.5	The hypercube and interconnection graphs . . . . .	67
4.2.3.6	Petersen graphs . . . . .	70
4.2.3.7	Complements of cycles . . . . .	71
4.3	Computational methods . . . . .	71
4.3.1	Brute force (exact) algorithms . . . . .	72
4.3.1.1	The Garey–Johnson algorithm . . . . .	72
4.3.1.2	The Harris–Harris algorithm . . . . .	73
4.3.1.3	The Pach & Toth odd crossing number algorithm . . . . .	77
4.3.1.4	Székely’s independent–odd crossing number algorithm . . . . .	81
4.3.2	Heuristic methods . . . . .	85
4.3.2.1	Recursive graph bisection . . . . .	85
4.3.2.2	Two–page layout algorithms . . . . .	88
4.3.2.3	Shahrokhi, Székely, Sýkora and Vrfo’s probabilistic embedding algorithm . . . . .	91
4.4	Chapter summary . . . . .	95
<b>5</b>	<b>Exact methods and novel results</b>	<b>97</b>
5.1	The crossing number of $\mathcal{K}_{1,1,1,1,n}$ . . . . .	97
5.2	An implementation of the Garey–Johnson algorithm . . . . .	100
5.2.1	Description of the <b>GareyJohnson</b> algorithm . . . . .	101
5.2.2	Description of the <b>TestPlanar</b> algorithm . . . . .	101
5.2.3	Description of the <b>ConstructGraph</b> algorithm. . . . .	102
5.2.4	Reducing the number of cases . . . . .	103
5.2.4.1	Symmetry considerations for multipartite graphs . . . . .	103
5.2.4.2	Partial verification . . . . .	104
5.2.4.3	Independent crossing subgraphs . . . . .	104
5.2.5	Example execution of the Garey–Johnson algorithm . . . . .	108
5.3	All drawings may be transformed to two–page layouts . . . . .	110
5.4	Chapter summary . . . . .	115
<b>6</b>	<b>Heuristic methods and novel results</b>	<b>117</b>
6.1	A lower bound algorithm for the crossing number . . . . .	117
6.1.1	Theoretical improvements . . . . .	118
6.1.1.1	How does the vertex congestion relate to crossings at a vertex? . . . . .	118
6.1.1.2	Reconsidering the maximum number of crossings due to edge con- gestion . . . . .	121

6.1.2	A heuristic for edge and vertex congestion computation . . . . .	123
6.1.2.1	The implementation of the algorithm . . . . .	125
6.1.2.2	Determining good vertex embeddings . . . . .	128
6.1.2.3	Complexity of the algorithm . . . . .	129
6.2	Upper bound algorithms for the crossing number . . . . .	129
6.2.1	Edge layout heuristics . . . . .	130
6.2.1.1	An iterating greedy algorithm . . . . .	132
6.2.1.2	The Cimikowski–Shope neural network algorithm . . . . .	134
6.2.1.3	A hybrid genetic and local optimization algorithm . . . . .	137
6.2.2	Vertex arrangement heuristics . . . . .	143
6.3	A new Székely–esque upper bound algorithm . . . . .	153
6.4	Constructing a planarized graph . . . . .	156
6.4.1	Spine drawings . . . . .	156
6.4.2	Planarizing drawings . . . . .	156
6.5	Chapter summary . . . . .	159
<b>7</b>	<b>Computational Results</b>	<b>161</b>
7.1	Convergence of upper bound algorithms . . . . .	161
7.1.1	Algorithmic complexity of the algorithm <b>GreedySide</b> . . . . .	161
7.1.2	Convergence of the Genetic algorithm . . . . .	163
7.1.3	Convergence of the tabu search algorithm . . . . .	165
7.2	Algorithmic output . . . . .	167
7.2.1	Difficult problem instances for the <b>GreedySide</b> algorithm . . . . .	167
7.2.2	Output of the lower bound algorithm . . . . .	169
7.2.3	Output of the Garey–Johnson algorithm . . . . .	169
7.2.4	Results for complete multipartite graphs . . . . .	173
7.3	Chapter summary . . . . .	223
<b>8</b>	<b>Conclusions</b>	<b>225</b>
8.1	Summary of work contained in this thesis . . . . .	225
8.2	Possible future work and open questions . . . . .	226
<b>A</b>	<b>Kuratowski’s Theorem</b>	<b>229</b>
A.1	Connectedness . . . . .	229
A.2	Overview of the method of proof . . . . .	230
A.3	Handling 1–connected graphs and subdivided graphs . . . . .	230
A.4	C–components . . . . .	231
A.5	The overlap graph . . . . .	233
A.6	The <i>coup-de-grâce</i> . . . . .	234

<b>B</b>	<b>Computer Implementations</b>	<b>239</b>
B.1	Edge layout heuristics . . . . .	239
B.1.1	The <b>GreedySide</b> algorithm . . . . .	240
B.1.2	The Cimikowski-Shope neural network algorithm . . . . .	241
B.1.3	The genetic algorithm . . . . .	244
B.1.3.1	The selection mechanism . . . . .	244
B.1.3.2	The mutation mechanism . . . . .	245
B.1.3.3	The crossover mechanism . . . . .	246
B.1.3.4	The genetic algorithm driver . . . . .	249
B.2	Vertex arrangement heuristics . . . . .	251
B.2.1	Preconditioning algorithms . . . . .	251
B.2.1.1	Nicholson’s heuristic . . . . .	251
B.2.1.2	Pósa’s heuristic probabilistic algorithm for finding Hamiltonian cycles . . . . .	254
B.2.2	Tabu algorithm . . . . .	260
B.3	The lower bound algorithm . . . . .	263
B.3.1	Compute weights . . . . .	263
B.3.2	Lowerbound . . . . .	264
B.3.3	LowerTabu . . . . .	267
B.4	Garey-Johnson . . . . .	271
B.4.1	GareyJohnson . . . . .	272
B.4.2	ConstructGraph and TestPlanar . . . . .	275
B.4.3	TestPlanar . . . . .	277
B.5	Miscellaneous algorithms . . . . .	278
B.5.1	The spine subdivision algorithm . . . . .	278
B.5.2	The graph planarization algorithm . . . . .	280
B.5.3	A subgraph testing algorithm for complete multipartite graphs . . . . .	282



# List of Figures

1.1	A layout for the brick factory problem . . . . .	2
2.1	A graphical representation of an undirected graph of order 7 and size 8 . . . . .	5
2.2	Illustration the complement of a graph . . . . .	6
2.3	Illustration of isomorphism between graphs . . . . .	7
2.4	Illustration of a subgraph, spanning subgraph and induced subgraph . . . . .	7
2.5	Illustration of graph minors . . . . .	8
2.6	Illustration the Cartesian product of the graphs $\mathcal{C}_3$ and $\mathcal{P}_2$ . . . . .	9
2.7	Illustration of the concept of a complete graph . . . . .	9
2.8	Illustrations of multipartite and bipartite graphs . . . . .	10
2.9	Illustration of graph subdivision . . . . .	10
2.10	A plane drawing and a non-plane drawing of a planar graph . . . . .	11
2.11	A drawing of an outerplanar graph . . . . .	11
2.12	Illustration of a maximally planar graph . . . . .	12
2.13	Two non-planar graphs, $\mathcal{K}_5$ and $\mathcal{K}_{3,3}$ . . . . .	12
2.14	A decomposition of $\mathcal{K}_6$ into two planar graphs . . . . .	13
2.15	A graphical representation of a directed graph and of a bidirected graph . . . . .	14
2.16	Illustrations of trees . . . . .	15
2.17	Illustration of a plane to sphere stereographic projection . . . . .	16
2.18	Methods of generating 2-manifolds . . . . .	16
2.19	A graph isomorphic to $f(\phi)$ . . . . .	22
3.1	Crossings in self-crossing edges can always be removed . . . . .	26
3.2	Edge intersections involving subsections of edges may be always be removed . . . . .	28
3.3	Tangential edge intersections may be removed . . . . .	28
3.4	Removal of more than a single crossing point at the same point of $\mathbb{R}^2$ . . . . .	28
3.5	Illustration of the general technique for the removal of invalid edge intersections . . . . .	30
3.6	A procedure for reducing the number of crossings between non-adjacent edges . . . . .	31

3.7	A topological and a combinatorial book embedding . . . . .	32
3.8	The possible configurations of two edges in a book . . . . .	32
3.9	A book layout on two pages is equivalent to a circular layout . . . . .	34
3.10	A 3-layered drawing of a graph on 15 vertices . . . . .	35
4.1	Illustrations of ideas in Tutte’s theory . . . . .	46
4.2	The distinct $\mathcal{K}_5$ subgraphs in $\mathcal{K}_6$ . . . . .	49
4.3	A graph-to-graph embedding of $\mathcal{K}_5$ into $\mathcal{K}_{4,3}$ . . . . .	50
4.4	Illustration of crossings types caused in graph-to-graph embeddings . . . . .	51
4.5	Graph-to-graph embeddings may cause crossings at vertices . . . . .	52
4.6	$\mathcal{C}_m \times \mathcal{C}_n$ . . . . .	55
4.7	Construction for an upper bound to $\nu(\mathcal{C}_m \times \mathcal{C}_n)$ . . . . .	58
4.8	$\mathcal{K}_{6,6}$ bipartite construction . . . . .	61
4.9	Optimal drawings of $\mathcal{K}_{1,3,n}$ , $\mathcal{K}_{2,3,n}$ and $\mathcal{K}_{1,1,1,n}$ . . . . .	63
4.10	An upper bound construction for $\nu(\mathcal{K}_n)$ . . . . .	64
4.11	Two variations, $\mathcal{T}_{3,n}$ and $\mathcal{X}_{3,n}$ , on the toroidal grid graph $\mathcal{C}_3 \times \mathcal{C}_n$ . . . . .	65
4.12	$\mathcal{P}_m \times \mathcal{C}_n$ is planar . . . . .	68
4.13	Drawings of hypercube and cube connected cycle networks . . . . .	68
4.14	Drawings of butterfly, wrapper butterfly and Beneš graphs . . . . .	69
4.15	The mesh of trees, and the reduced mesh of trees . . . . .	70
4.16	Crossing permutations considered in the Garey–Johnson algorithm . . . . .	72
4.17	A step by step example of how the Harris–Harris algorithm proceeds with $\mathcal{K}_5$ . . . . .	74
4.18	2-cell and non-2-cell embeddings of $\mathcal{K}_4$ into the torus . . . . .	75
4.19	There are multiple ways to insert crossing edges into a planar graph . . . . .	76
4.20	“Twisting” a pair of adjacent edges, causes a crossing between the pair . . . . .	79
4.21	Transformations of the Pach–Tóth algorithm to render a graph drawing convex . . . . .	80
4.22	Vertex separation described by means of imaginary rays emanating from the vertices . . . . .	82
4.23	All configurations affecting the parity of the number of crossings between two edges . . . . .	84
4.24	A bisection tree for $\mathcal{K}_{4,4}$ . . . . .	86
4.25	The process of reconstructing a drawing of $\mathcal{K}_{4,4}$ from its bisection tree . . . . .	87
4.26	Bhatt and Leighton’s tree of meshes . . . . .	88
4.27	Case 2 of Shahrokhi, Székely, Sýkora and Vrto’s probabilistic embedding algorithm . . . . .	93
4.28	Case 3 of Shahrokhi, Székely, Sýkora and Vrto’s probabilistic embedding algorithm . . . . .	93
4.29	Case 6 of Shahrokhi, Székely, Sýkora and Vrto’s probabilistic embedding algorithm . . . . .	94
5.1	A drawing $\phi$ of $\mathcal{K}_{1,1,1,1,n}$ realising $\nu_\phi(\mathcal{K}_{1,1,1,1,n}) = \nu(\mathcal{K}_{4,n}) + n$ . . . . .	98



5.2	The two possible drawings of $\mathcal{K}_4$ in the plane . . . . .	98
5.3	The graph $\mathcal{F} = \mathcal{K}_4 \cup \langle E_{\mathcal{K}_4}(z_1) \rangle$ . . . . .	99
5.4	Symmetry considerations for multipartite graphs in the Garey–Johnson algorithm	103
5.5	Illustration of how an intersection graph specifies independent crossing subgraphs	105
5.6	Illustration of two independent crossing subgraphs . . . . .	109
5.7	Steps in the execution of the Garey–Johnson algorithm . . . . .	109
5.8	The insertion of subdivision vertices around a vertex . . . . .	111
5.9	Subdividing a graph might lower its book crossing number . . . . .	112
5.10	Projection of a straight line drawing of a graph onto the $x$ -axis . . . . .	114
6.1	For each sub–path shared by a pair of paths, at most a single crossing need occur	119
6.2	No crossings need occur at internal vertices of shared sub–paths . . . . .	119
6.3	Paths starting at the same vertex need never cross one another . . . . .	120
6.4	Moving to a vertex with a lower congestion value is more desirable . . . . .	124
6.5	The intersection graph indicates which edges may possibly cross . . . . .	131
6.6	Illustration of the operation of the crossover operator . . . . .	138
6.7	Circular layouts permit “cell” encodings . . . . .	141
6.8	Step $k$ of the tabu illustration . . . . .	148
6.9	Step $k + 1$ of the tabu illustration . . . . .	148
6.10	Step $k + 2$ of the tabu illustration . . . . .	149
6.11	A mechanical method for drawing construction from Székely’s algorithm . . . . .	154
6.12	Edges need not touch the circle between every pair of adjacent lying vertices . . .	154
6.13	A normal spine drawing . . . . .	156
6.14	The ordering of vertices on the spine determines the order of edge crossings . . .	157
7.1	Algorithmic complexity of the <b>GreedySide</b> algorithm w.r.t. graph order . . . . .	162
7.2	Algorithmic complexity of the <b>GreedySide</b> algorithm w.r.t. graph density . . . . .	163
7.3	An illustration of the effects of the variation of the genetic parameters . . . . .	164
7.4	An illustration of the effects of preconditioning methods on tabu searches . . . . .	165
7.5	Illustration of the effect of the variation of tabu parameters on convergence . . .	166
7.6	$\mathcal{G}^{**}$ . . . . .	167
7.7	A subdivision of $\mathcal{G}^{**}$ . . . . .	168
7.8	A drawing of $\mathcal{K}_6$ generated by the Garey–Johnson algorithm . . . . .	170
7.9	Non-planar complete multipartite graphs of order 6. . . . .	175
7.10	Non-planar complete multipartite graphs of order 7. . . . .	175
7.11	Non-planar complete multipartite graphs of order 8. . . . .	177

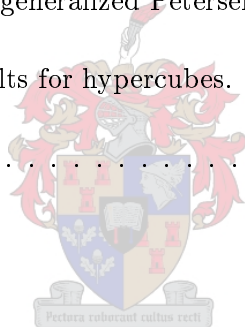
---

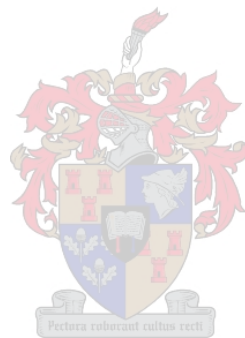
7.12	Non-planar complete multipartite graphs of order 9. . . . .	179
7.13	Non-planar complete multipartite graphs of order 10. . . . .	182
7.14	Non-planar complete multipartite graphs of order 11. . . . .	186
7.15	Non-planar complete multipartite graphs of order 12. . . . .	195
7.16	Non-planar complete multipartite graphs of order 13. . . . .	207
A.1	Transformations to handle 1-connectivity and degree 2 vertices . . . . .	230
A.2	Illustration of C-components . . . . .	231
A.3	The two types of C-component overlapping . . . . .	232
A.4	Overlapping C-components drawn on the same side of a cycle cause crossings . .	233
A.5	A graph and its corresponding overlap graph . . . . .	233
A.6	Finding the minimal cycle length on which C-components may be attached . . .	235
A.7	Case 1 of overlappings in Kuratowski's theorem . . . . .	236
A.8	Case 3 of overlappings in Kuratowski's theorem . . . . .	236
A.9	Case 4 of overlappings in Kuratowski's theorem . . . . .	237
A.10	Sub-case B in Case 4 of overlappings in Kuratowski's theorem . . . . .	238
B.1	An illustration of a rotational transformation with the vertex $v_{i-1}$ as a pivot. . .	254



# List of Tables

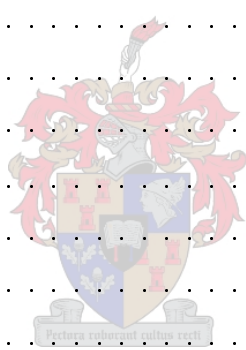
2.1	Definition of the boolean complement. . . . .	21
2.2	Definition of the binary operators OR and AND. . . . .	21
4.1	Crossing number results for the product of cycles $\mathcal{C}_m \times \mathcal{C}_n$ . . . . .	65
4.2	Crossing number results for products with small graphs. . . . .	66
4.3	Crossing number results for the hypercube $\mathcal{Q}_d$ . . . . .	69
4.4	Crossing number bounds for generalized Petersen graphs. . . . .	71
7.1	Lower and upper bound results for hypercubes. . . . .	169
A.1	Overlap possibilities . . . . .	235





# List of Algorithms

5.1	GareyJohnson: The Garey–Johnson algorithm . . . . .	100
5.2	TestPlanar: The permutation enumeration algorithm . . . . .	102
5.3	ConstructGraph . . . . .	103
5.4	GareyJohnson': The revised Garey–Johnson algorithm . . . . .	107
5.5	TestPlanar': The revised permutation enumeration algorithm . . . . .	108
6.1	ComputeWeights . . . . .	126
6.2	ComputeEmbedding . . . . .	127
6.3	GreedySide . . . . .	133
6.4	$b_v^{(\downarrow)}$ or $b_v^{(\uparrow)}$ . . . . .	135
6.5	Cimikowski-Shope . . . . .	136
6.6	GeneticAlgorithm . . . . .	140
6.7	NeighbourhoodSearch . . . . .	145
6.8	TabuSearch . . . . .	151
6.9	ComputePlanarOrderings . . . . .	159





# Glossary

**2-cell** A *surface*  $S$  is said to be 2-cell if it has the property that every point in  $S$  has an open neighbourhood that may be continuously deformed to an open disc.

**adjacent edges** An *edge*  $e$  in a *graph*  $\mathcal{G}$  is said to be *adjacent* to an *edge*  $f \in E(\mathcal{G})$  if  $e$  and  $f$  are *incident* to a common *vertex* in  $\mathcal{G}$ .

**adjacent vertices** A *vertex*  $u$  in a *graph*  $\mathcal{G}$  is said to be *adjacent* to a *vertex*  $v \in V(\mathcal{G})$  if there exists an *edge*  $\{u, v\} \in E(\mathcal{G})$ .

**algorithm** An ordered sequence of operations for solving a problem in a finite number of steps.

**alternating** A pair of *edges*  $e = \{v_i, v_k\}$  and  $f = \{v_j, v_\ell\}$  are said to be alternating if the relative order of their *incident vertices* on the *spine* of a *book* is either  $v_i, v_j, v_k, v_\ell$ , or  $v_j, v_i, v_\ell, v_k$ . The four *incident vertices* are also said to be alternating.

**arc** An *edge* of a *directed graph*. An arc has an orientation towards one of its two *incident vertices*. An arc *incident* to *vertices*  $v_i$  and  $v_j$ , and oriented towards  $v_j$ , is denoted  $(v_i, v_j)$ .

**aspiration criterion** An aspiration criterion is a condition under which a *tabu search algorithm* will override the *tabu value* of a move. For example, a *tabu search algorithm* might choose to perform a *tabu move* if the move will result in an improvement of its current solution.

**asymptotic bound**  $f(n) = O(g(n))$  if there exist a  $c \in \mathbb{R}^+$  and an  $n_0 \in \mathbb{N}$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ . The function  $O$  denotes the order of magnitude of  $g(n)$ .  $f(n) = \Omega(g(n))$  if there exist a  $c \in \mathbb{R}^+$  and an  $n_0 \in \mathbb{N}$  such that  $0 \leq cg(n) \leq f(n)$  for all  $n \geq n_0$ . The function  $g$  is said to be an asymptotic lower bound for  $f$ .

**attribute based memory** A type of memory structure employed by a *tabu search algorithm*. The attribute based memory is used to mark *moves* as *tabu* for a given number of iterations in a *tabu search algorithm*.

**bijection** A function that is both an *injection* and *surjective*.

**bipartite graph** A *multipartite graph* with two partite sets.

**bisection** The process of the removal of a set of *edges* from a *graph*  $\mathcal{G}$  such that the *vertex* set  $V(\mathcal{G})$  of  $\mathcal{G}$  is partitioned into two sets  $V_1(\mathcal{G})$  and  $V_2(\mathcal{G})$  with the properties that  $|V_1(\mathcal{G})|, |V_2(\mathcal{G})| \geq (1/3)|V(\mathcal{G})|$ .

**bisection width** The minimum number of *edges*, denoted  $b(\mathcal{G})$ , in a *graph*  $\mathcal{G}$  whose removal causes  $\mathcal{G}$  to be *bisected*.

**book** An  $n$ -page *book* is the union of  $n$  half-planes, called *pages*, that intersect each other on their finite boundaries. The intersection line is called the *spine* of the book.

**(combinatorial) book drawing** A *graph drawing* in which the *vertices* of the *graph* are placed at distinct positions on the *spine* of a *book*, and where each *edge* is drawn wholly on a single *page* of the *book*.

**book crossing number** The smallest number of *edge* intersections in a *book drawing* of a *graph*  $\mathcal{G}$ , denoted  $\nu_n(\mathcal{G})$  for an  $n$ -page *book*.

**boolean** A boolean variable may assume either the boolean value “TRUE,” or the boolean value “FALSE.” A boolean expression has the property of evaluating to a boolean value.

**cardinality** The number of elements in a set  $S$ , denoted  $|S|$ .

**Cartesian product** The Cartesian product of two *graphs*  $\mathcal{G}$  and  $\mathcal{H}$ , denoted  $\mathcal{G} \times \mathcal{H}$ , is the *graph* with *vertex* set  $V(\mathcal{H}) \times V(\mathcal{G})$ , two *vertices*  $(u_1, u_2)$  and  $(v_1, v_2)$  being *adjacent* in  $\mathcal{G} \times \mathcal{H}$  if and only if either  $u_1 = v_1$  and  $\{u_2, v_2\} \in E(\mathcal{H})$  or  $u_2 = v_2$  and  $\{u_1, v_1\} \in E(\mathcal{G})$ .

**C-component** For a *cycle*  $\mathcal{C}$  in a *graph*  $\mathcal{G}$ , a C-component is the union of a maximally *connected component* in  $\mathcal{G}$  that results from the removal of  $\mathcal{C}$  from  $\mathcal{G}$ , and the set of *edges incident* to the *component* and to  $\mathcal{C}$ . This structure is used in the proof of Kuratowski’s theorem, presented in Appendix A.

**chromosome** An individual in a *population* of candidates employed by a *genetic algorithm*.

**clasp vertex** A *vertex* of a C-component that intersects the *cycle*  $\mathcal{C}$  that defines the C-components.

**clique** An *induced subgraph* of a *graph*, with the property of being *complete*.

**clique number** The largest order of a *clique* in a *graph*  $\mathcal{G}$ , denoted  $\omega(\mathcal{G})$ .

**circular drawing** A type of *drawing* in which *vertices* are drawn, equi-spaced, around the circumference of an imaginary circle, and *edges* are drawn either wholly in the interior, or wholly in the exterior of the circle. Circular *drawings* are equivalent to 2-page *book drawings*.

**class NP** The class of all *decision problems* that may be verified in *polynomial time* if provided with a *certificate* to the problem at hand.

**class NP-complete** A *decision problem*  $L$  is in the class NP-complete if  $L \in \mathbf{NP}$  and if  $L_1 \preceq L$  for all  $L_1 \in \mathbf{NP}$ .

**class NP-hard** A *problem*  $L$  is in the class NP-hard if  $L_1 \preceq L$  for all  $L_1 \in \mathbf{NP}$ .  $L$  need not necessarily be in NP.

**class P** The class of all *decision problems* that may be solved in *polynomial time*.

**closed neighbourhood** The closed neighbourhood of a *vertex*  $v$  in a *graph*  $\mathcal{G}$ , denoted  $N_{\mathcal{G}}[v]$ , is the union  $N_{\mathcal{G}}(v) \cup \{v\}$ , where  $N_{\mathcal{G}}(v)$  is the *open neighbourhood* of  $v$ .

**compact 2-manifold** A *surface* that may be constructed by the addition of handles or of *crosscaps* to the sphere.

**complement** The complement of a *graph*  $\mathcal{G}$ , denoted  $\overline{\mathcal{G}}$ , is a *graph* with the property that  $V(\overline{\mathcal{G}}) = V(\mathcal{G})$  and that contains an *edge*  $\{u, v\}$  if and only if  $\{u, v\} \notin E(\mathcal{G})$ .

**complete graph** The complete *graph* of order  $n$ , denoted  $\mathcal{K}_n$ , has the property that every pair of *vertices* are *joined* by an *edge*.



**component** A *subgraph*  $\mathcal{H}$  of a *graph*  $\mathcal{G}$  is said to be a component of  $\mathcal{G}$  if  $\mathcal{H}$  has the property of being maximally *connected* (i.e., if there is no *vertex*  $u \in V(\mathcal{G}) \setminus V(\mathcal{H})$  for which a  $u - v$  *path* exists where  $v \in V(\mathcal{H})$ ).

**connected** A *graph*  $\mathcal{G}$  is said to be connected if  $\mathcal{G}$  contains a  $u - v$  *path* for every pair *vertices*  $u, v \in V(\mathcal{G})$ .

**$k$ -connected** A *graph*  $\mathcal{G}$  is said to be  $k$ -connected, if the removal of any set of  $j$  *vertices*,  $j < k$ , does not *disconnect*  $\mathcal{G}$ .

**constant time** An operation that executes in  $O(1)$  time is said to be constant time operation, or is said to run in constant time.

**coset** For a *subgroup*  $\mathcal{S}$  of a *group*  $\mathcal{B}$  with the *group* operator  $\odot$ , and for an element  $x \in \mathcal{B}$ , define the set  $x \odot \mathcal{S}$  as  $\{x \odot d : d \in \mathcal{S}\}$ . Then  $x \odot \mathcal{S}$  is said to be a left coset of  $\mathcal{S}$ , or simply a coset of  $\mathcal{S}$ . There is the corresponding concept of a *right coset*  $\mathcal{S} \odot x$ , that is defined as the set  $\{d \odot x : d \in \mathcal{S}\}$ . If  $\odot$  is commutative, the left and right cosets are equal.

**crosscap** A crosscap is created by removing the interior of a disc in a *surface*, and by associating opposite ends on the disc, so that a line entering a point on the disc will leave a diametrically opposed point, on the disc.

**cross-coboundary** For a *drawing*  $\phi$ , an initial cross-coboundary is a *crossing chain* corresponding to the action of “pulling” an *edge*  $\{v_i, v_j\}$  over a *vertex*  $v_k$  in  $\phi$ . A general cross-coboundary is a *crossing chain* expressible as a finite sum of *initial cross-coboundaries*. A cross-coboundary is denoted  $c(ij, k)$ .

**crossing chain** An algebraic structure that describes the *crossing configuration* of a *drawing*  $\phi$  of a *graph*  $\mathcal{G}$ . A crossing chain is constructed as the sum of indeterminates of the form  $[ij, kl]$ , that correspond to pairs of *edges*  $\{v_i, v_j\}$  and  $\{v_k, v_\ell\}$  in  $E(\mathcal{G})$ . An indeterminate has a coefficient  $\lambda(ij, kl)$  that is congruent to the number of crossings between the *edges*  $\{v_i, v_j\}$  and  $\{v_k, v_\ell\}$  in  $\phi$ . Arithmetic may be performed on crossing chains in much the same way as on polynomials.

**crossing number** The crossing number of a *graph*  $\mathcal{G}$  in the *plane*, denoted  $\nu(\mathcal{G})$ , is the minimum number of pairwise *edge* crossings that can exist in a *drawing*  $\phi$  of  $\mathcal{G}$  in *single-cross normal form*.

**crossing set** A set of pairs of *edges* of a *graph*  $\mathcal{G}$  containing those pairs of *edges* in  $\mathcal{G}$  that cross one another.

**cycle** A *walk* in a *graph* in which the begin- and end-*vertices* coincide, and in which no *vertex* is otherwise repeated.

**decision problem** A problem that may be interpreted as a binary question, and may therefore be answered either “yes” or “no.”

**degree** The degree of a *vertex*  $v$  in a *graph*  $\mathcal{G}$ , denoted  $\deg_{\mathcal{G}}(v)$  is the number of *vertices adjacent* to  $v$  in  $\mathcal{G}$ .

**density** The ratio between the *size* of a *graph*  $\mathcal{G}$ , and the *size* of the *complete graph* of the *same order* as  $\mathcal{G}$ .

**directed graph** A *graph*, in which each *edge* is considered *oriented* towards one of its *incident vertices*. An *edge*  $e$  of a *directed graph*  $\mathcal{G}$ , with *incident vertices*  $v_i$  and  $v_j$ , is called an *arc*, and is denoted  $(v_i, v_j)$  if it is oriented towards  $v_j$ , and otherwise  $(v_j, v_i)$  if it is oriented towards  $v_i$ . Therefore,  $(v_i, v_j) \neq (v_j, v_i)$ , and both may be present in  $\mathcal{G}$ .

**disconnected** A *graph* that is not *connected*.

**drawing** A drawing  $\phi$  of a *graph*  $\mathcal{G}$ , is an *injection*  $\phi^{(v)}$  of the *vertices* of  $\mathcal{G}$  to points in a *surface*  $\mathcal{S}$ , and a function  $\phi^{(e)}(\mathcal{G})$  mapping *edges* of  $\mathcal{G}$  to *continuous curves* in  $\mathcal{S}$ .

**edge** A subset of the *vertex set* of a *graph*  $\mathcal{G}$  with two elements, belonging to the possibly empty set of edges  $E(\mathcal{G})$  of  $\mathcal{G}$ .

**edge congestion** In a *graph-to-graph embedding*  $\psi$  of a *graph*  $\mathcal{G}$  into a *graph*  $\mathcal{H}$ , the *edge congestion* of an *edge*  $e' \in E(\mathcal{H})$ , denoted  $c^{(e)}(e', \psi)$ , is defined as the number of *edges* in  $\mathcal{G}$  that map to *paths* in  $\mathcal{H}$  containing  $e'$ .

**edge contraction** A procedure in which the *vertices incident* to an *edge*  $e = \{u, v\}$ , as well as the *edge* itself, are replaced by a single *vertex*  $v'$  that is *adjacent* to all *vertices* in the *neighbourhoods* of  $u$  and  $v$ .

**edge induced subgraph** Any *subgraph*  $\mathcal{H}$  of a *graph*  $\mathcal{G}$  is an *edge induced subgraph* of its constituent *edges*.

**edge layout** A function mapping *edges* of a *graph* to *page numbers* of a *book*. An *edge layout* and a *vertex arrangement* specify a *book layout*.

**edge set partitioning** A technique for deriving analytical lower bounds, or exact results on the *crossing number* of a *graph*  $\mathcal{G}$ . The *edge set* of  $\mathcal{G}$  is partitioned into two subsets  $E(\mathcal{G}) = A \cup B$ , such that the *graph induced* by one of the subsets, say  $\langle A \rangle_{\mathcal{G}}$ , is a *graph* of which the *crossing number* is known. Combinatorial arguments are then used to derive lower bounds to *crossing number* of the *graph induced* by the set  $B$ , that in turn provides bounds to the *crossing number* of  $\mathcal{G}$ .

**embedding** A *drawing* of a *graph*  $\mathcal{G}$  in a *surface* with the property that no two *edges* in  $\mathcal{G}$  intersect. Embeddings are generalizations of *plane drawings*.

**end-vertex** A *vertex*  $v$  of a *graph*  $\mathcal{G}$  with *degree* one; *i.e.*,  $\deg_{\mathcal{G}}(v) = 1$ .

**field** A set of elements  $\mathcal{W}$ , along with two binary operators  $\odot$  and  $\oplus$  and identity elements  $\mathbf{1}$  and  $\mathbf{0}$  corresponding to the two binary operators respectively, is said to be a field  $\mathcal{M}$ , denoted  $\mathcal{M} = (\mathcal{W}, \odot, \oplus, \mathbf{1}, \mathbf{0})$ , if (1)  $\odot$  forms a *group* over  $\mathcal{W} \setminus \{\mathbf{0}\}$  with  $\mathbf{1}$  as the identity element and where  $\odot$  is, in addition, commutative, (2)  $\oplus$  forms a *group* over  $\mathcal{W}$  with  $\mathbf{0}$  as the identity element, and distributivity for  $\odot$  over  $\oplus$  is defined — *i.e.*,  $\mathbf{x} \odot (\mathbf{y} \oplus \mathbf{z}) = \mathbf{x} \odot \mathbf{y} \oplus \mathbf{x} \odot \mathbf{z}$ .

**frequency based memory** A type of memory structure employed by a *tabu search algorithm*. The frequency based memory normally records the number of times that certain *moves* have been performed in the execution of a *tabu search algorithm*, thereby allowing frequently executed *moves* to be penalized, or *moves* that have been executed seldom to be favoured.

**fully triangulated** A *planar graph*  $\mathcal{G}$  is said to be fully triangulated if each *region* in a *plane drawing* of  $\mathcal{G}$  is bounded by exactly three *edges* of  $\mathcal{G}$ .

**genetic algorithm** A heuristic optimization technique designed with the intention of simulating the natural process of evolution.

**(orientable) [(non-orientable) genus]** The *(orientable) [(non-orientable)]* genus of a *graph*  $\mathcal{G}$ , denoted  $\Upsilon(\mathcal{G})$  [ $\hat{\Upsilon}(\mathcal{G})$ ] is equal to the minimum genus for which an *orientable (non-orientable) compact 2-manifold* has an *embedding* of  $\mathcal{G}$ .

**girth** The shortest *cycle* of a *graph*  $\mathcal{G}$ , denoted  $g(\mathcal{G})$ . If  $\mathcal{G}$  contains no *cycles*, then  $g(\mathcal{G}) = \infty$ .

**graph** See *undirected graph*.

**graph-to-graph embedding** A *graph-to-graph embedding* of a *graph*  $\mathcal{G}$  into a *graph*  $\mathcal{H}$  is a pair of *injections*  $\psi(\mathcal{G}) = (\psi^{(v)}, \psi^{(e)})$  where  $\psi^{(v)} : V(\mathcal{G}) \rightarrow V(\mathcal{H})$  and  $\psi^{(e)} : E(\mathcal{G}) \rightarrow \{p : p \text{ is a path in } \mathcal{H}\}$ . An *edge* mapping  $\psi^{(e)}(\{u, v\})$  is subject to the constraint of being a  $\psi^{(v)}(u) - \psi^{(v)}(v)$  *path* in  $\mathcal{H}$ .

**greedy algorithm** A simple heuristic optimization technique whereby a the action leading to the largest improvement in the solution quality, is selected at every step. The *neighbourhood search technique* is an example of a *greedy algorithm*.

**group** A set  $\mathcal{W}$ , along with a binary operator  $\odot$  and an identity element  $\mathbf{1}$  is said to constitute a group if  $\odot$  is associative over the elements of  $\mathcal{W}$ , if  $\mathcal{W}$  is closed with respect to the application of  $\odot$ , if there is an identity element for  $\odot$  in  $\mathcal{W}$ , and if each element in  $\mathcal{W}$  has an inverse under  $\odot$ .

**Hamiltonian cycle** A *cycle*  $\mathcal{C}$  in a *graph*  $\mathcal{G}$  with the property that  $\mathcal{C}$  contains every *vertex* of  $\mathcal{G}$ .

**hypercube** The hypercube of dimension  $n$ , denoted  $\mathcal{Q}_n$ , has  $2^n$  *vertices*, each of which are labelled by a unique  $n$ -bit binary string, and two *vertices* are *adjacent* if their labels differ only in a single position.

**independent crossing subgraph** A *subgraph* of crossing *edges* which may be viewed as a “black box,” by *contracting* all crossings into a single artificial *vertex*.

**independent-odd crossing number** The smallest number of non-*adjacent* pairs of *edges* that intersect an odd number of times in a *drawing* of a *graph*  $\mathcal{G}$  in *normal form*, denoted  $\nu^{(i)}(\mathcal{G})$ .

**induced subgraph** See *vertex induced subgraph* and *edge induced subgraph*.

**injection** A function  $f : D \rightarrow R$  with the property that it maps each element from its domain  $D$ , to a unique element in its range  $R$  — *i.e.*, for an element  $x \in R$  such that  $f(x) = y$ , there is no other element  $x' \in R$ ,  $x' \neq x$  such that  $f(x') = y$ . Injective functions are also referred to as “one-to-one” functions.

**intersection graph (book drawings)** A *graph*  $\mathcal{X}$  constructed from a given *vertex* arrangement of a *graph*  $\mathcal{G}$  on the *spine* of a *book*, containing a *vertex* for every *edge* in  $E(\mathcal{G})$ . A pair of *vertices* in  $\mathcal{X}$  are *adjacent* if the corresponding *edges* in  $\mathcal{G}$  *alternate* on the *spine*.

**intersection graph (Garey-Johnson algorithm)** The *graph*  $\mathcal{X}$  used to obtain *independent crossing subgraphs* for a given *crossing configuration* in a *graph*  $\mathcal{G}$ . Each *edge* in  $\mathcal{G}$  is represented by a *vertex* in  $\mathcal{X}$  and a pair of *vertices* in  $\mathcal{X}$  are *adjacent* if the corresponding pair of *edges* in  $\mathcal{G}$  is present in the crossing configuration. The *components* in  $\mathcal{X}$  that are not *isolated vertices* specify the *independent crossing subgraphs* in  $\mathcal{G}$ .

**intractable problem** A *decision problem* for which no *polynomial time algorithm* is known to solve the problem.

**isolated vertex** A *vertex*  $v$  of a *graph*  $\mathcal{G}$  with *degree* zero; *i.e.*,  $\deg_{\mathcal{G}}(v) = 0$ .

**isomorphic** Two *graphs*  $\mathcal{G}$  and  $\mathcal{H}$  are said to be *isomorphic* if there exists a *bijection*  $\phi : V(\mathcal{G}) \rightarrow V(\mathcal{H})$  such that  $\{u, v\} \in E(\mathcal{G})$  if and only if  $\{\phi(u), \phi(v)\} \in E(\mathcal{H})$ .

**isomorphism** A *bijjective* function showing that two *graphs* are *isomorphic*.

**join** An *edge*  $\{u, v\} \in E(\mathcal{G})$  of a *graph*  $\mathcal{G}$  is said to *join* the *vertices*  $u, v \in V(\mathcal{G})$ .

**layered drawing** In a  $k$ -*layered drawing* of a *graph*  $\mathcal{G}$ , the *vertices* of  $\mathcal{G}$  are placed on  $k$  distinct levels, such that *vertices* on level  $i$  are only *adjacent* to *vertices* on layers  $i - 1$  and  $i + 1$ .

**leaf** An *end-vertex* of a *tree*.

**length** The number of *edges* in a *walk* of a *graph*.

**lexicographic ordering** An ordering of sequences of elements, in which sequences are compared starting with their respective leftmost elements, such that the significance of elements in sequences decrease as comparison moves from the left to the right of the sequences. Entries in language dictionaries are sorted lexicographically.

**manifold** See *compact 2-manifold*.

**maximally planar** A *graph*  $\mathcal{G}$  is said to be *maximally planar* if for any  $e \in E(\overline{\mathcal{G}})$  the *graph*  $\langle E(\mathcal{G}) \cup e \rangle$  is *non-planar*.

**maximal planar subgraph** A *subgraph*  $\mathcal{H}$  of a *graph*  $\mathcal{G}$  is a *graph* with the properties that  $V(\mathcal{H}) = V(\mathcal{G})$ ,  $E(\mathcal{H}) \subset E(\mathcal{G})$  and such that  $\mathcal{H}$  is *planar*, but for any *edge*  $e \in E(\mathcal{G}) \setminus E(\mathcal{H})$ ,  $\langle \mathcal{H} \cup \{e\} \rangle$  is *non-planar*.

**minor** A *graph*  $\mathcal{M}$  obtained from a *graph*  $\mathcal{G}$  by means of a series of *edge contractions* and/or *edge deletions*.

**move** A well defined mechanism by which a candidate solution in a *neighbourhood search* or *tabu algorithm* is transformed into a new candidate solution. For example, this might entail the swapping of two elements in a permutation, or the changing of a binary bit in a string of such bits.

**multipartite graph** A multipartite *graph* with  $n$  partite sets, also known as an  $n$ -partite *graph*, is a *graph*  $\mathcal{G} = (V, E)$ , of which the *vertex* set may be partitioned into  $n$  partite sets  $V = V_1 \cup V_2 \cup \dots \cup V_n$  such that for any  $u, v \in V_i$ ,  $1 \leq i \leq n$  there does not exist an *edge*  $\{u, v\} \in E(\mathcal{G})$ .

**mutate** A procedure employed by a *genetic algorithm* to perturb a candidate solution (*chromosome*) with the purpose of escaping local optima.

**neighbourhood** See *open neighbourhood*.

**neighbourhood search** A simple heuristic optimization technique whereby a the most improving candidate solution reachable from the current candidate solution is continually selected, until no improvement is possible.

**neural network** A heuristic optimization technique that bases its functioning on a simplified model of the functioning of the human brain.

**normal form** A *drawing*  $\phi(\mathcal{G})$  of a *graph*  $\mathcal{G}$  is said to be in *normal form* if (1) every pair of curves in  $\phi$  intersect a finite number of times, (2) curves in  $\phi$  are nowhere tangential and (3) no three curves in  $\phi$  have a common intersection.

**non-orientable** A *compact 2-manifold* is said to be *non-orientable* if it may be generated from the sphere by the addition of a number of *crosscaps*.

**NP** See *class NP*, *class NP-complete* and *class NP-hard*.

**odd crossing number** The smallest number of pairs of *edges* that intersect an odd number of times in a *drawing* of a *graph*  $\mathcal{G}$  in *normal form*, denoted  $\nu^{(o)}(\mathcal{G})$ .

**(open) neighbourhood** The open neighbourhood of a *vertex*  $v$  in a *graph*  $\mathcal{G}$ , denoted  $N_{\mathcal{G}}(v)$ , is the set of *vertices* that are *adjacent* to  $v$  in  $\mathcal{G}$ . See also *closed neighbourhood of a vertex*.

**order** The number of *vertices* in a *graph*  $\mathcal{G} = (V, E)$ , denoted  $|V(\mathcal{G})|$ .

**orbit** An orbit  $P \subseteq S$  of a *bijection*  $f : S \rightarrow S$  on a set  $S$  is a minimal subset of  $S$  that is closed under the application of  $f$ . Minimality requires that no subset of  $P$  is an orbit, and closure requires that if  $x \in P$  then  $f(x) \in P$ .

**orientable** A *compact 2-manifold* is said to be *orientable* if it may be generated from the sphere by the addition of a number of *handles*.

**outerplanar graph** A *graph* is said to be *outerplanar* if it is *planar* and if it permits the construction of an *outerplane drawing*.

**outerplane drawing** A *plane drawing* of a *graph*  $\mathcal{G}$  in the *plane* with the property that every vertex is present in the same *region* in the drawing.

**overlap** A pair of *C-components* overlap either if they have three *clasp vertices* in common, or if each has at least two *clasp vertices* and the *clasp vertices* from the *C-components* are *alternating*.

**overlap graph** For a cycle  $\mathcal{C}$  in a *graph*  $\mathcal{G}$ , the *overlap graph* with respect to  $\mathcal{C}$  contains a *vertex* for each *C-component* of  $\mathcal{C}$ , where a pair of *vertices* are *joined* by an *edge* if their corresponding *C-components* overlap.

**P** see *class P*.

**page** half *plane* intersecting the *spine* of a *book* on its closed *boundary*.

**pairwise crossing number** The smallest number of pairs of *edges* that intersect in any *drawing* of a *graph*  $\mathcal{G}$  in *normal form*, denoted  $\nu^{(p)}(\mathcal{G})$ .

**n-partite graph** See *multipartite graph*.

**path** A *walk* in a *graph*  $\mathcal{G}$  in which no *vertex* is repeated.

**planar** A *graph* is said to be *planar* if it permits a *plane drawing*.

**n-planar crossing number** The minimum, over all partitionings of a *graph*  $\mathcal{G}$  into  $n$  *edge-disjoint subgraphs*, of the sum of the *crossing numbers* of the *subgraphs*.

**plane** The space  $\mathbb{R}^2$ .

**plane drawing** A *drawing* of a *planar graph*  $\mathcal{G}$  in the *plane* with the property that no pair of *edges* in  $\mathcal{G}$  intersect.

**polynomial time algorithm** An *algorithm* of which the *time complexity* may be asymptotically bounded by a polynomial function with respect to its input size.

**polynomial time reducible** A *decision problem*  $L_1$  is polynomial time reducible to a *decision problem*  $L_2$  if a polynomial time mapping, from problem instances of  $L_1$  to problem instances of  $L_2$  exists.

**product of graphs** See *Cartesian Product*.

**rectilinear drawing** A *drawing* of a *graph*  $\mathcal{G}$  in which the *edges* of  $\mathcal{G}$  are drawn as straight lines.

**region** The disjoint subsets of *plane* that are not part of the *vertex* or *edge drawings* of a *graph*, are called regions of the *drawing*.

**simple curve** A simple curve  $c$  in  $\mathbb{R}^2$  is a continuous *injection*  $c : [0, 1] \rightarrow \mathbb{R}^2$ .

**single-cross normal form** A *drawing* in *normal form* with the additional properties that *adjacent edges* never cross, and that no pair of non-adjacent *edges* cross each other more than once, is said to be in single-cross normal form.

**single-edge graph-to-graph embedding** A restricted type of *graph-to-graph embedding*. If a *graph*  $\mathcal{G}$  is single-edge *graph-to-graph embedded* into a *graph*  $\mathcal{H}$ , then *edges* in  $\mathcal{G}$  do not map to *paths* in  $\mathcal{H}$ , but instead to *edges* in  $\mathcal{H}$ . A single-edge *graph-to-graph embedding* is usually denoted by the symbol  $\bar{\psi}$ .

**size** The number of *edges* in a *graph*  $\mathcal{G} = (V, E)$ , denoted  $|E(\mathcal{G})|$ .

**source** The *vertex*  $u$  *incident* to an *arc*  $(u, v)$  from which the *arc* is oriented.

**spanning subgraph** A *subgraph*  $\mathcal{H} = (V', E')$  of a *graph*  $\mathcal{G} = (V, E)$  is said to be a spanning subgraph if  $V' = V$ .

**spine** The intersection of the half *planes* that form the *pages* of a *book*.

**standard counting method** A method for bounding the *crossing number* of a *graph*  $\mathcal{G}$  whereby lower bounds to the *crossing numbers* of *subgraphs* of  $\mathcal{G}$  are added, and crossings that are counted multiple times, due to the intersections of *subgraphs*, are subtracted appropriately.

**star** A *bipartite graph* of the form  $\mathcal{K}_{1,n}$ .

**subdivision** A *graph*  $\mathcal{H}$  is said to be a subdivision of a *graph*  $\mathcal{G}$ , if  $\mathcal{H}$  is obtained from  $\mathcal{G}$  by inserting *vertices* of *degree* two into the *edges* of  $\mathcal{G}$ .

**subcover** A subcover of a *cover*  $\mathcal{U}$  of a set  $S$ , is a subset  $\mathcal{U}'$  of  $\mathcal{U}$ , that *covers*  $S$ .

**subgraph** A subgraph  $\mathcal{H} = (V', E')$  of a *graph*  $\mathcal{G} = (V, E)$  is a *graph* with the properties that  $V' \subseteq V$  and  $E' \subseteq E$ . The subgraph relation is denoted  $\mathcal{H} \subseteq \mathcal{G}$ .

**subgroup** A subgroup  $\mathcal{S}$  of a *group*  $\mathcal{B}$  contains a subset of the *group* elements of  $\mathcal{B}$  and satisfies the *group* axioms over this subset.

**surface** A locally 2-dimensional space, such as the *plane*, the sphere or the torus.

**surjective function** A function  $f : D \rightarrow R$  with the property that for every element  $y \in R$ , there exists an  $x \in D$ , such that  $f(x) = y$ . Surjective functions are also said to be “onto.”

**tabu** A *move* of a *tabu search algorithm* is said to be *tabu* if it has a non-zero *tabu value*.

**tabu search** A heuristic optimization technique that may be seen as a generalization of the *neighbourhood search* technique, whereby a change in solution attributes resulting in a *move* from one candidate solution to the next is classified as *tabu* for a number of steps, so that any *moves* that would introduce the same changes in attributes are also classified as *tabu* and such *moves* would be avoided whilst the *tabu* status for the given change in attributes holds. This behaviour may be overridden via a number of aspiration criteria if a *tabu* classified *move* will lead to a desirable candidate solution.

**tabu tenure** The number of iterations for which a *move* will have a non-zero *tabu value* in a *tabu search algorithm*.

**tabu value** A *move* with a non-zero *tabu value* is not considered by a *tabu search algorithm*, unless it satisfies the set of *aspiration criteria* of the *algorithm*.

**target** The *vertex*  $v$  *incident* to an *arc*  $(u, v)$  towards which the *arc* is oriented.

**thickness** The thickness of a *graph*  $\mathcal{G}$ , denoted  $\theta(\mathcal{G})$ , is the minimum number of *planar edge disjoint subgraphs* into which  $\mathcal{G}$  may be partitioned.

**toroidal grid graph** The *Cartesian product* of a pair of *cycles*.

**tractable problem** A *decision problem* for which a *polynomial time algorithm* is known to solve the problem.

**tree** An acyclic *graph*. A tree of *order*  $n$  has *size*  $n - 1$ . A tree has the property that each of its edges is a *bridge*.

**tuple** An ordered set of elements,  $s_1, s_2, \dots, s_n$ , denoted  $(s_1, s_2, \dots, s_n)$ .

**underlying graph** The underlying *graph* of a *directed graph*  $\mathcal{D} = (V, E)$  is the *undirected graph*  $\mathcal{G} = (V, E')$  obtained from  $\mathcal{D}$  by replacing all *arcs* by *edges*. If  $E$  contains both the *arcs*  $(v_i, v_j)$  and  $(v_j, v_i)$ , only a single *edge*  $\{v_i, v_j\}$  is added to  $E'$ .

**undirected graph** A *graph*,  $\mathcal{G}$  (sometimes indicated by  $\mathcal{G} = (V, E)$ ) consists of a non-empty finite set  $V = V(\mathcal{G})$  called the *vertex* set, as well as a (possibly empty) finite set  $E = E(\mathcal{G})$  of 2-element subsets of  $V(\mathcal{G})$ , called the *edge* set.

**vector** An element of a *vector space*.

**vector space** For a *field*  $\mathcal{M}$  with operators  $\oplus$  and  $\otimes$ , and a set  $\mathcal{W}$  of elements, if, for any  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathcal{W}$  and for any  $k, \ell \in \mathcal{M}$ , the following axioms are satisfied, then  $\mathcal{W}$  constitutes a *vector space* over  $\mathcal{M}$ , and the elements from  $\mathcal{W}$  are called *vectors*, whilst the elements of  $\mathcal{M}$  are called *scalars*.

1. Closure of  $\mathcal{W}$  under  $\oplus$ : If  $\mathbf{u}$  and  $\mathbf{v}$  are elements in  $\mathcal{W}$ , then  $\mathbf{u} \oplus \mathbf{v} \in \mathcal{W}$ .
2. Commutativity of  $\mathcal{W}$  under  $\oplus$ :  $\mathbf{u} \oplus \mathbf{v} = \mathbf{v} \oplus \mathbf{u}$ .
3. Associativity of  $\mathcal{W}$  under  $\oplus$ :  $(\mathbf{u} \oplus \mathbf{v}) \oplus \mathbf{w} = \mathbf{u} \oplus (\mathbf{v} \oplus \mathbf{w})$ .
4. Existence of an identity element in  $\mathcal{W}$  under  $\oplus$ : There is an element  $\mathbf{0}$  in  $\mathcal{W}$ , called a *zero vector* of  $\mathcal{W}$ , such that  $\mathbf{0} \oplus \mathbf{u} = \mathbf{u} \oplus \mathbf{0} = \mathbf{u}$  for all  $\mathbf{u} \in \mathcal{W}$ .

5. Existence of inverse elements in  $\mathcal{W}$  under  $\oplus$ : For each  $\mathbf{u} \in \mathcal{W}$ , there is an element  $-\mathbf{u} \in \mathcal{W}$ , called a *negative* of  $\mathbf{u}$ , such that  $\mathbf{u} \oplus (-\mathbf{u}) = (-\mathbf{u}) \oplus \mathbf{u} = \mathbf{0}$ .
6. Closure under scalar multiplication: If  $k$  is a scalar in  $\mathcal{M}$  and  $\mathbf{u}$  is an element of  $\mathcal{W}$ , then  $k \otimes \mathbf{u} \in \mathcal{W}$ .
7. Distributivity of vectors sums:  $k \otimes (\mathbf{u} \oplus \mathbf{v}) = (k \otimes \mathbf{u}) \oplus (k \otimes \mathbf{v})$ .
8. Distributivity of scalars sums:  $(k \oplus \ell) \otimes \mathbf{u} = (k \otimes \mathbf{u}) \oplus (\ell \otimes \mathbf{u})$ .
9. Associativity of scalar multiplication:  $k \otimes (\ell \otimes \mathbf{u}) = (k \otimes \ell) \otimes \mathbf{u}$ .
10. Scalar multiplication identity:  $1 \otimes \mathbf{u} = \mathbf{u}$ .

**vertex** An element of the non-empty *vertex* set  $V(\mathcal{G})$  of a *graph*  $\mathcal{G}$ .

**vertex arrangement** The order of *vertices* on the *spine* of a *book* in a *book drawing* of a *graph*.

**vertex congestion** In a *graph-to-graph embedding*  $\psi$  of a *graph*  $\mathcal{G}$  into a *graph*  $\mathcal{H}$ , the *vertex congestion* of a *vertex*  $v' \in V(\mathcal{H})$ , denoted  $c^{(v)}(v', \psi)$ , is defined as the number of *edges* in  $\mathcal{G}$  that map to *paths* in  $\mathcal{H}$  containing  $v'$ .

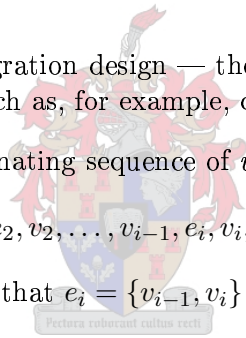
**vertex induced subgraph** A *vertex induced subgraph*  $\mathcal{H} = (V', E')$  of a *graph*  $\mathcal{G} = (V, E)$  has the property that for a pair of *vertices*  $u, v \in V'$ , the *graph*  $\mathcal{H}$  contains the *edge*  $\{u, v\}$  if  $\mathcal{G}$  contains the *edge*  $\{u, v\}$ .

**VLSI design** Very Large Scale Integration design — the design of electronic circuits requiring large numbers of transistors, such as, for example, computer processor design.

**walk** A *walk* in a *graph*  $\mathcal{G}$  is an alternating sequence of *vertices* and *edges*

$$v_0, e_1, v_1, e_2, v_2, \dots, v_{i-1}, e_i, v_i, \dots, v_{n-1}, e_n, v_n,$$

also called a  $v_0 - v_n$  walk, such that  $e_i = \{v_{i-1}, v_i\}$  for  $i = 1, 2, \dots, n$ .





# Translations of Terminology

ENGLISH	AFRIKAANS
adjacent	naasliggend
arc	boog
basis	basis
bipartite	tweeledig
boundary	rand
bounded	begrens
Cartesian	Cartesiese
$n$ -cell	$n$ -sel
clique	klied
clique number	kliedgetal
connected	samehangend
continuous	kontinu
continuous curve	kontinue kromme
crossing number	kruisingsgetal
curve	kromme
decision theory	besluitnemingsteorie
degree	graad
directed graph	gerigte grafiek
disconnected	onsamehangend
edge	lyn
edge set	lynversameling
embedding	inbedding
exterior	buitekant
field	liggaam
genetic algorithm	genetiese algoritme
genus	genus
graph	grafiek
group	groep
incident to	insident
independent-odd crossing number	onafhanklike-onewe kruisingsgetal
induced	geïnduseer
interior	binnekant
intractable	onuitvoerbaar
isomorphic	isomorf
isomorphism	isomorfisme
manifold	variëteit
minor	minor

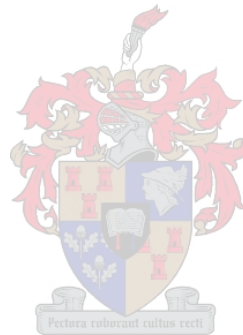
ENGLISH	AFRIKAANS
multipartite	veeledig
neighbourhood	buurgebied of buurpuntversameling
non-orientable	nie-oriënteerbaar
non-planar	nie-planêr
NP-complete	<b>NP</b> -volledig
NP-hard	<b>NP</b> -moeilik
odd crossing number	onewe kruisingsgetal
orientable	oriënteerbaar
outerplanar	buiteplanêr
outerplanar crossing number	buiteplanêre kruisingsgetal
pairwise crossing number	paarsgewyse kruisingsgetal
partite set	partisieversameling
permutation	permutasie
planar	planêr
planar graph	planêre grafiek
rectilinear	reglynig
rectilinear drawing	reglynige tekening
region	gebied
<i>n</i> -regular	<i>n</i> -regulier
rotation scheme	rotasieskema
space	ruimte
sphere	sfeer
subgraph	deelgrafiek
subdivision	subverdeling
tabu algorithm	tabu-algoritme
topology	topologie
torus	torus
toroidal	op die torus
tractable	uitvoerbaar
undirected graph	ongerigte grafiek
vector	vektor
vertex	punt
vertex set	puntversameling

# List of Reserved Symbols

$\binom{a}{b}$	Binomial coefficient given by $\frac{a!}{b!(a-b)!}$ when $a \geq b$ , and 0 otherwise.
$\subseteq$	The relation $\mathcal{H}_1 \subseteq \mathcal{H}_2$ between two sets states that $\mathcal{H}_1$ is a subset of, or equal to the set $\mathcal{H}_2$ . When applied to graphs, $\subseteq$ is interpreted as meaning that $\mathcal{H}_1$ is a subgraph of $\mathcal{H}_2$ .
$\times$	The operation $\mathcal{G}_1 \times \mathcal{G}_2$ generates the Cartesian product of $\mathcal{G}_1$ and $\mathcal{G}_2$ .
$\{a, b, \dots\}$	Set of elements $a, b$ , etc.
$\{v_i, v_j\}$	Edge joining vertices $v_i$ and $v_j$ .
$(v_i, v_j)$	Arc with source vertex $v_i$ and target vertex $v_j$ .
$\langle S \rangle_{\mathcal{G}}$	Induced subgraph within the graph $\mathcal{G}$ for a set $S$ of either edges or of vertices. The subscript may be omitted if the graph $\mathcal{G}$ is clear from the context.
$\preceq$	The relation $L_1 \preceq L_2$ , defined between two decision problems, states that the decision problem $L_1$ is polynomially transformable to the decision problem $L_2$ . In other words, $L_1$ is no harder to solve than $L_2$ .
$[ij, k\ell]$	Indeterminate used to distinguish edge crossings in crossing chains.
$b(\mathcal{G})$	Bisection width of a graph $\mathcal{G}$ .
$c^{(e)}(\psi)$	Maximum edge congestion of an edge in a graph-to-graph embedding $\psi$ .
$c^{(v)}(\psi)$	Maximum vertex congestion of a vertex in a graph-to-graph embedding $\psi$ .
$c^{(e)}(e', \psi)$	Edge congestion of the edge $e'$ in a graph-to-graph embedding $\psi$ .
$c^{(v)}(v', \psi)$	Vertex congestion of the vertex $v'$ in a graph-to-graph embedding $\psi$ .
$\mathcal{C}_n$	Cycle of length $n$ .
$\mathcal{D}$	Generic directed graph.
$\Delta(\mathcal{G})$	Maximum degree of any vertex of a graph $\mathcal{G}$ .
$\deg_{\mathcal{G}}(v)$	Degree of a vertex $v \in V(\mathcal{G})$ in a graph $\mathcal{G}$ .
$\delta(\mathcal{G})$	Minimum degree of any vertex in a graph $\mathcal{G}$ .
$e_i$	Edge with label $i$ .
$E(\mathcal{G})$	Edge set of a graph $\mathcal{G}$ .
$E(v)$	Set of edges incident to the vertex $v$ .
$g(\mathcal{G})$	Girth of a graph $\mathcal{G}$ .
$\mathcal{G}, \mathcal{H}, \dots$	Graphs are identified by capital letters typeset in calligraphy starting with $\mathcal{G}$ .
$\overline{\mathcal{G}}$	Complement of the graph $\mathcal{G}$ .

$\text{GF}(n)$	Galois field of order $n$ .
$\mathcal{K}_n$	Complete graph on $n$ vertices.
$\mathcal{K}_{m,n}$	Complete bipartite graph with partite sets of cardinalities $m$ and $n$ .
$\lambda(ij, k\ell)$	Parameter that is congruent to the parity of the number of crossings between $\{v_i, v_j\}$ and $\{v_k, v_\ell\}$ — used in Tutte’s theory.
$\mathbb{N}$	Set of natural numbers $\{1, 2, 3, \dots\}$
$\mathbb{N}_0$	Set of natural numbers including 0, <i>i.e.</i> , $\mathbb{N} \cup 0$
$\mathcal{N}_n$	Non–orientable compact 2–manifold of genus $n$ .
<b>NP</b>	Class of decision problems for which solutions may be verified in polynomial time, given additional information in the form of a certificate.
$\nu(\mathcal{G})$	Crossing number of a graph $\mathcal{G}$ .
$\nu^{(i)}(\mathcal{G})$	Independent–odd crossing number of a graph $\mathcal{G}$ .
$\nu^{(o)}(\mathcal{G})$	Odd crossing number of a graph $\mathcal{G}$ .
$\nu^{(p)}(\mathcal{G})$	Pair crossing number of a graph $\mathcal{G}$ .
$\nu_n(\mathcal{G})$	$n$ –page crossing number of a graph $\mathcal{G}$ .
$\nu_n^{(B)}(\mathcal{G})$	$n$ –planar crossing number of a graph $\mathcal{G}$ .
$\nu_\phi(\mathcal{G})$	Number of crossings in a drawing $\phi$ of a graph $\mathcal{G}$ .
$\mathcal{O}(f(x))$	Asymptotic (upper–bound) complexity behaviour of order $f(x)$ .
<b>P</b>	Class of decision problems that may be solved in polynomial time.
$\mathcal{P}_n$	Path of length $n$ .
$\phi$	Isomorphism, or drawing of a graph, depending on the context in which it is used.
$\phi(\tilde{e})$	Drawing of the interior of the edge $e$ .
$\phi(\mathcal{G})$	Drawing of a graph $\mathcal{G}$ .
$\psi$	Graph–to–graph embedding.
$\overline{\psi}(\mathcal{G})$	Single–edge graph–to–graph embedding of a graph $\mathcal{G}$ into another graph.
$\psi(e)$	Mapping of an edge to a path in another graph in a graph–to–graph embedding.
$\psi(\mathcal{G})$	Graph–to–graph embedding from a graph $\mathcal{G}$ to into another graph.
$\psi(v)$	Mapping of a vertex to a vertex in another graph in a graph–to–graph embedding.
$\mathcal{Q}_d$	Hypercube of dimension $d$ .
$R$	Region in a drawing of a graph.
$\mathbb{R}$	Set of real numbers.
$\mathbb{R}^+$	The set of positive real numbers.
$\mathbb{R}^n$	Euclidean space of dimension $n$ ; $\mathbb{R}^1 = \mathbb{R}$ .
$\mathcal{S}_n$	Orientable compact 2–manifold of genus $n$ .
$\theta(\mathcal{G})$	Thickness of a graph $\mathcal{G}$ .
$\Upsilon(\mathcal{G})$	Orientable genus of a graph $\mathcal{G}$ .

$v_i$	Vertex with label $i$ .
$V(\mathcal{G})$	Vertex set of a graph $\mathcal{G}$ .
$\mathcal{V}(n, q)$	Vector space of dimension $n$ with vector components from $\mathbb{Z}_q$ ; $q$ is prime and arithmetic is performed modulo $q$ .
$\mathbb{Z}_n$	The set of residue classes of the integers modulo $n$ .
$\mathbb{Z}_n^*$	The set $\mathbb{Z}_n \setminus \{0\}$ .
$x(\phi)$	Crossing chain corresponding to a drawing $\phi$ .
$\Omega(f(x))$	Asymptotic lower-bound complexity behaviour of order $f(x)$ .





# Chapter 1

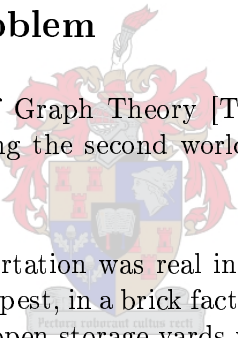
## Introduction

Almost all questions that one can ask about crossing numbers remain unsolved.

— *Paul Erdős (1913–1996)*

### 1.1 The brick factory problem

In the 1977 volume of the *Journal of Graph Theory* [Tur77], the experiences of the Hungarian mathematician, Paul Turán, during the second world war in his fascist home country were described, in his words, as follows:



In July 1944 the danger of deportation was real in Budapest, and a reality outside Budapest. We worked near Budapest, in a brick factory. There were some kilns where the bricks were made and some open storage yards where the bricks were stored. All the kilns were connected by rail with all the storage yards. The bricks were carried on small wheeled trucks to the storage yards. All we had to do was to put the bricks on the trucks at the kilns, push the trucks to the storage yard, and unload them there. We had a reasonable piece rate for the trucks, and the work itself was not difficult; the trouble was only at the crossings. The trucks generally jumped the rails there, and the bricks fell out of them; in short, this caused a lot of trouble and loss of time which was rather precious to all of us (for reasons not to be discussed here). We were all sweating and cursing at such occasions, I too; but *nolens–volens* the idea occurred to me that this loss of time could have been minimized if the number of crossings of the rails had been minimized. But what is the minimum number of crossings? I realized after several days that the actual solution could have been improved, but the exact solution of the general problem with  $m$  kilns and  $n$  storage yards seemed to be very difficult and again I postponed my study of it to times when my fears for my family would end. (But the problem occurred to me again not earlier than 1952, at my first visit to Poland where I met Zarankiewicz. I mentioned to him my “brick factory” problem ...). This problem has ... become a notoriously difficult unsolved problem.

Turán really described a graph theoretic problem, where the kilns and storage yards are vertices in a graph, and where the rails are its edges. The minimum number of crossings that he referred to

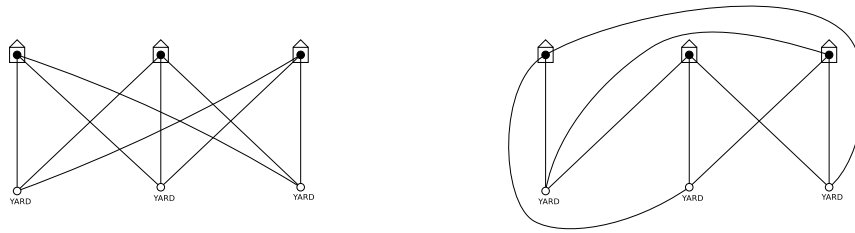


Figure 1.1: A layout for the brick factory problem

has become known as the *crossing number of a graph* (or more precisely in the view of topological variations of the problem, *the plane crossing number of a graph*).

An example of two different layouts for a simple brick factory with three kilns and three storage yards may be seen in Figure 1.1. In part (a), there are a total of 9 crossings, whereas the layout in part (b) displays only a single crossing. This latter layout also achieves the crossing number for the graph.

The theory of the crossing number of a graph has proven to be useful for far more than brick factory design. Probably its most important application to date has been in the design of electronic circuits ([Lei83, SV94, BL84, Lei84, Cim98, Cim96]). The theory of more restricted versions of the crossing number problem for graphs, has also proved useful in methods for reordering sparse matrices<sup>1</sup> [CCDG82] and the implementation of parallel sorting algorithms [Tar72].

## 1.2 A short history of the crossing number problem

Most of the initial research on the crossing number problem, concentrated on finding analytical bounds (and preferably exact values) for crossing numbers of specific classes of graphs. See, for example, the survey by Guy and Erdős [EG73]. Prior to the 1980s the algorithmic aspect of the problem was somewhat neglected; Nicholson [Nic68] was probably the first to develop a heuristic algorithm for the problem in 1968.

In 1983, Garey and Johnson [GJ83] proved that the problem of determining whether the crossing number of a bipartite graph is smaller than or equal to a given  $k$ , is **NP**-complete. From the increasing body of work dedicated to the crossing number problem, it could hardly be said that this served to discourage research in the field.

The academic community that concerns itself with the design of electronic processors (VLSI design), developed a framework for approximating upper bounds on the crossing numbers of graphs with fixed maximum degrees (this is because electronic circuits are typically modelled by graphs where the maximum vertex degree is four). This theory plays an important role in some of the current best approximation algorithms for bounding the crossing number of a general graph. Leighton’s work [Lei83] is seminal in this regard.

It seems that for a period of time, researchers who were involved in pure graph theory and those who were involved in VLSI design, were unaware of each other’s efforts (or at least, the former was ignorant of the work done by the latter). For example, in [SSV95], Shahrokhi, Székely,

---

<sup>1</sup>It is an important problem to find means of reducing the total number of operations needed to solve very large matrix systems, since the  $O(n^3)$  time required for full Gaussian elimination is prohibitive for such systems. Reordering rows and columns in sparse matrices often makes it possible to reduce the amount of required work significantly.



Sýkora and Vrto state<sup>2</sup> that “a recently published paper in graph theory [Mad91] announced the lower bound of  $\Omega(2^{k+0.215\log^2 k})$  for the crossing number of the  $k$ -dimensional cube, where the lower bound of  $\Omega(4^k)$  is easily achievable by standard VLSI techniques [Lei83]”. Largely owing to work done by these four researchers, this situation has changed, and a number of recent papers have gone some way to adapting VLSI techniques for computing upper bounds to the crossing numbers of arbitrary graphs, and for graph layouts on different surfaces — see [SSV95, Szé04, SSSV96a, SSSV94, SSSV97a, SV94] for but a few examples of original work and surveys in this area.

In recent years, many ingenious combinatorial techniques have been developed to attack the analytical side of the crossing number problem. Some problems that seemed intractable just a few years ago, are all but solved now. An example of this is the crossing number of the toroidal grid graph (normally denoted  $\mathcal{C}_m \times \mathcal{C}_n$ ) — the problem of proving that the upper bound value for the crossing number of this graph class is in fact equal to its crossing number, has frustrated the best of minds, and an entertaining survey [Mye98] was devoted to it: “The crossing number of  $\mathcal{C}_m \times \mathcal{C}_n$ : a reluctant induction.” Yet, a recent paper by Glebsky and Salazar [SG04] proves that, for a given  $m$ , this is true for all but a finite number of cases (for which it is still conjectured). Similar impressive results have been established in recent years.

The crossing number problem remains one of the most difficult problems in graph theory to date. But, as Paul Erdős noted, the problem shows its worth by fighting back, and the crossing number problem has proven to be a worthy opponent, but an opponent that rewards those who persevere.

### 1.3 Scope and objectives of this thesis

The scope of this thesis is in combinatorial approaches to the problem of approximation of the plane crossing number of a graph. In particular, combinatorial frameworks for use in computer algorithms are considered. The objectives for this thesis are:

1. to give precise definitions of the concepts of graph drawings, and of the crossing number,
2. to provide the reader with a thorough survey of the state of the art in crossing number research as it stands in 2004, as well as a brief overview of other, related parameters that have been defined for non-planar graphs,
3. to supply a concrete, computer implementation for the Garey–Johnson [GJ83] algorithm for finding exact crossing number results,
4. to develop and implement a lower bound algorithm, based on the notion of graph embedding,
5. to implement heuristic methods based on two-page book embeddings for obtaining upper bounds to the plane crossing numbers of graphs,
6. to apply the implemented upper bound algorithms so as to compute bounds for small complete multipartite graphs, and to catalog the results in the thesis.

---

<sup>2</sup>Citations updated to reflect citations for this thesis

## 1.4 Thesis overview

Apart from this introductory chapter, this thesis comprises a further 7 chapters. A brief overview of some basic concepts from graph theory and complexity theory are given in Chapter 2. As part of this graph theoretical framework, the theory of planarity is also described in this chapter.

Chapter 3 is devoted to giving precise definitions of the various terms related to the graph crossing number problem. There are many types of crossing numbers in the literature, and only those that are considered in this thesis are given a rigorous treatment. The other types of crossing numbers are mentioned at the end of the chapter, and a brief bibliography is given for these related problems. Due to a lack of precise and widely accepted terminology in some of these areas, some new terminology was adopted for this thesis, and hence it is recommended that the reader at least peruse this chapter.

The existing literature for problems related to the graph crossing number problem is surveyed in Chapter 4. The main focus of the chapter is on the general problem of minimizing the number of crossings of edges when graphs are drawn in the plane ( $\mathbb{R}^2$ ). Some related problems are mentioned in passing, although it would have taken the chapter too far afield if the myriad specializations of the problem had been considered. A bibliography of results for some more specialized variants of the problem may be found at the end of this chapter.

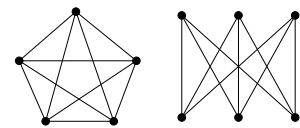
Chapters 5 and 6 detail novel contributions of this thesis. Chapter 5 deals with theory related to the exact computation of the crossing number of a graph, and on means of pruning the total number of computations in such an exact algorithm. Chapter 6 is concerned with heuristic algorithms for approximating the crossing number of a graph. A lower bound algorithm is given, followed by several variants of upper bound algorithms.

Results from the computational trials of algorithms implemented from Chapter 6 are deliberated upon in Chapter 7. Convergence properties of the algorithms used are also considered.

Finally, Chapter 8 concludes the thesis with a summary of contributions of this thesis, and with some open questions emanating from the work contained in this thesis.

# Chapter 2

## Prerequisites



— The graphs  $\mathcal{K}_5$  and  $\mathcal{K}_{3,3}$

The purpose of this chapter is to introduce those basic notions from graph theory (§ 2.1), topology (§ 2.2), abstract algebra (§ 2.3) and complexity theory (§ 2.4), that will be required in later chapters of this thesis.

### 2.1 Basic graph theoretic concepts

An *undirected graph*, or simply, a *graph*  $\mathcal{G} = (V, E)$  is a finite, nonempty set  $V(\mathcal{G})$ , together with a (possibly empty) set  $E(\mathcal{G})$  of unordered two-element subsets of  $V(\mathcal{G})$ . The elements of  $V$  are called *vertices*, while those of  $E$  are called *edges*. The number of vertices in a graph  $\mathcal{G}$  is called the *order* of  $\mathcal{G}$ , denoted by  $|V(\mathcal{G})|$ , while the number of edges in  $\mathcal{G}$  is called the *size* of  $\mathcal{G}$ , denoted by  $|E(\mathcal{G})|$ . If the unordered pair  $e = \{u, v\}$  is an edge of the graph  $\mathcal{G}$ , it is said that the vertices  $u$  and  $v$  are *adjacent* in  $\mathcal{G}$  and that the edge  $e$  *joins*  $u$  and  $v$  in  $\mathcal{G}$ . The edge  $e$  is said to be *incident* with the vertices  $u$  and  $v$ . The vertex  $u$  is also said to be a *neighbour* of  $v$  and *vice versa*.

A graphical representation of an order 7 graph  $\mathcal{G}_1$  of size 8 is shown in Figure 2.1. The vertex set is  $V(\mathcal{G}_1) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$  and the edge set is  $E(\mathcal{G}_1) = \{\{v_1, v_6\}, \{v_1, v_7\}, \{v_2, v_4\}, \{v_3, v_5\}, \{v_3, v_6\}, \{v_3, v_7\}, \{v_4, v_5\}, \{v_5, v_6\}\}$ . The vertices  $v_1$  and  $v_6$  are adjacent in  $\mathcal{G}_1$ , while  $v_1$  and  $v_2$  are not.

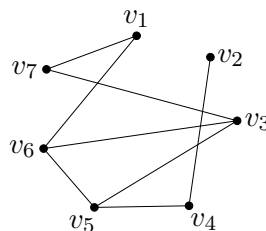


Figure 2.1: A graphical representation of the undirected graph  $\mathcal{G}_1$  of order 7 and size 8.

### 2.1.1 Neighbourhoods

The *open neighbourhood*<sup>1</sup> of a vertex  $v$  in a graph  $\mathcal{G}$  is defined as the set  $N_{\mathcal{G}}(v) = \{u : \{u, v\} \in E(\mathcal{G})\}$ . For any vertex  $v$  in a graph  $\mathcal{G}$ , the number of vertices adjacent to  $v$ , *i.e.*,  $|N_{\mathcal{G}}(v)|$ , is called the *degree* of  $v$  in  $\mathcal{G}$ , denoted by  $\deg_{\mathcal{G}} v$ . If the degree of a vertex is 0, it is called an *isolated vertex*, while if the degree is 1, it is called an *end-vertex*. The minimum degree of vertices in  $\mathcal{G}$  is denoted by  $\delta(\mathcal{G})$ , while the maximum degree of the vertices is denoted by  $\Delta(\mathcal{G})$ . The *edge neighbourhood*<sup>2</sup> of a vertex  $v$  in a graph  $\mathcal{G}$  is the set  $E_{\mathcal{G}}(v) = \{e : e \text{ is incident to } v \text{ in } \mathcal{G}\}$ . When the reference to the graph  $\mathcal{G}$  is clear from the context, the subscripts are often omitted, and one only writes  $N(v)$ ,  $\deg v$  and  $E(v)$ .

Referring to the graph  $\mathcal{G}_1$  in Figure 2.1, the open neighbourhood of the vertex  $v_5$  is  $N_{\mathcal{G}_1}(v_5) = \{v_3, v_4, v_6\}$ , while its edge neighbourhood is  $E_{\mathcal{G}_1}(v_5) = \{\{v_3, v_5\}, \{v_4, v_5\}, \{v_5, v_6\}\}$ . The graph has no isolated vertices, but  $v_2$  is, in fact, an end-vertex. The minimum degree of  $\mathcal{G}_1$  is therefore  $\delta(\mathcal{G}_1) = 1$ , while the maximum degree is  $\Delta(\mathcal{G}_1) = 3$ .

### 2.1.2 Graph complements, isomorphisms, subgraphs, cliques and minors

The *complement*  $\bar{\mathcal{G}}$  of a graph  $\mathcal{G}$  is the graph for which  $V(\bar{\mathcal{G}}) = V(\mathcal{G})$  and  $\{u, v\} \in E(\bar{\mathcal{G}})$  if and only if  $\{u, v\} \notin E(\mathcal{G})$ . A graph  $\mathcal{G}_2$  is shown in Figure 2.2(a), while its complement  $\bar{\mathcal{G}}_2$  is the graph shown in Figure 2.2(b).

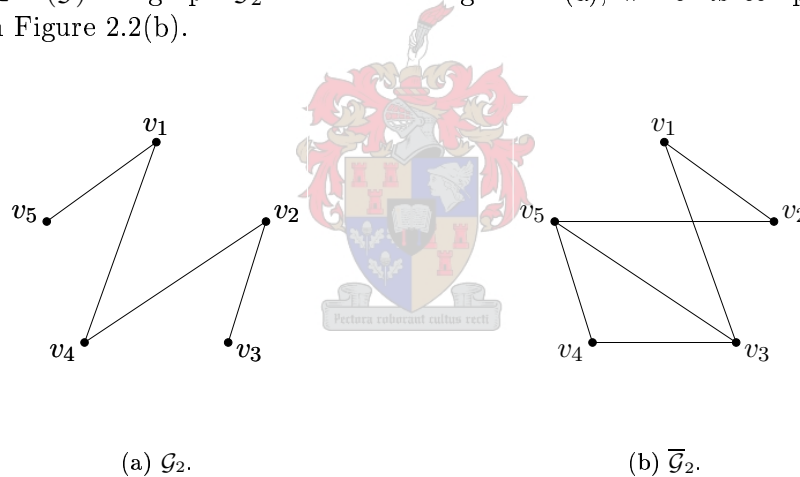


Figure 2.2: Illustration of  $\mathcal{G}_2$  and its complement.

Two graphs  $\mathcal{G}$  and  $\mathcal{H}$  are called *isomorphic*, denoted by  $\mathcal{G} \cong \mathcal{H}$ , if there exists a one-to-one mapping  $\phi : V(\mathcal{G}) \rightarrow V(\mathcal{H})$  such that  $\{u, v\} \in E(\mathcal{G})$  if and only if  $\{\phi(u), \phi(v)\} \in E(\mathcal{H})$ . The function  $\phi$  is called an isomorphism. If  $\phi$  maps  $\mathcal{G}$  onto itself, it is called an automorphism. A graph  $\mathcal{G}$  is said to be *vertex-transitive* if for every pair  $u, v \in V(\mathcal{G})$  there is an automorphism that maps  $u$  to  $v$ . The graph  $\mathcal{G}_4$  shown in Figure 2.3(b) is isomorphic to  $\mathcal{G}_3$ , shown in Figure 2.3(a).

A graph  $\mathcal{H}$  is called a *subgraph* of  $\mathcal{G}$  if  $V(\mathcal{H}) \subseteq V(\mathcal{G})$  and  $E(\mathcal{H}) \subseteq E(\mathcal{G})$ , and is called a *spanning subgraph* of  $\mathcal{G}$  if  $V(\mathcal{H}) = V(\mathcal{G})$  and  $E(\mathcal{H}) \subseteq E(\mathcal{G})$ . For a non-empty vertex subset  $S \subseteq V(\mathcal{G})$  of a graph  $\mathcal{G}$  the so-called *vertex induced subgraph* of  $S$  in  $\mathcal{G}$ , denoted by  $\langle S \rangle_{\mathcal{G}}$ , is the subgraph of  $\mathcal{G}$  with vertex set  $V(\langle S \rangle_{\mathcal{G}}) = S$  and edge set  $E(\langle S \rangle_{\mathcal{G}}) = \{\{u, v\} \in E(\mathcal{G}) : u, v \in S\}$ . For a set of edges  $T$  in  $\mathcal{G}$ , the *edge induced subgraph* of  $T$  in  $\mathcal{G}$ , denoted by  $\langle T \rangle_{\mathcal{G}}$ , is the subgraph of

<sup>1</sup>There is the corresponding concept of a “closed neighbourhood” of  $v$  in  $\mathcal{G}$ , defined as  $N_{\mathcal{G}}[v] = N_{\mathcal{G}}(v) \cup \{v\}$ . However, this concept is not used in the thesis.

<sup>2</sup>This term is not widely used.

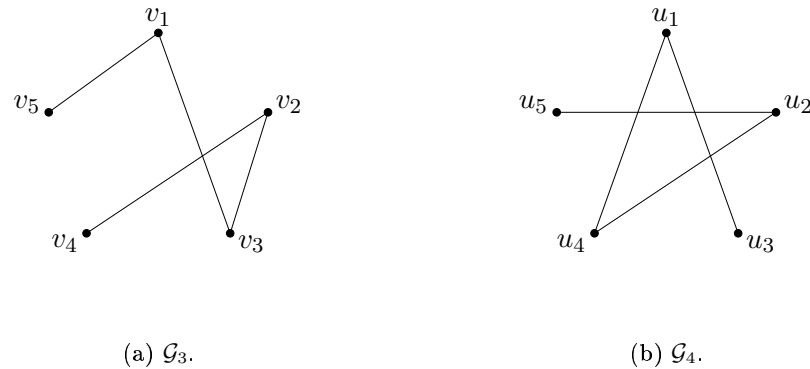


Figure 2.3: Illustration of isomorphism between graphs. Here  $\mathcal{G}_3 \cong \mathcal{G}_4$ .

$\mathcal{G}$  with the vertex set  $V(\langle T \rangle_{\mathcal{G}}) = \{v : v \text{ is incident to an edge in } T\}$  and edge set  $E(\langle T \rangle_{\mathcal{G}}) = T$ . When the reference to the graph  $\mathcal{G}$  is clear from the context, the subscript  $\mathcal{G}$  is omitted. Also, when it is clear from the context that an induced subgraph is either a vertex or an edge induced subgraph, it will simply be called an induced subgraph.

A *clique*  $C$  in a graph  $\mathcal{G}$ , is a vertex induced subgraph of  $\mathcal{G}$  with the property that each vertex in  $C$  is joined to every other vertex in  $C$ . The order of the largest clique in  $\mathcal{G}$ , denoted  $\omega(\mathcal{G})$ , is known as the *clique number* of  $\mathcal{G}$ .

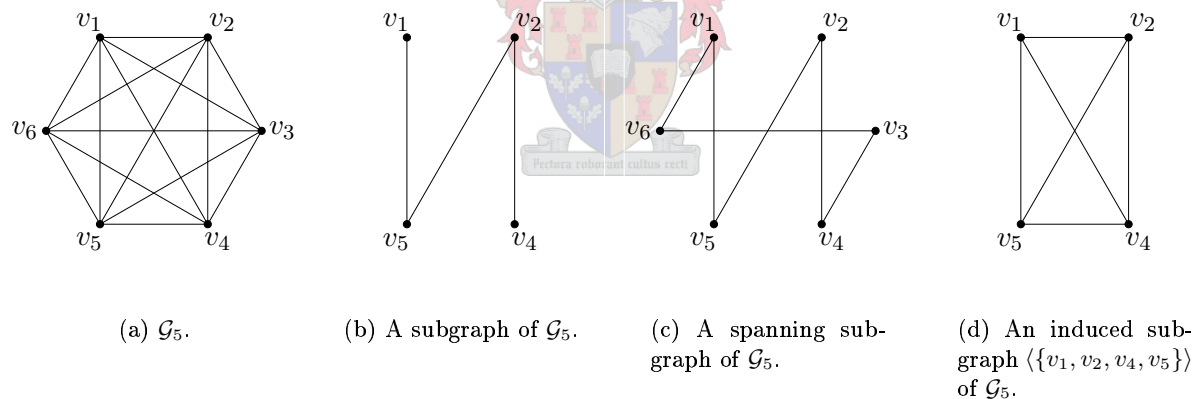


Figure 2.4: Illustration of a subgraph, spanning subgraph and induced subgraph.

The graph shown in Figure 2.4(b) is an example of a subgraph of  $\mathcal{G}_5$ , shown in Figure 2.4(a), while the graph in Figure 2.4(c) is a spanning subgraph of  $\mathcal{G}_5$ . Lastly, the vertex induced subgraph  $\langle \{v_1, v_2, v_4, v_5\} \rangle_{\mathcal{G}_5}$  is illustrated in Figure 2.4(d). This subgraph is a clique in  $\mathcal{G}_5$  of order 4. It is, however, not the largest clique in the graph — the  $\mathcal{G}_5$  is itself a clique, and therefore  $\omega(\mathcal{G}_5) = 6$ . All three subgraphs in Figures 2.4(b)–(d) are edge induced subgraphs of their respective edge sets.

A *minor*  $\mathcal{M}$  of a graph  $\mathcal{G}$ , is a graph  $\mathcal{G}'$  obtained from  $\mathcal{G}$  by a number of *edge contractions* and/or *edge deletions* in  $\mathcal{G}$ . An edge contraction on an edge  $e \in E(\mathcal{G})$  is an operation by which the two incident vertices,  $v_i$  and  $v_j$ , of  $e = \{v_i, v_j\}$  are “joined” to form a single new vertex  $v'_i$ , so that if  $\{v_i, v_k\} \in E(\mathcal{G})$ , then  $\{v'_i, v_k\} \in E(\mathcal{G}')$  and if  $\{v_j, v_\ell\} \in E(\mathcal{G})$ , then  $\{v'_i, v_\ell\} \in E(\mathcal{G}')$  — this introduces the possibility of parallel edges emanating from  $v'_i$ . The edge  $e$  is itself removed

completely. An edge deletion is simply the removal of an edge from  $E(\mathcal{G})$ .

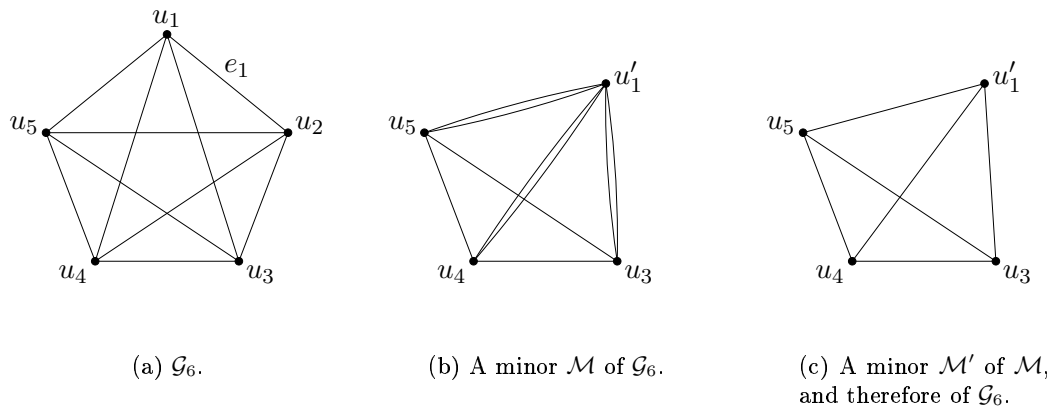


Figure 2.5: Forming a minor from  $\mathcal{M}$  by contracting the edge  $e_1$ , and by deleting parallel edges.

An example of an edge contraction is shown Figures 2.5(a)–(c). The graph  $\mathcal{G}_6$ , shown in Figure 2.5(a) has the edge labelled  $e_1$  contracted to form the graph  $\mathcal{M}$  shown in Figures 2.5(b). The parallel edges are removed to give the minor  $\mathcal{M}'$  in Figure 2.5(c).

### 2.1.3 Connectedness

A *walk* in a graph  $\mathcal{G}$  is an alternating sequence of vertices and edges

$$v_0, e_1, v_1, e_2, v_2, \dots, v_{i-1}, e_i, v_i, \dots, v_{n-1}, e_n, v_n,$$

also called a  $v_0 - v_n$  walk, such that  $e_i = \{v_{i-1}, v_i\}$  for  $i = 1, 2, \dots, n$ . The number of edges in the walk defines its length, while the number of vertices defines its order. When referring to a walk, the edges are often omitted. An example of a walk in the graph  $\mathcal{G}_5$  in Figure 2.4(a) is  $v_1, v_3, v_5, v_1, v_4$ . A walk in which no vertex is repeated is called a *path*. A *cycle* is a walk of length  $n \geq 3$  in which the begin- and end-vertices,  $v_0$  and  $v_n$ , are the same, but in which no other vertices repeat. The *girth* of a graph  $\mathcal{G}$ , denoted  $g(\mathcal{G})$ , is the length of the shortest cycle in  $\mathcal{G}$ . Considering the graph  $\mathcal{G}_5$  in Figure 2.4(a), the walk  $v_1, v_3, v_5$  is a path of order 3 and length 2, while  $v_1, v_3, v_5, v_1$  is a cycle of length 3.

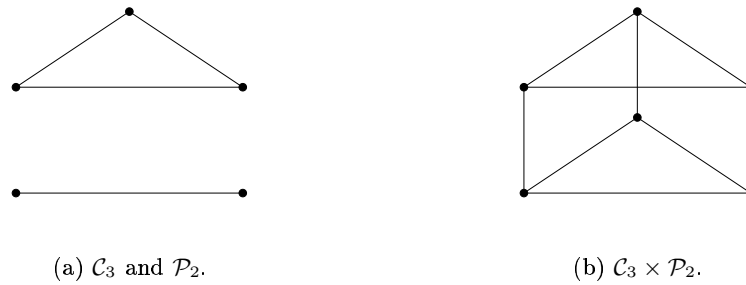
For vertices  $u$  and  $v$  of a graph  $\mathcal{G}$ ,  $u$  is said to be connected to  $v$  if  $\mathcal{G}$  contains a  $u - v$  path. The graph  $\mathcal{G}$  is called a *connected* graph if the vertices  $u$  and  $v$  are connected for any pair  $u, v \in V(\mathcal{G})$ . A graph that is not connected is said to be *disconnected*. A subgraph  $\mathcal{H}$  of  $\mathcal{G}$  is called a *component* of  $\mathcal{G}$  if  $\mathcal{H}$  is a maximally connected subgraph of  $\mathcal{G}$ .

### 2.1.4 Special graphs

A graph solely consisting of a *path* of order  $n$  is so called and denoted by  $\mathcal{P}_n$ . Similarly, a graph consisting of a single *cycle* of length  $n$  is so called and denoted by  $\mathcal{C}_n$ . Paths and cycles are called odd [or even] if they have odd [or even] lengths.

The *Cartesian product* of two graphs  $\mathcal{G}$  and  $\mathcal{H}$ , denoted by  $\mathcal{G} \times \mathcal{H}$ , is the graph with vertex set  $V(\mathcal{H}) \times V(\mathcal{G})$ , two vertices  $(u_1, u_2)$  and  $(v_1, v_2)$  being adjacent in  $V(\mathcal{G} \times \mathcal{H})$  if and only if either

$$u_1 = v_1 \text{ and } \{u_2, v_2\} \in E(\mathcal{H}),$$

Figure 2.6: Illustration the Cartesian product of the graphs  $C_3$  and  $P_2$ .

or

$$u_2 = v_2 \text{ and } \{u_1, v_1\} \in E(\mathcal{G}).$$

From the symmetry in the definition it follows that  $\mathcal{G} \times \mathcal{H} \cong \mathcal{H} \times \mathcal{G}$ . This concept is illustrated in Figure 2.6(b) for the graphs  $C_3$  and  $P_2$ , which are shown individually in Figure 2.6(a). A Cartesian product of paths is sometimes called a *grid graph*, whilst a Cartesian product of cycles is also known as a *toroidal grid graph*.

A graph  $\mathcal{G}$  is called *r-regular* if each vertex of  $\mathcal{G}$  has degree  $r$ . A graph is referred to as regular if it is  $r$ -regular for some  $r \in \mathbb{N}_0$ . A *complete graph* of order  $p$ , denoted by  $\mathcal{K}_p$ , is a graph in which every distinct pair of vertices are adjacent. The complete graph  $\mathcal{K}_p$  is therefore  $(p - 1)$ -regular. As an illustration of the concept, the complete graphs  $\mathcal{K}_5$  and  $\mathcal{K}_6$  are shown in Figure 2.7.

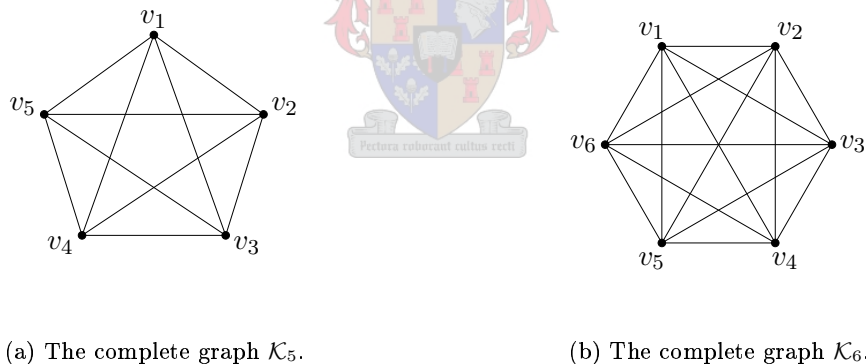


Figure 2.7: Illustration of the concept of a complete graph.

A graph  $\mathcal{G}$  is called *n-partite*,  $n \geq 2$ , if the vertex set may be partitioned into  $n$  subsets  $V_1, V_2, \dots, V_n$ , such that no edge of  $\mathcal{G}$  joins two vertices from the same subset. For  $n = 2$ ,  $\mathcal{G}$  is called *bipartite*, otherwise it is called *multipartite*. If a vertex in a partition set  $V_i$  of a multipartite graph  $\mathcal{G}$  is adjacent to every vertex in the other sets  $\{V_j : j \neq i\}$  for any vertex in  $\mathcal{G}$ , then  $\mathcal{G}$  is called *complete n-partite*. Such a graph  $\mathcal{G}$  with  $|V_i| = p_i$ ,  $i = 1, 2, \dots, n$ , is denoted by  $\mathcal{K}_{p_1, p_2, \dots, p_n}$ . The bipartite graph  $\mathcal{K}_{1, n} \cong \mathcal{K}_{n, 1}$  is a popular graph, called an *n-star*. Illustrations of multipartite and bipartite graphs are shown in Figure 2.8.

The following theorem, a proof of which may be found in [CO93], pp. 26–27, relates bipartiteness to the occurrence of cycles in a graph.

**Theorem 2.1.1** *A graph  $\mathcal{G}$  is bipartite if and only if it has no odd cycles.* ■

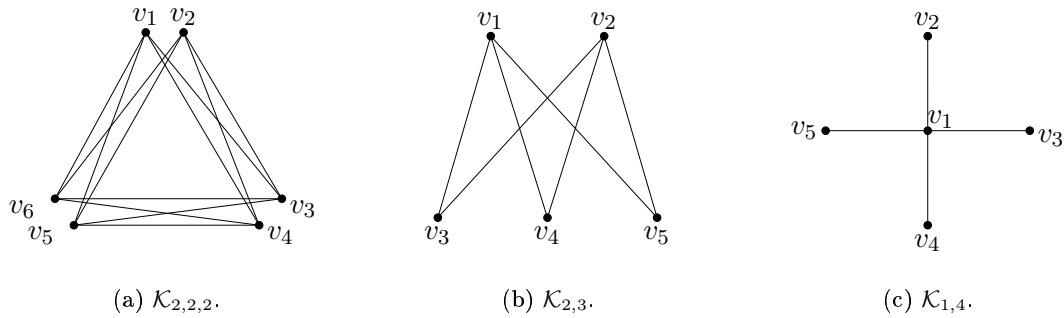


Figure 2.8: Illustrations of multipartite and bipartite graphs.

### 2.1.5 Subdivision, planarity and thickness

A *subdivision* of a graph  $\mathcal{G}$  is a graph  $\mathcal{H}$  that is obtained by replacing the edges of  $\mathcal{G}$  with paths. The vertices in the path which are not end-vertices, are called *subdivision vertices*. Clearly, each subdivision vertex has a degree of 2. The graph  $\mathcal{G}_7$ , a drawing of which is shown in Figure 2.9(b), is a subdivision of the graph  $\mathcal{K}_5$ , which is shown in Figure 2.9(a).

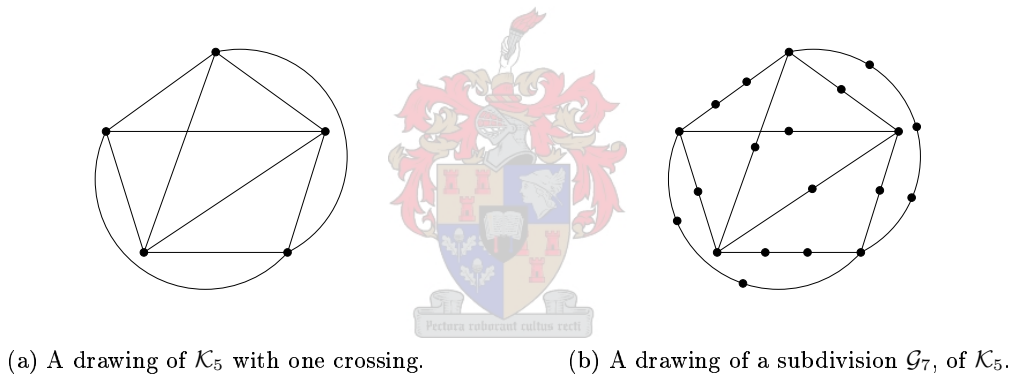


Figure 2.9: Illustration of graph subdivision.

A graph  $\mathcal{G}$  is said to be *planar* if it is possible to draw  $\mathcal{G}$  in the plane (*i.e.*,  $\mathbb{R}^2$ ), so that no curves representing edges of  $\mathcal{G}$  in the drawing intersect or cross one another, and such that no curves intersect the points representing the vertices of  $\mathcal{G}$ . An intersection of a pair of edges is called a *crossing*. A drawing of  $\mathcal{K}_5$  with one crossing is shown in Figure 2.9(a). A drawing of a planar graph  $\mathcal{G}$  that achieves no edge crossings is called a *plane drawing* of  $\mathcal{G}$ . The graph  $\mathcal{K}_4$  is a planar graph; a drawing of  $\mathcal{K}_4$  containing one crossing is shown in Figure 2.10(a), whilst a plane drawing of  $\mathcal{K}_4$  is shown in Figure 2.10(b).

The disjoint *maximally connected* subsets of the plane that are not part of the vertex or edge drawings (*i.e.*, the white areas in a drawing) in a plane drawing of a (planar) graph, are called the *regions* of the drawing. There are four regions in the plane drawing of  $\mathcal{K}_4$ , which are labelled  $R_1, R_2, R_3$  and  $R_4$  in Figure 2.10(c).

A graph  $\mathcal{G}$  is said to be *outerplanar* if it is possible to construct a plane drawing of  $\mathcal{G}$  in which all vertices of  $\mathcal{G}$  are present in a single region of the drawing. Such a drawing is said to be an *outerplane drawing* of  $\mathcal{G}$ . This property is far more restrictive than the property of planarity, and many graphs that are planar fail to be outerplanar. It may be seen from the drawing in



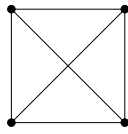
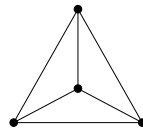
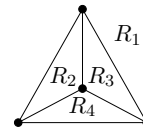
(a) A drawing of  $\mathcal{K}_4$  with a single crossing.(b) A drawing of  $\mathcal{K}_4$  with no crossings.(c) A plane drawing of  $\mathcal{K}_4$  has four regions.

Figure 2.10: A planar graph permits plane drawings (drawings with no crossings), although such a graph may still be drawn with crossings.

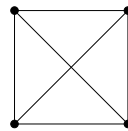
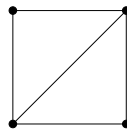
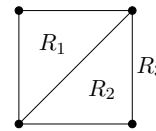
(a)  $\mathcal{K}_4$  is not outerplanar.(b) A drawing of  $\mathcal{K}_4 - \{e\}$  with no crossings.(c) An outerplane drawing of  $\mathcal{K}_4 - \{e\}$  has three regions.

Figure 2.11: An outerplanar graph permits the construction of a plane drawing in which all vertices are present in the same region.

Figure 2.11(a) that  $\mathcal{K}_4$  does not permit an outerplane drawing and is therefore not outerplanar. The removal of a single edge from  $\mathcal{K}_4$  renders an outerplanar graph  $\mathcal{K}_4 - \{e\}$ , which is shown in Figure 2.11(b). The regions for this drawing are shown in Figure 2.11(c), from which it is clear that all of the vertices are present in the region  $R_3$ .

Whilst working in the context of convex polyhedra (whose corresponding graphs are planar graphs), Leonhard Euler realized that there is a fixed relation between the number of regions in a drawing of a polyhedron, and the number of vertices and edges in the polyhedron — a drawing of a polyhedron is equivalent to a plane drawing of its corresponding graph. Euler did not provide a complete proof of the following theorem, since he tried to prove it geometrically, which is far more difficult than in the context of graphs; a proof of this theorem may be found in [CO93].

**Theorem 2.1.2** *In a plane drawing of a connected graph  $\mathcal{G}$  with  $R$  regions, it holds that*

$$|E(\mathcal{G})| - |V(\mathcal{G})| = R - 2. \quad (2.1)$$

A planar graph  $\mathcal{G}$  is said to be *fully triangulated* if each of the regions in a plane drawing of  $\mathcal{G}$  is bounded by exactly three edges. The addition of an edge to a fully triangulated graph, renders the resulting graph non-planar, hence the fact that a fully triangulated graph is also referred to as a *maximally planar* graph. The graph  $\mathcal{G}_8$  shown in Figure 2.12, is a fully triangulated graph.

Euler's formula (2.1) may be used to compute the number of edges present in a maximally planar graph.

**Theorem 2.1.3** *If a graph  $\mathcal{G}$  is maximally planar, then*

$$|E(\mathcal{G})| = 3|V(\mathcal{G})| - 6. \quad (2.2)$$

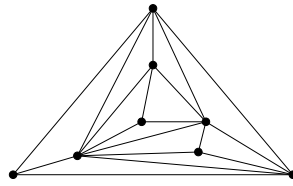


Figure 2.12: Illustration of a maximally planar graph  $\mathcal{G}_8$ .

**Proof:** A maximally planar graph is fully triangulated, since if any region is bounded by four or more edges, additional edges may be drawn within the region to ensure full triangulation. In a fully triangulated graph, every region is bounded by three edges. By summing the number of edges bounding each region, each edge will be counted twice (since an edge has “two sides” each of which must be in a different region). Therefore, there are exactly  $3R/2$  edges in a fully triangulated graph. By using (2.1), it follows that

$$R = 2|V(\mathcal{G})| - 4 \tag{2.3}$$

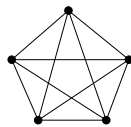
Replacement of the variable  $R$  in (2.1) by the right hand side of (2.3), renders the result (2.2). ■

It follows that a graph of order  $n$  which is not fully triangulated, must contain fewer edges than a fully triangulated graph of order  $n$ .

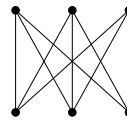
**Corollary 2.1.1** *If a graph  $\mathcal{G}$  is planar and  $|V(\mathcal{G})| \geq 3$ , then*

$$|E(\mathcal{G})| \leq 3|V(\mathcal{G})| - 6. \tag{2.4}$$

**Proof:** A graph which is not fully triangulated, may be rendered so by the addition of the appropriate edges. Thus, it contains fewer edges than a fully triangulated result. ■



(a)  $\mathcal{K}_5$ .



(b)  $\mathcal{K}_{3,3}$ .

Figure 2.13: Two non-planar graphs.

The cornerstone of the theory of planarity is Kuratowski’s theorem. This theorem characterizes all planar graphs in terms of the non-existence of subdivisions of  $\mathcal{K}_5$  or  $\mathcal{K}_{3,3}$ , representations of which may be seen in Figures 2.13(a) and (b) respectively.

**Theorem 2.1.4** (Kuratowski [Kur30], 1930) *A graph is planar if and only if it contains no subgraph isomorphic to a subdivision of  $\mathcal{K}_5$  or to  $\mathcal{K}_{3,3}$ .*

**Proof:** The proof of the first part of the theorem, namely that a non-planar graph contains a subdivision of either  $\mathcal{K}_5$  or of  $\mathcal{K}_{3,3}$ , is tedious, and may be found in Appendix A. It is a

simple matter to prove the second part of Theorem 2.1.4, namely that a graph is non-planar if it contains a subdivision of either  $\mathcal{K}_5$ , or of  $\mathcal{K}_{3,3}$ . ■

In the proof of Theorem 2.1.4, it must be shown that both  $\mathcal{K}_5$  and  $\mathcal{K}_{3,3}$  are non-planar, and that a subdivision of a non-planar graph is itself non-planar.

**Theorem 2.1.5** *The graphs  $\mathcal{K}_5$  and  $\mathcal{K}_{3,3}$  are non-planar.*

**Proof:**  $\mathcal{K}_5$  has 5 vertices and 10 edges. Using (2.4), one obtains  $|E(\mathcal{K}_5)| = 10 > 9 = 3|V(\mathcal{K}_5)| - 6$ . By the contra-positive of Corollary 2.1.1,  $\mathcal{K}_5$  is therefore non-planar.

Now, suppose that a plane drawing of  $\mathcal{K}_{3,3}$  exists. In such a drawing, every region must be bounded by 4 or 6 edges (because, according to Theorem 2.1.1, a bipartite graph contains no cycles of odd length). By summing the number of edges over all  $R$  regions, one obtains  $4R \leq 2|E(\mathcal{K}_{3,3})|$  (since, each edge is counted in two regions). Letting  $R = (1/2)|E(\mathcal{K}_{3,3})|$  in (2.1), the inequality  $|E(\mathcal{K}_{3,3})| \leq 2|V(\mathcal{K}_{3,3})| - 4$  results. Finally,  $\mathcal{K}_{3,3}$  has 6 vertices, but this implies, from the above equation, that  $|E(\mathcal{K}_{3,3})| \leq 8$ , which contradicts the fact that  $|E(\mathcal{K}_{3,3})| = 9$ . ■

If an edge that is crossed by another edge is subdivided in a graph, then it seems obvious that some of the sub-edges in the subdivided edge will be crossed. This is proved in the following proposition.

**Proposition 2.1.1** *A subdivision  $\mathcal{H}$  of a non-planar graph  $\mathcal{G}$ , is non-planar.*

**Proof:** Let  $\mathcal{H}$  be a subdivision of a graph  $\mathcal{G}$  and suppose that  $\mathcal{H}$  is planar. In a drawing of  $\mathcal{H}$ , let every vertex of degree two, as well its incident edges, be replaced by a single edge, joining its adjacent vertices. This renders a drawing of  $\mathcal{G}$  without any crossings, which is a contradiction. ■

Kuratowski's theorem does not directly lead to an efficient algorithm for ascertaining the planarity of a graph. The most efficient algorithm for planarity testing is due to Hopcroft and Tarjan [HT74]. This remarkable algorithm runs in  $O(|V(\mathcal{G})|)$  time. The basic ideas used in the implementation of the algorithm rest on the concepts used in the proof of Kuratowski's theorem.

The *thickness of a graph*  $\mathcal{G}$ , denoted  $\theta(\mathcal{G})$ , is the smallest number of subsets into which  $E(\mathcal{G})$  may be partitioned, such that each of the subgraphs on the vertex set  $V(\mathcal{G})$  and a subset of edges is planar. Therefore, it follows that  $\theta(\mathcal{G}) = 1$ , if  $\mathcal{G}$  is planar.

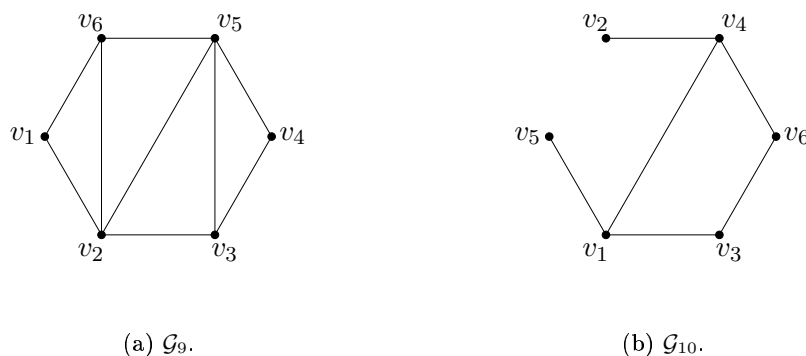


Figure 2.14: A decomposition of  $\mathcal{K}_6$  into two planar graphs,  $\mathcal{G}_9$  and  $\mathcal{G}_{10}$ .

The graph  $\mathcal{K}_6$  is non-planar, since it contains the graph  $\mathcal{K}_5$  as a subgraph, which has been shown to be non-planar in Theorem 2.1.5, and therefore  $\theta(\mathcal{K}_6) > 1$ . A decomposition of  $\mathcal{K}_6$  into two planar subgraphs  $\mathcal{G}_9$  and  $\mathcal{G}_{10}$  is shown in Figure 2.14, and this suffices to prove that  $\theta(\mathcal{K}_6) = 2$ .

### 2.1.6 Directed graphs

An *arc* is an edge of a graph that is directed towards one of the vertices with which it is incident. A *directed graph*, or simply, a *digraph*  $\mathcal{D}$ , is a nonempty set of vertices  $V(\mathcal{D})$  and a (possibly empty) set  $E(\mathcal{D})$  of *ordered* pairs from  $V(\mathcal{D})$ , which are the arcs of  $\mathcal{D}$ . The arcs  $(u, v)$  and  $(v, u)$  in a digraph are different, and each is said to be the *opposite* of the other. The *underlying graph*  $\mathcal{G}$  of a digraph  $\mathcal{D}$ , is the graph that is obtained if all arcs are replaced by edges. A directed graph may not contain both the arcs  $(u, v)$  and  $(v, u)$ . If  $(u, v)$  is an arc of a digraph  $\mathcal{D}$ , then  $u$  is said to be *adjacent to*  $v$ , and  $v$  is said to be *adjacent from*  $u$ . The vertex  $u$  is said to be the *source* of  $(u, v)$ , and the vertex  $v$  is said to be its *target*.

A variation of the concept of a directed graph, is that of a *bidirected graph*, whose definition is the same as that of a directed graph, except for the fact that where a pair of vertices in a directed graph may only be joined by a single arc, a pair of vertices  $u$  and  $v$  in a bidirected graph must be joined either by the pair of opposite facing arcs  $(u, v)$  and  $(v, u)$  or by no arcs at all. The underlying graph of a bidirected graph is obtained by replacing each pair of arcs  $(u, v)$  and  $(v, u)$  by a single edge  $\{u, v\}$ .

A graphical representation of an order 7 digraph,  $\mathcal{G}_{11}$  of size 8 is shown in Figure 2.15(a), whilst a graphical representation of an order 7 bidirected graph,  $\mathcal{G}_{12}$  of size 16 is shown in Figure 2.15(b). The vertex set for the digraph is  $V(\mathcal{G}_{11}) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$  and the vertex set for the bidirected graph is  $V(\mathcal{G}_{12}) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ , whilst the edge set for the digraph is  $E(\mathcal{G}_{11}) = \{(v_1, v_6), (v_2, v_4), (v_3, v_5), (v_3, v_6), (v_3, v_7), (v_5, v_4), (v_5, v_6), (v_7, v_1)\}$  and the edge set for the bidirected graph is  $E(\mathcal{G}_{12}) = \{(v_1, v_6), (v_6, v_1), (v_2, v_4), (v_4, v_2), (v_3, v_5), (v_5, v_3), (v_3, v_6), (v_6, v_3), (v_3, v_7), (v_7, v_3), (v_5, v_4), (v_4, v_5), (v_5, v_6), (v_6, v_5), (v_7, v_1), (v_1, v_7)\}$ . In the digraph  $\mathcal{G}_{11}$ , the vertex  $v_1$  is adjacent to the vertex  $v_6$  and  $v_6$  is adjacent from  $v_1$  in  $\mathcal{G}_{11}$ . In the bidirected graph  $\mathcal{G}_{12}$ , the vertex  $v_1$  is both adjacent to and adjacent from the vertex  $v_6$  and likewise for  $v_6$ . The graph  $\mathcal{G}_1$  shown in Figure 2.1 is the underlying graph of both  $\mathcal{G}_{11}$  and  $\mathcal{G}_{12}$ .

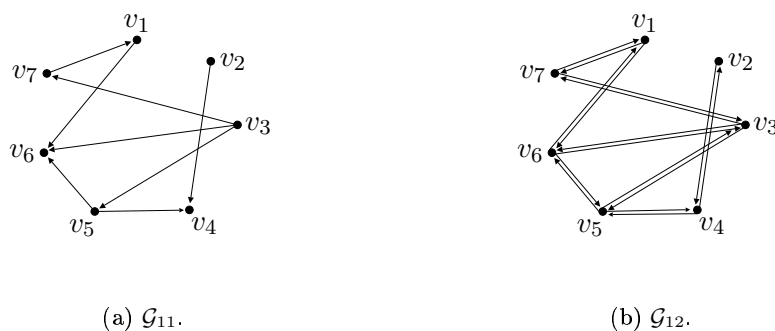


Figure 2.15: A graphical representation of the directed graph  $\mathcal{G}_{11}$ , of order 7, and size 8 and of a bidirected graph  $\mathcal{G}_{12}$ , of order 7 and size 16.

Most of the definitions in this chapter may be amended easily to apply to directed graphs and bidirected graphs; in general, the definitions follow almost immediately. Since these concepts are not required for the way in which directed graphs are used in this thesis, the interested reader may find the appropriate definitions in [Har69, CO93].

The simplest connected graph structure is known as a *tree*, which is an acyclic connected graph. A *leaf* of a tree  $T$  is an end-vertex of  $T$ . A tree of order 10 is shown in Figure 2.16(a), in which



(a) A tree of order 10 and size 9, with 5 leaves.

(b) A tree with directed edges of order 15 and size 14, with 8 leaves.

Figure 2.16: Illustrations of trees, with leaves indicated as white vertices.

the 5 leaves are indicated as white vertices. At a conceptual level, trees serve as a useful model for reasoning about decision and recursive sub-partitioning problems. In this case, the edges are considered as directed (*i.e.*, an edge is directed towards one of its incident vertices). In a decision problem, a vertex corresponds to a configuration resulting from prior choices. All choices that may be made given the current configuration, are modelled as edges directed towards new configurations (*i.e.*, vertices), and away from the current vertex. There is normally one vertex that represents the initial configuration, before any choices have been made. In a partitioning problem, where a set is partitioned into  $x$  subsets, and where each of the  $x$  subsets is, in turn, partitioned into a number of subsets, a vertex  $v$  represents a subset, and the edges directed away from  $v$  are directed towards vertices which are subsets of a partition of  $v$  (or technically, of the subset represented by  $v$ ). A tree of order 15, containing directed edges, is shown in Figure 2.16(b). The vertex  $r$  represents an initial configuration of a decision problem represented by the binary tree.

## 2.2 Basic concepts in topology

The crossing number problem has been studied in the context of other types of “surfaces” (more specifically, other types of *topological spaces*) apart from the plane. The so-called orientable and non-orientable compact 2-manifolds (or simply compact 2-manifolds as they are often called), which are well understood concepts from the field of topology, have been a natural and popular choice. This is especially true because the plane as a surface for graph drawing is equivalent to the sphere, which is an orientable 2-manifold.

This equivalence of the plane and sphere in this regard may be seen from a projection shown in Figure 2.17(a), known as the stereographic projection. Let  $N$  and  $S$  be two opposing points on a sphere  $\Sigma$ , such that  $S$  is the only point on  $\Sigma$  that intersects the plane  $\Pi$ . Let  $\ell$  be a ray from  $N$  to a vertex  $v$  of a graph drawn in  $\Pi$ . There is only one point on  $\Sigma$  which intersects  $\ell$ , and this is the projected position of  $v$  on  $\Sigma$ . This projection is performed for every vertex, and the same process is repeated for each edge, where the edges are “traced” out on  $\Sigma$ , along their paths in  $\Pi$ .

The properties that render a topological space a compact 2-manifold are not discussed here; the interested reader is referred to Lee’s [Lee00] introduction on the subject. For the purposes of this thesis, it suffices to know that such spaces are locally 2-dimensional (they are therefore plane-like at a local level) and due to a theorem by Brahana [Bra21], these spaces may be constructed by attaching a number either of *handles*, or a number of *crosscaps* to the sphere.

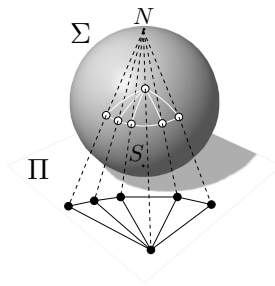
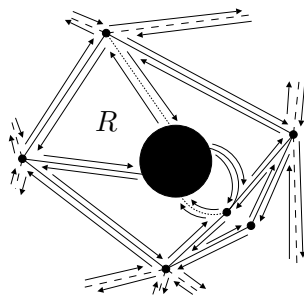
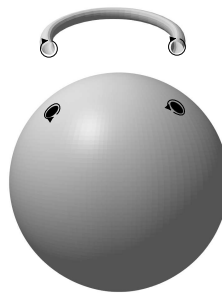


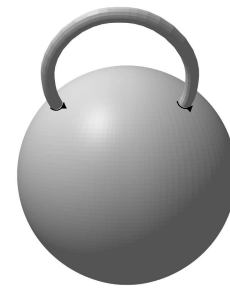
Figure 2.17: An illustration of a plane to sphere stereographic projection.



(a) A surface with a crosscap represented by a black disc.



(b) The sphere and cylinder are both orientable.



(c) The surface formed from the sphere and cylinder is orientable.

Figure 2.18: Methods of generating 2-manifolds.

**Non-orientable surfaces:** To obtain a non-orientable surface, crosscaps are added to the sphere. A crosscap may be created by removing the interior of a disc in the sphere and by associating opposite ends on this disc, so that any line entering a point on the disc will leave a diametrically opposed point on the disc. This is shown in Figure 2.18(a), with the black disc representing the removed interior. The number of crosscaps that characterize a non-orientable surface is called its *genus*, and the non-orientable surface of genus  $n$  is denoted  $\mathcal{N}_n$ .

With this construction, the meaning of “non-orientability” may be qualified. The region  $R$  in Figure 2.18(a) moves through the crosscap, and is an ideal candidate for illustration. A region in a graph drawing is given an orientation when the edges are considered as being directed, so that the target of an edge meets the source of the edge following it around the region. In a normal plane drawing of a graph, this results in an orientation that is either clockwise, or anti-clockwise. Contrast this with the orientation of the region  $R$  shown in Figure 2.18(a) — when it moves through the crosscap, its orientation changes to the opposite direction. For this reason, it is said to be non-orientable.

**Orientable surfaces:** From the previous definition of non-orientable surfaces, orientable surfaces are defined as those surfaces for which consistent orientations may be defined for each region. To obtain an orientable surface, handles are added to the sphere. To add a handle, two disjoint discs are chosen in the sphere, and their boundaries are oriented in the same direction (Figure 2.18(b)). Next a (truncated) cylinder is taken, and its ends are oriented so that the orientations of each of its two “lids” correspond to the orientations of the two respective discs in the sphere. The “lids” of the cylinder with the borders of the discs in the sphere are matched such that the orientations agree (Figure 2.18(c)). The result is a new orientable surface, which is

homeomorphic (*i.e.* may be deformed) to the torus. This procedure may be repeated an arbitrary number of times to obtain a sphere with  $n$  handles. Denote this surface as  $\mathcal{S}_n$ . The number,  $n$ , is referred to as the *genus* of the surface.

## 2.3 Basic concepts in abstract algebra

A *group*  $\mathcal{B} = (\mathcal{W}, \odot, \mathbf{1})$  is a set of elements  $\mathcal{W}$ , called the *group elements*, a binary operation  $\odot$ , called the *group operator*, and an *identity element*  $\mathbf{1} \in \mathcal{W}$ . The operator  $\odot$  is applied to two group members, and one writes  $x \odot y$  to denote the application of  $\odot$  to the elements  $x, y \in \mathcal{W}$ . The following four axioms characterize a group in terms of the properties of the operator  $\odot$ , and the identity element  $\mathbf{1}$ . For an element  $x \in \mathcal{W}$ , one may also write  $x \in \mathcal{B}$ . For all  $x, y, z \in \mathcal{B}$ ,

1. Associativity:  $(x \odot y) \odot z = x \odot (y \odot z)$ .
2. Closure:  $x \odot y \in \mathcal{B}$ . The group is said to be *closed* under the operation  $\odot$ .
3. Identity element: there is an identity element,  $\mathbf{1} \in \mathcal{B}$  such that  $\mathbf{1} \odot x = x \odot \mathbf{1} = x$ .
4. Existence of Inverses: for each element  $x \in \mathcal{B}$ , there is an inverse element in  $\mathcal{B}$ , denoted  $x^{-1}$ , such that  $x \odot x^{-1} = x^{-1} \odot x = \mathbf{1}$ .

The group  $\mathcal{B}$  is said to be finite, if  $\mathcal{W}$  is finite, and conversely, it is said to be infinite, if  $\mathcal{W}$  is infinite. If the additional axiom of commutativity — *i.e.*, for all  $x, y \in \mathcal{B}$ , it is true that  $x \odot y = y \odot x$  — is added, the resulting group is said to be an *Abelian* group.

An example group, denoted  $(\mathbb{Z}_n^*, \times, 1)$ , has the set of group elements  $\{1, 2, \dots, n-1\}$ , integer multiplication modulo  $n$  (where  $n$  is a prime number) as the group operator, and 1 as the identity element. Associativity is guaranteed by the operation of multiplication, the group is closed under multiplication, because operations are taken modulo 7, and all elements have inverses, which is proven by noting that  $2 \times 4 \equiv 1 \pmod{7}$ ,  $3 \times 5 \equiv 1 \pmod{7}$  and  $6 \times 6 \equiv 1 \pmod{7}$ .

A *subgroup*  $\mathcal{S}$  of a group  $\mathcal{B}$ , contains a subset of the group elements of  $\mathcal{B}$ , and satisfies the group axioms. It is therefore required to contain the identity element of  $\mathcal{B}$ . The subgroup relation is denoted  $\mathcal{S} \subseteq \mathcal{B}$ .

The group  $\mathcal{S}_1$  with the set of group elements  $\{1, 6\}$  is a subgroup of  $(\mathbb{Z}_7^*, \times, 1)$ , and it may be verified that it satisfies all the group axioms. Another subgroup of  $(\mathbb{Z}_7^*, \times, 1)$ , is the group  $\mathcal{S}_2$ , which has the group elements  $\{1, 2, 4\}$ . As opposed to this, a subgroup of  $(\mathbb{Z}_7^*, \times, 1)$  cannot, for example, be formed from the set  $\{1, 2, 3\}$ , since  $2 \times 2 \equiv 4 \notin \{1, 2, 3\} \pmod{7}$ .

For a subgroup  $\mathcal{S}$  of a group  $\mathcal{B}$ , and for an element  $x \in \mathcal{B}$ , define the set  $x \odot \mathcal{S}$  as the set  $\{x \odot d : d \in \mathcal{S}\}$ . Then  $x \odot \mathcal{S}$  is said to be a *left coset* of  $\mathcal{S}$ , or simply a *coset*. There is the corresponding concept of a *right coset*  $\mathcal{S} \odot x$ , which is defined as the set  $\{d \odot x : d \in \mathcal{S}\}$ . This latter notion, is however, not used in this thesis. A coset of  $\mathcal{S}_1$  is, for example, the set  $3 \times \mathcal{S}_1 = \{3, 4\}$ . As another example, consider the coset  $5 \times \mathcal{S}_2 = \{5, 3, 6\}$  of  $\mathcal{S}_2$ .

A *field*  $\mathcal{M} = (\mathcal{W}, \odot, \oplus, \mathbf{1}, \mathbf{0})$  is a set of elements  $\mathcal{W}$ , called the *field elements* on which two binary operations  $\odot$  and  $\oplus$ , called the *field operators*, are defined. The first field operator,  $\odot$ , has the element  $\mathbf{1}$  as its identity element, whilst the second field operator,  $\oplus$ , has the element  $\mathbf{0} \in \mathcal{W}$  as its identity element. For an element  $x \in \mathcal{W}$ , one may also write  $x \in \mathcal{M}$ . The following ten axioms characterize a field in terms of the properties of its field operators and their corresponding identity elements. For all  $x, y, z \in \mathcal{M}$ ,

1. Closure under  $\odot$ :  $x \odot y \in \mathcal{M}$ . The field is said to be *closed* under the operation  $\odot$ .
2. Associativity of  $\odot$ :  $(x \odot y) \odot z = x \odot (y \odot z)$ .
3. Commutativity of  $\odot$ :  $x \odot y = y \odot x$ .
4. Identity element for  $\odot$ : there is an identity element,  $\mathbf{1} \in \mathcal{M}$  such that  $\mathbf{1} \odot x = x \odot \mathbf{1} = x$ .
5. Existence of inverses under  $\odot$ : for each element  $x \in \mathcal{M} \setminus \{\mathbf{0}\}$ , there is an inverse element in  $\mathcal{M}$ , denoted  $x^{-1}$ , such that  $x \odot x^{-1} = x^{-1} \odot x = \mathbf{1}$ .
6. Closure under  $\oplus$ :  $x \oplus y \in \mathcal{M}$ . The field is said to be *closed* under the operation  $\oplus$ .
7. Associativity of  $\oplus$ :  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ .
8. Commutativity of  $\oplus$ :  $x \oplus y = y \oplus x$ .
9. Identity element for  $\oplus$ : there is an identity element,  $\mathbf{0} \in \mathcal{M}$  such that  $\mathbf{0} \oplus x = x \oplus \mathbf{0} = x$ .
10. Existence of inverses under  $\oplus$ : for each element  $x \in \mathcal{M}$ , there is an inverse element in  $\mathcal{M}$ , denoted  $-x$ , such that  $x \oplus (-x) = (-x) \oplus x = \mathbf{0}$ .
11. Distributivity:  $x \odot (y \oplus z) = (x \odot y) \oplus (x \odot z)$  and  $(x \oplus y) \odot z = (x \odot z) \oplus (y \odot z)$ .

If  $\mathcal{W}$  is finite, then  $\mathcal{M}$  is known as a *Galois field* and is denoted  $\text{GF}(|\mathcal{W}|)$ . It can be shown (see [Rot84]) that a Galois field must contain either a prime number of elements, or a prime power number of elements.

To extend the group example above, consider the field  $\text{GF}(7) = (\mathbb{Z}_7, \times, +, 1, 0)$  with field elements  $0, 1, \dots, 6$ , and multiplication and addition modulo  $n$ . It has already been shown that all the positive elements in  $\mathbb{Z}_7$  form a group under the operation  $\times$ . It is shown similarly that the elements in  $\mathbb{Z}_7$  form a group under the operation  $+$ . Distributivity is ensured by the normal distributive semantics of multiplication over the sum of elements.

For a field  $\mathcal{M}$  with operators  $\oplus$  and  $\otimes$ , and a set  $\mathcal{W}$  of elements, if, for any  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathcal{W}$  and for any  $k, \ell \in \mathcal{M}$ , the following axioms are satisfied, then  $\mathcal{W}$  constitutes a *vector space* over  $\mathcal{M}$ , and the elements from  $\mathcal{W}$  are called *vectors*, whilst the elements of  $\mathcal{M}$  are called *scalars*.

1. Closure of  $\mathcal{W}$  under  $\oplus$ : If  $\mathbf{u}$  and  $\mathbf{v}$  are elements in  $\mathcal{W}$ , then  $\mathbf{u} \oplus \mathbf{v} \in \mathcal{W}$ .
2. Commutativity of  $\mathcal{W}$  under  $\oplus$ :  $\mathbf{u} \oplus \mathbf{v} = \mathbf{v} \oplus \mathbf{u}$ .
3. Associativity of  $\mathcal{W}$  under  $\oplus$ :  $(\mathbf{u} \oplus \mathbf{v}) \oplus \mathbf{w} = \mathbf{u} \oplus (\mathbf{v} \oplus \mathbf{w})$ .
4. Existence of an identity element in  $\mathcal{W}$  under  $\oplus$ : There is an element  $\mathbf{0}$  in  $\mathcal{W}$ , called a *zero vector* of  $\mathcal{W}$ , such that  $\mathbf{0} \oplus \mathbf{u} = \mathbf{u} \oplus \mathbf{0} = \mathbf{u}$  for all  $\mathbf{u} \in \mathcal{W}$ .
5. Existence of inverse elements in  $\mathcal{W}$  under  $\oplus$ : For each  $\mathbf{u} \in \mathcal{W}$ , there is an element  $-\mathbf{u} \in \mathcal{W}$ , called a *negative* of  $\mathbf{u}$ , such that  $\mathbf{u} \oplus (-\mathbf{u}) = (-\mathbf{u}) \oplus \mathbf{u} = \mathbf{0}$ .
6. Closure under scalar multiplication: If  $k$  is a scalar in  $\mathcal{M}$  and  $\mathbf{u}$  is an element of  $\mathcal{W}$ , then  $k \otimes \mathbf{u} \in \mathcal{W}$ .
7. Distributivity of vectors sums:  $k \otimes (\mathbf{u} \oplus \mathbf{v}) = (k \otimes \mathbf{u}) \oplus (k \otimes \mathbf{v})$ .
8. Distributivity of scalars sums:  $(k \oplus \ell) \otimes \mathbf{u} = (k \otimes \mathbf{u}) \oplus (\ell \otimes \mathbf{u})$ .



9. Associativity of scalar multiplication:  $k \otimes (\ell \otimes \mathbf{u}) = (k \otimes \ell) \otimes \mathbf{u}$ .

10. Scalar multiplication identity:  $1 \otimes \mathbf{u} = \mathbf{u}$ .

A common type of vector space  $\mathcal{V}$  over a field  $\mathcal{M}$ , has component-based vectors, which have the form  $[x_1, x_2, \dots, x_n]$  where  $x_1, x_2, \dots, x_n \in \mathcal{M}$ . Vector addition is defined as the *component-wise* application of  $\oplus$  to a pair of component-based vectors — *i.e.*,  $[x_1, x_2, \dots, x_n] \oplus [y_1, y_2, \dots, y_n] = [x_1 \oplus y_1, x_2 \oplus y_2, \dots, x_n \oplus y_n]$  — and scalar multiplication is defined as the application of  $\otimes$  to each component with the given scalar — *i.e.*,  $k \otimes [x_1, x_2, \dots, x_n] = [k \otimes x_1, k \otimes x_2, \dots, k \otimes x_n]$ . The *dimension* of such a component-based vector space is the number of components in its vectors, which is  $n$  in this case. The component-based vector space of dimension  $n$  over the field  $\mathcal{M}$  is denoted  $\mathcal{M}^n$ . To verify that such spaces are indeed vector spaces, it may be verified that the first five axioms are guaranteed by the properties of the field operator  $\oplus$  (due to the fact that it is applied in a component-wise fashion). Axiom 6 follows directly, because for a vector  $[x_1, x_2, \dots, x_n]$  and a scalar  $k \in \mathcal{M}$ ,  $k \otimes x_1, k \otimes x_2, \dots, k \otimes x_n \in \mathcal{M}$ . Axiom 7 follows because

$$\begin{aligned} k \otimes ([x_1, x_2, \dots, x_n] \oplus [y_1, y_2, \dots, y_n]) &= k \otimes ([x_1 \oplus y_1, x_2 \oplus y_2, \dots, x_n \oplus y_n]) \\ &= [k \otimes (x_1 \oplus y_1), k \otimes (x_2 \oplus y_2), \dots, k \otimes (x_n \oplus y_n)] \\ &= [(k \otimes x_1) \oplus (k \otimes y_1), \dots, (k \otimes x_n) \oplus (k \otimes y_n)] \\ &= (k \otimes [x_1, x_2, \dots, x_n]) \oplus (k \otimes [y_1, y_2, \dots, y_n]). \end{aligned}$$

Conformance to axioms 8 and 9 is shown similarly. Finally axiom 10 follows due to the component-wise application of  $\odot$ .

The Euclidean space of dimension  $n$ ,  $\mathbb{R}^n$ , is a classic example of a component-based vector space over  $\mathbb{R}$ . An important class of component-based vector spaces are those spaces over the Galois fields. The vector space of order  $n$  over the Galois field  $GF(q)$  is denoted  $\mathcal{V}(n, q)$ .

Consider the space  $\mathcal{V}(3, 3)$ , where vectors have three components, and each component is in the set  $\{0, 1, 2\}$ . Examples of operations are  $[1\ 2\ 1] \oplus [0\ 1\ 2] \equiv [1\ 0\ 0] \pmod{3}$ ,  $2[0\ 1\ 2] \equiv [0\ 2\ 1] \pmod{3}$  and  $0[0\ 1\ 2] \equiv [0\ 0\ 0] \pmod{3}$ .

## 2.4 Basic concepts in complexity theory

*Algorithmic complexity* is measured by a *time complexity* variable and a *space complexity* variable, usually expressed in terms of the input size  $n$  of the algorithm in question. These variables measure respectively the number of basic operations performed, and the memory required by the algorithm. The order of magnitude of a function, denoted by means of the symbol  $O$ , is defined as follows: Let  $f$  and  $g$  be two real-valued functions. Then  $f(n) = O(g(n))$  if there exist a  $c \in \mathbb{R}^+$  and an  $n_0 \in \mathbb{N}$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ . Informally, the order of magnitude is given by the term growing the fastest as the input size  $n$  of the algorithm increases. The function  $g$  is said to be an asymptotic upper bound for  $f$ . In the same way that the  $O$  notation denotes an asymptotic upper bound for a function, the  $\Omega$  function denotes an asymptotic *lower bound* for a function. It is defined as follows: Let  $f$  and  $g$  be two real-valued functions. Then  $f(n) = \Omega(g(n))$  if there exist a  $c \in \mathbb{R}^+$  and an  $n_0 \in \mathbb{N}$  such that  $0 \leq cg(n) \leq f(n)$  for all  $n \geq n_0$ . The function  $g$  is said to be an asymptotic lower bound for  $f$ .

An algorithm for which the order of magnitude of its time complexity is of the form  $O(n^k)$ , for some  $k \in \mathbb{R}^+$  in terms of its input size  $n$ , is called a *polynomial time* algorithm. If a problem cannot (with current knowledge) be solved by a polynomial time algorithm, it is referred to

as an *intractable* or hard problem, otherwise it is called a *tractable* problem. While the term complexity usually refers to the time complexity of an algorithm, the importance of the space complexity should not be disregarded in practical algorithm implementations.

*Decision theory* is the branch of complexity theory where the problems to be solved are interpreted as binary questions, that may be answered “yes” or “no.” Since any computational problem may be reduced to a decision problem, it is possible, without loss of generality, to consider decision theory only in the theoretical analysis of complexity issues. The class **P** is defined as the set of decision problems that can be solved by way of a polynomial time algorithm. The class **NP** constitutes the set of decision problems of which a solution can be verified in polynomial time, given some additional information. This additional information used to verify the correctness of a solution is called a *certificate*. It is clear that  $\mathbf{P} \subseteq \mathbf{NP}$ . As an example, consider the following decision problem.

**CLIQUE NUMBER**

**INSTANCE:** A graph  $\mathcal{G}$  and  $k \in \mathbb{N}$ .

**QUESTION:** Does  $\mathcal{G}$  have clique number  $\omega(\mathcal{G}) \geq k$ ?

The following proposition shows that the decision problem CLIQUE NUMBER belongs to the class **NP**, by using a clique of  $\mathcal{G}$  of order  $k$ , say  $\langle v_1, v_2, \dots, v_k \rangle$ , as certificate.

**Proposition 2.4.1** *CLIQUE NUMBER*  $\in$  **NP**.

**Proof:** The following algorithm verifies whether the induced graph  $\langle v_1, v_2, \dots, v_k \rangle_{\mathcal{G}}$  is a clique in  $\mathcal{G}$ , a graph of order  $n$ , say.

**Input:** The graph  $\mathcal{G}$  and vertex subset  $S = \{v_1, v_2, \dots, v_k\}$ .

**Step 1:** Test whether  $|E(\langle S \rangle)| = \frac{1}{2}k(k-1)$ .

**Step 2:** If true, return TRUE. Otherwise, return FALSE.

Note that Step 1 may be completed in  $O(k^2)$  time. It is therefore concluded that the algorithm will produce an output in polynomial time. ■

Let  $L_1$  and  $L_2$  be two decision problems. The problem  $L_1$  is *polynomial time reducible* to  $L_2$ , denoted  $L_1 \preceq L_2$ , if there exists a mapping  $f$  from the instances of  $L_1$  to the instances of  $L_2$ , such that:

- (a)  $f$  is computable (deterministically) in polynomial time, and
- (b)  $I$  is a solution to an instance of  $L_1$  if and only if  $f(I)$  is a solution to an instance of  $L_2$ .

In other words,  $L_1 \preceq L_2$  means that an algorithm exists for solving  $L_1$  by performing a polynomial time reduction of an instance of  $L_1$  to an instance of  $L_2$  and by solving  $L_2$ , thereby solving  $L_1$ . Informally, the algorithm that solves  $L_2$  may be seen as a “subroutine” of an algorithm that solves  $L_1$ ;  $L_2$  is therefore at least as difficult to solve as  $L_1$ . There are two classes of problems that may be defined in terms of polynomial time reductions, namely the class **NP-hard** and the class **NP-complete**. The class **NP-complete** is considered first.

**Definition 2.4.1** A decision problem  $L \in \mathbf{NP}$ -complete if

(a)  $L \in \mathbf{NP}$ , and

(b)  $L_1 \preceq L$  for all  $L_1 \in \mathbf{NP}$ . ■

In the class  $\mathbf{NP}$ , the  $\mathbf{NP}$ -complete problems may be seen as computationally the most difficult, since they are at least as difficult to solve as any other problem in  $\mathbf{NP}$ . Although it is not currently known whether the classes  $\mathbf{P}$  and  $\mathbf{NP}$  are distinct, the definitions imply that, if a decision problem  $L$  exists for which  $L \in \mathbf{NP}$ -complete and  $L \in \mathbf{P}$ , then  $\mathbf{P} = \mathbf{NP}$ . The well-known satisfiability (SAT) problem serves as an example of an  $\mathbf{NP}$ -complete problem. In order to describe this problem, the following terminology is introduced.

A *clause* is a boolean expression involving one or more boolean variables (variables with values 0 or 1) conjoined by means of the boolean operation OR. This operation is denoted by  $\vee$ , as in the example  $x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4$ , where  $\bar{x}$  denotes the complement of the boolean variable  $x$ . A boolean expression is said to be in *conjunctive normal form*, called a *cnf-formula*, if it comprises several clauses conjoined with the AND operation, denoted by  $\wedge$ . Definitions of the two boolean operations OR and AND, as well as the complement of a variable, are shown in Tables 2.1 and 2.2

$a$	$\bar{a}$
0	1
1	0

Table 2.1: Definition of the boolean complement.

$a$	$b$	$a \vee b$	$a \wedge b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 2.2: Definition of the binary operators OR ( $\vee$ ) and AND ( $\wedge$ ).

An example of a cnf-formula is  $x_1 \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3)$ . A boolean expression in conjunctive normal form is called a 3cnf-formula if each clause consists of exactly 3 variables, for example

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_3 \vee x_5 \vee \bar{x}_6).$$

A boolean expression is said to be *satisfiable* if an assignment of values for the boolean variables exists for which the expression evaluates to 1. Two satisfiability problems are stated below, and are known to be  $\mathbf{NP}$ -complete. The reader is referred to [Coo71, Lev73], or [Sip97] pp. 254–260, for proof of these results.

### Satisfiability (SAT)

**INSTANCE:** A cnf-formula  $f(x_1, \dots, x_n)$ ,  $n \in \mathbb{N}$ .

**QUESTION:** Does an assignment of values to the boolean variables  $x_1, \dots, x_n$  exist for which  $f$  evaluates to 1?

**3-Satisfiability (3SAT)**

**INSTANCE:** A 3cnf-formula  $f(x_1, \dots, x_n)$ ,  $n \in \mathbb{N}$ .

**QUESTION:** Does an assignment of values to the boolean variables  $x_1, \dots, x_n$  exist for which  $f$  evaluates to 1?

The following result follows immediately from the definition of **NP**-completeness, stated in Definition 2.4.1.

**Proposition 2.4.2** *If  $L_1 \in \mathbf{NP}$ -complete and  $L_1 \preceq L_2$ , with  $L_2 \in \mathbf{NP}$ , then  $L_2 \in \mathbf{NP}$ -complete.*

**Proof:** Due to the fact that  $L_1$  is **NP**-complete,  $L' \preceq L_1$ , for all  $L' \in \mathbf{NP}$ . The reduction  $L' \preceq L_1 \preceq L_2$ , occurs in polynomial time, because its algorithmic complexity is the sum of the complexities of the reductions between  $L' \preceq L_1$  and between  $L_1 \preceq L_2$ . Therefore, according to the definition of **NP**-completeness,  $L_2$  is **NP**-complete. ■

The problem **CLIQUE NUMBER** will be used as example, to illustrate how a decision problem may be proved to be **NP**-complete, by mapping an instance of **SAT** (a known **NP**-complete problem) in polynomial time to the decision problem in question.

Let  $\phi$  be the cnf-formula

$$\phi = (x_1^1 \vee x_2^1 \vee \dots \vee x_{p_1}^1) \wedge (x_1^2 \vee x_2^2 \vee \dots \vee x_{p_2}^2) \wedge \dots \wedge (x_1^k \vee x_2^k \vee \dots \vee x_{p_k}^k),$$

consisting of  $k$  clauses. The mapping  $f$  from  $\phi$  to a graph  $f(\phi)$  is defined as follows. The graph  $f(\phi)$  is a multipartite graph with  $k$  partite sets of cardinalities  $p_1, p_2, \dots, p_k$  respectively. The vertices of the partite set of cardinality  $p_i$  are labelled  $x_1^i, x_2^i, \dots, x_{p_i}^i$ . The edge set of the graph  $f(\phi)$  is the same as that of the corresponding complete multipartite graph, except for the edges between contradictory labels, *i.e.*, labels of which the representative variables in  $\phi$  are complements of each other. For example, if  $\phi = (x \vee y) \wedge \bar{x} \wedge (\bar{y} \vee z \vee x)$ , the mapping  $f$  would result in the graph  $f(\phi)$  shown in Figure 2.19.

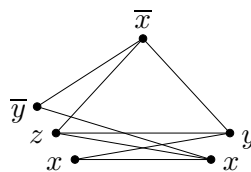


Figure 2.19: The graph  $f(\phi)$  attained from the mapping  $f$ , with  $\phi = (x \vee y) \wedge \bar{x} \wedge (\bar{y} \vee z \vee x)$ .

The following lemma shows that the mapping  $f$  is sufficient to solve an instance of **SAT** as an instance of **CLIQUE NUMBER**. The proof is similar to that in [Sip97], pp. 251–253.

**Lemma 2.4.1** *Let  $\phi$  be a cnf-formula with  $k$  clauses and let  $f$  be the mapping defined above. Then the graph  $f(\phi)$  contains a  $k$ -clique (*i.e.*,  $\omega(\mathcal{G}) \geq k$ ) if and only if  $\phi$  is satisfiable.*

**Proof:** Suppose  $\phi$  has a satisfying assignment of boolean variables for  $f$ . In that satisfying assignment, at least one variable in each clause is assigned the value 1. In each clause of  $\phi$ , select a variable with an assignment of 1 and consider the vertices of  $\mathcal{G}$  corresponding to these variables under the mapping  $f$ . The number of vertices selected is  $k$ , since  $\phi$  consists of  $k$  clauses. Each

vertex–label is in a different clause and no two of these are complements of each other, since all have an assignment of 1. Therefore every pair of these selected vertices are adjacent. This selection forms a clique in  $\mathcal{G}$  of order  $k$  and hence  $\omega(\mathcal{G}) \geq k$ .

Suppose  $\omega(\mathcal{G}) \geq k$ . Then  $\mathcal{G}$  contains a clique of order  $k$ . Consider the variables in  $\phi$  corresponding to the vertices of such a clique. It follows that no two of these variables are in the same clause and no two are complements of each other, since otherwise the corresponding vertices would not be adjacent in  $\mathcal{G}$ . Therefore each clause contains exactly one of the selected variables. Consider an assignment to the boolean variables in  $\phi$  where every variable corresponding to the clique–vertices are assigned the value 1, and the others 0. Such an assignment is always possible, since none of the clique–variables are contradictory, and  $f$  evaluates to 1 for such an assignment. It follows that  $\phi$  is satisfiable. ■

The following theorem may now be used to show that the decision problem CLIQUE NUMBER is **NP**–complete.

**Theorem 2.4.1**  $SAT \preceq CLIQUE\ NUMBER$

**Proof:** A polynomial time algorithm outline is presented to solve SAT for cnf–formulas with  $k$  clauses,  $k \in \mathbb{N}$ , which employs CLIQUE NUMBER as a subroutine.

**Input:** A cnf–formula,  $\phi$ , with  $k$  clauses.

**Step 1:** Determine the corresponding multipartite graph  $f(\phi)$  with  $k$  partite sets, where  $f$  is as defined in the above discussion.

**Step 2:** If  $f(\phi)$  contains a  $k$ –clique, return TRUE. Otherwise, return FALSE. ■

It is known that  $SAT \in \mathbf{NP}$ –complete [Sip97]. From Theorem 2.4.1 it now also holds that  $SAT \preceq CLIQUE\ NUMBER$ , with  $CLIQUE\ NUMBER \in \mathbf{NP}$ , according to Proposition 2.4.1. Note that Step 1 in the proof of Theorem 2.4.1 may be completed in polynomial time, since a cnf–formula  $\phi$  with  $k$  clauses may be determined by searching the edges of the graph as each vertex is considered. It follows from Proposition 2.4.2 that  $CLIQUE\ NUMBER \in \mathbf{NP}$ –complete.

The definition of the class **NP**–hard is more relaxed than the definition of the class **NP**–complete. In fact, all **NP**–complete problems are also **NP**–hard.

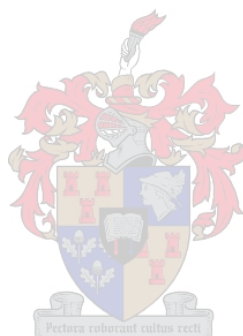
**Definition 2.4.2** A problem  $L$  is **NP**–hard if  $L_1 \preceq L$  for all  $L_1 \in \mathbf{NP}$ . ■

Multidimensional optimization problems are often **NP**–hard. In such problems, a function  $f$  must be minimized over a vector of variables  $\mathbf{v}$ . One can construct a decision problem by asking whether a  $\mathbf{v}$  exists such that  $f(\mathbf{v}) \leq k$ , where  $k$  is given as an input. Of course, this does not convert the optimization problem as a whole into a decision problem, since the problem of determining the smallest  $k$  for which a  $\mathbf{v}$  exists cannot be phrased as a decision problem. If the above–mentioned decision problem is **NP**–complete, a solution to the optimization problem will provide a solution to any problem in **NP**.

This section has only skimmed the surface of complexity theory. The reader is referred to [Sip97], pp. 223–270, for a more extensive discussion on the topic.

## 2.5 Chapter summary

In this chapter, the basic concepts of graph theory, group theory and complexity theory, relevant to this thesis, were introduced for the benefit of the reader. The appropriate graph theoretic concepts were discussed in § 2.1.1–2.1.6 and the relevant notions in topology were described in § 2.2. Algebraic definitions were given in § 2.3, and the last section, § 2.4, familiarised the reader with the most basic concepts in complexity theory.



# Chapter 3

## Crossing Numbers

... a good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher.

— *Bertrand Russell (1872–1970)*

The study of the crossing number of a graph in the plane forms part of a larger field of study known as topological graph theory. This field is primarily concerned with the representation of graphs in spaces that are locally 2-dimensional. All discussions in this thesis will, however, centre around the plane,  $\mathbb{R}^2$ , for simplicity; although the text has been written so that the reader may substitute any locally 2-dimensional space for  $\mathbb{R}^2$ .

The basic operation of interest, is a formalization of the notion of a *drawing* of a graph, which is a mapping of its vertices to points in  $\mathbb{R}^2$ , and of its edges to continuous curves in  $\mathbb{R}^2$ , so that (a) images of edges intersect a finite number of times, and (b) they are nowhere tangential. This is the topic of § 3.1.1 and § 3.1.2.

Certain commonly used notions, derived from the notion of a graph drawing, such as *embeddings* (drawings of graphs without intersecting edges) and *book drawings* (drawings of graphs where vertices fall on the “spine of a book,” and each edge is drawn on only a single page) are also considered, as they are important in the modern theory of the crossing number of a graph. They may be found in § 3.1.3.

It is in the context of these definitions of drawings, that precise definitions of the various kinds of important crossing numbers of a graph are given.

### 3.1 Drawings and embeddings

The concept of a graph drawing must be made precise, if one is to investigate the crossing number problem. It cannot be overstated just how important it is to be completely specific and unambiguous about the definition of a graph drawing, since various researchers have apparently considered a number of subtly different interpretations, due to a lack of consensus of what exactly constitutes a “valid” graph drawing.

#### 3.1.1 From vertices to points, and edges to curves

The simplest way to map a vertex into the plane, is to associate it with a single point in  $\mathbb{R}^2$ . Associating a vertex with a set (such as a disc) in  $\mathbb{R}^2$  gives no advantage over the single point

definition, and certainly places restrictions over the exterior bounds into which the vertex may be embedded.



Figure 3.1: Crossings in self-crossing edges can always be removed. The graph in (a) contains an edge  $e$  that crosses itself. This self-crossing has been removed in (b).

If a curve representing an edge (of a graph) crosses itself (in a drawing of the graph) as does the edge  $e$  in Figure 3.1(a), such a self-crossing may be removed without introducing additional crossings into the drawing. This may be achieved by “cutting off” the loop that forms at the crossing, as is shown in Figure 3.1(b).

Therefore, it makes sense to require that the (curve) image of an edge does not cross itself in what will be considered valid graph drawings. Thus the notion of a *simple curve*.

**Definition 3.1.1** A simple curve  $c$  in  $\mathbb{R}^2$  is a continuous injection  $c : [0, 1] \rightarrow \mathbb{R}^2$ . In the terminology of topology, it is homeomorphic to  $[0, 1]$ . The set of all simple curves in  $\mathbb{R}^2$  is denoted  $\mathcal{L}_{\mathbb{R}^2}$ . ■

### 3.1.2 Definitions of graph drawings

The notion of a graph drawing may be formalized as follows.

**Definition 3.1.2** A drawing  $\phi_{\mathbb{R}^2}(\mathcal{G}) = (\phi_{\mathbb{R}^2}^{(v)}(\mathcal{G}), \phi_{\mathbb{R}^2}^{(e)}(\mathcal{G}))$  of a graph  $\mathcal{G} = (V, E)$  in  $\mathbb{R}^2$  is an injection  $\phi_{\mathbb{R}^2}^{(v)} : V(\mathcal{G}) \rightarrow \mathbb{R}^2$  and a function  $\phi_{\mathbb{R}^2}^{(e)} : E(\mathcal{G}) \rightarrow \mathcal{L}_{\mathbb{R}^2}$ . For every  $e = \{v_i, v_j\} \in E(\mathcal{G})$  and  $\phi_{\mathbb{R}^2}^{(e)}(e) = c$ , it is either the case that  $c(0) = \phi_{\mathbb{R}^2}^{(v)}(v_i)$  and  $c(1) = \phi_{\mathbb{R}^2}^{(v)}(v_j)$ , or that  $c(0) = \phi_{\mathbb{R}^2}^{(v)}(v_j)$  and  $c(1) = \phi_{\mathbb{R}^2}^{(v)}(v_i)$ . Finally, the interior of the edge  $e$  is defined as  $\phi_{\mathbb{R}^2}^{(e)}(\tilde{e}) = c \setminus \{c(0), c(1)\}$ . For each edge  $e \in E(\mathcal{G})$  and for each vertex  $v \in V(\mathcal{G})$ , it must hold<sup>1</sup> that  $\phi_{\mathbb{R}^2}^{(v)}(v) \cap \phi_{\mathbb{R}^2}^{(e)}(\tilde{e}) = \emptyset$ . A maximally connected subset of  $\mathbb{R}^2 \setminus \phi_{\mathbb{R}^2}(\mathcal{G})$  is called a region of  $\phi_{\mathbb{R}^2}(\mathcal{G})$ . ■

As a matter of convenience and at the cost of being somewhat imprecise, vertices and edges of a graph  $\mathcal{G}$  will be identified and referred to by their images under  $\phi_{\mathbb{R}^2}(\mathcal{G})$ , when no ambiguity results. Furthermore, when it is necessary to qualify a drawing of a vertex or of an edge, the superscripts (v) and (e) differentiating the drawing functions will be discarded when it is clear what the type of the operand is.

The subscript  $\mathbb{R}^2$  used in the notation introduced above makes it explicit that drawings may be defined for topological spaces that are locally 2-dimensional, other than  $\mathbb{R}^2$ . However, since

<sup>1</sup>This means that only interiors of edges may intersect one another. Because the topic of this thesis is about edge intersections, it will be understood that a crossing between a pair of edges will, in fact, be a crossing between the interiors of the edges. In the same vein the notation will be abused, and the tilde above the edge will be omitted after this chapter.



this thesis focuses on drawings in  $\mathbb{R}^2$ , the subscript  $\mathbb{R}^2$  will be omitted from the notation in the remainder of the thesis, unless the resolution of ambiguity demands its presence. As a further abuse of notation, a drawing  $\phi(\mathcal{G})$  will simply be denoted  $\phi$ , when it is clear from the context that the drawing is of a graph  $\mathcal{G}$ .

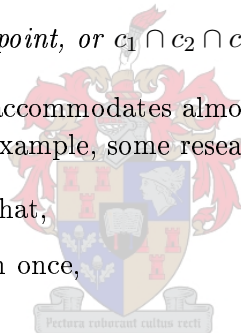
The definition of a drawing of a graph still leaves much to be desired. It allows for edges to have pairwise arbitrarily many mutual intersections, and these intersections need not necessarily lead to what might be considered true “crossings.” For this reason, a far narrower definition than Definition 3.1.2 is required, in order for drawings to be useful. Székely [Szé04] gives such a definition for what is called a drawing in “normal form:”

**Definition 3.1.3** *A drawing  $\phi(\mathcal{G})$  of a graph  $\mathcal{G}$  is said to be in normal form if, for any three curves  $c_1$ ,  $c_2$  and  $c_3$  in  $\mathbb{R}^2$  corresponding to edges in  $E(\mathcal{G})$ ,*

1. *any pair of edges cross each other a finite number of times, so that the number of disjoint maximal subsets in  $c_1 \cap c_2$  (and also,  $c_2 \cap c_3$  and  $c_1 \cap c_3$ ) is finite,*
2. *curves are nowhere tangential — if one side of a curve is seen as “left” and the other as “right”, then if  $c_1$  and  $c_2$  share a point,  $c_1$  is present both on the left and right of  $c_2$ ; more formally, if  $c_1$  and  $c_2$  share a point, then for any  $\varepsilon > 0$ , the disc  $U$  with centre  $p$  and radius  $\varepsilon$  is such that  $U \setminus c_2$  has two connected components, both intersecting  $c_1$ ,*
3. *no three edges share a crossing point, or  $c_1 \cap c_2 \cap c_3 = \emptyset$ . ■*

This definition is well justified, as it accommodates almost all the definitions of graph drawings used by researchers in the field. For example, some researchers additionally assume that either

- i. adjacent edges do not cross or that,
- ii. no pair of edges cross more than once,
- iii. or both.



Garey and Johnson [GJ83] did not impose (i), although they assumed (ii). On the other hand, Tutte [Tut70] assumed (i), but he assumed that edges may cross multiple times (this is, in fact, integral to the theory developed in [Tut70]; his theory is discussed in the next chapter).

The significance of graph drawings in normal form is that all graph drawings with a finite number of edge intersections (*i.e.*, all drawings for which the first property in Definition 3.1.3 holds) may be converted to drawings in normal form without increasing the total number of pairwise edge crossings.

1. If  $c_1 \cap c_2$  is not finite, then  $c_1$  and  $c_2$  must share subsections of curves. For each such shared subsection, this violation can be avoided by letting  $c_1$  run close alongside  $c_2$  where they would have intersected and then having only a single point of intersection, as shown in Figure 3.2. A shared curve subsection is present in part (a) of the figure, whilst this has been removed in part (b).
2. Since the number of intersections of two curves  $c_1$  and  $c_2$  must be finite,  $c_1$  and  $c_2$  can only be tangential by sharing single points. Since they then violate the requirement that they should be present on each other’s left- and right-hand sides respectively, the intersections can be removed by simply shifting  $c_1$  away from  $c_2$  at the point of intersection, as shown in Figure 3.3. The tangential intersections in Figure 3.3(a) have been removed in Figure 3.3(b).

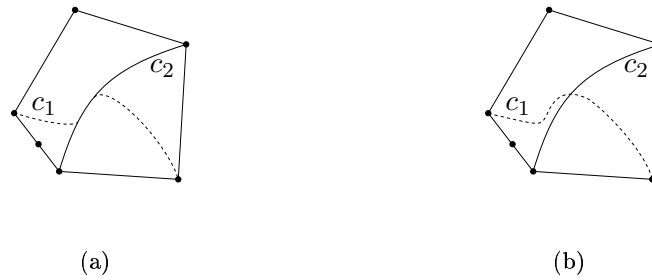


Figure 3.2: Edge intersections that involve subsections of edges may be removed by letting edges run close to one another. The edges  $c_1$  and  $c_2$  intersect one another at an infinite number of points in (a), whilst they only intersect each other at one point in (b).



Figure 3.3: Tangential edge intersections may be removed simply by moving curve sections at the intersections far enough apart. The edges  $c_1$  and  $c_2$  have a tangential intersection in (a). This intersection has been removed in (b) by moving the edges away from one another.

3. If three or more curves have a mutual crossing, as is the case for the four curves  $c_1$ ,  $c_2$ ,  $c_3$  and  $c_4$  in Figure 3.4(a), then the construction in Figure 3.4(b) can be applied to remove the mutual crossing. Four crossings were chosen to show that this method is easy to generalise — just as  $c_4$  is “looped” around the crossing of  $c_1$  and  $c_2$ , and  $c_3$  around  $c_4$ , other edges may be “looped” around  $c_3$ .



Figure 3.4: Removal of more than a single crossing point at the same point of  $\mathbb{R}^2$ . The edges  $c_1$ ,  $c_2$ ,  $c_3$  and  $c_4$  intersect at a common point in (a), whilst the edges  $c_3$  and  $c_4$  have been redrawn in (b), so as to “loop around” the intersection of  $c_1$  and  $c_2$ .

A deficiency of the methods described above, is that it has not been shown how the drawings of edges may be adjusted so that no crossings with other edges are inadvertently introduced. Some basic definitions relating to sets in  $\mathbb{R}^2$  are reviewed here, before a formal technique for ensuring that the above mentioned problem does not arise, is given.

### Open sets, closed sets and borders

The concept of a *closed set* in  $\mathbb{R}^2$  is a generalization of the concept of a *closed interval* in  $\mathbb{R}$ . A closed interval  $[a, b]$  in  $\mathbb{R}$  is the set  $\{x : a \leq x \leq b, a, x, b \in \mathbb{R}\}$ . An example of an closed set in  $\mathbb{R}^2$  is a closed rectangle of the form  $[a, b] \times [c, d]$ , where  $a < b$  and  $c < d$ , and where  $\times$  denotes the Cartesian product of the two sets.

Analogous to the way in which the concept of a closed set in  $\mathbb{R}^2$  generalizes the concept of a closed interval in  $\mathbb{R}$ , the concept of an *open set* in  $\mathbb{R}^2$  is a generalization of the concept of an *open interval* in  $\mathbb{R}$ . An open interval  $(a, b)$  in  $\mathbb{R}$  is the set  $\{x : a < x < b, a, x, b \in \mathbb{R}\}$ . An open rectangle in  $\mathbb{R}^2$  has the form  $(a, b) \times (c, d)$ , where  $a < b$  and  $c < d$ .

Note that the definitions above imply that it is possible to construct a set that is neither open nor closed. Consider, for example, the set  $(a, b]$ , which is defined as  $\{x : a < x \leq b, a, x, b \in \mathbb{R}\}$ .

For a closed set  $V \in \mathbb{R}^2$ , let  $U$  be the largest open set contained in  $V$  — *i.e.*, let  $U$  be such that there does not exist a set  $U' \in \mathbb{R}^2$  satisfying  $U \subset U' \subset V$ . The *border* of a closed set  $V$ , denoted  $\delta V$ , is the set  $V \setminus U$ . More informally, the border of a closed set is the closed curve surrounding its largest contained open set. The border of the closed rectangle above may be obtained by the removal of its contained open rectangle (also described above) and consists of the union of the four lines that are described by the sets  $\{(x, b) : a \leq x \leq c\}$ ,  $\{(x, d) : a \leq x \leq c\}$ ,  $\{(a, y) : b \leq y \leq d\}$  and  $\{(c, y) : b \leq y \leq d\}$ .

### Adjusting the drawings of edges

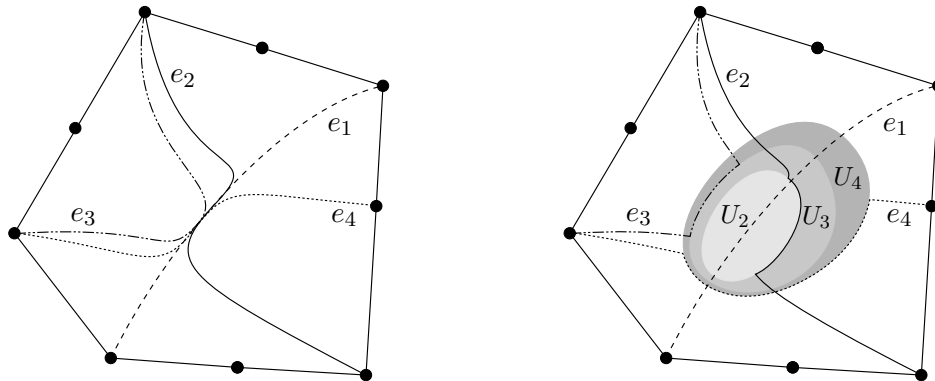
Now, in a drawing  $\phi$  of a graph, suppose that the images of  $n$  edges,  $\phi(e_1), \phi(e_2), \dots, \phi(e_n)$  intersect one another. The types of intersections possibly include all three types of intersections described above — shared subsections of edges, tangential intersections and multiple crossings at a single point. Each of the drawings  $\phi(e_i)$ ,  $i \in \{2, \dots, n\}$  will be replaced by drawings  $\phi'(e_i)$ ,  $i \in \{2, \dots, n\}$ , so that the final graph drawing  $\phi'$  will be in normal form. Let  $x_{i,j} = \phi(e_i) \cap \phi(e_j)$ , then the set  $C = \bigcup_{1 \leq i < j \leq n} x_{i,j}$  contains a number of disjoint subsets. Let  $c$  be such a maximal (connected) subset of  $C$  and suppose, without loss of generality, that each of the  $n$  images of edges intersects  $c$ . Note that  $c$  may be a larger set than the intersection  $\phi(e_1) \cap \phi(e_2) \cap \dots \cap \phi(e_n)$  of all  $n$  edge images.

Consider a collection  $U_2, U_3, \dots, U_n$  of  $n - 1$  closed sets with the property that  $c \subset U_2 \subset U_3 \subset \dots \subset U_n$  and with the property that  $U_n$  does not intersect any images of edges besides  $\phi(e_1), \phi(e_2), \dots, \phi(e_n)$  and that it also does not intersect any of the other disjoint subsets in  $C$ . This is always possible, because it is always possible to find a point  $p$  between any pair of points in  $\mathbb{R}^2$ . Thus, the border of  $U_n$  may be drawn in such a fashion as to fall between all points comprising  $c$  and all the forbidden surrounding points (of images of other edges and other subsets in  $C$ ). Likewise, the border of  $U_{n-1}$  may be drawn so as to fall between  $c$  and the border of  $U_n$  and so on. To render this method functional, it is also important to ensure that the border of  $U_i$  intersects  $\phi(e_i)$  at only two points (*i.e.*, where  $\phi(e_i)$  “enters”  $U_i$  and where it “leaves”  $U_i$ ;  $\phi(e_i)$  should not run along the border of  $U_i$ ).

For each edge  $e_i$  in the set of  $n$  edges, let  $L_i = \phi(e_i) \setminus U_i$  and  $P_i = \phi(e_i) \cap \delta U_i$ . The set  $L_i \cup P_i$  corresponds to the drawing  $\phi(e_i)$  truncated at the two points in  $P_i$  on the border of  $U_i$ . There are two paths,  $p_1$  and  $p_2$ , in  $\delta U_i$  that may be used to join the points in  $P_i$  — these are the two maximal disjoint subsets in  $\delta U_i \setminus P_i$ . One path corresponds to a path followed clockwise “around”  $U_i$  and the other path followed anti-clockwise “around”  $U_i$ . The path  $p_i$ ,  $i \in \{1, 2\}$ ,

whose selection will cause the fewest number of crossings in the edge drawing  $L_i \cup P_i \cup p_i$ , is chosen, and  $\phi'(e_i)$  is defined to be this drawing.

An example of the application of this method may be seen in Figure 3.5. In Figure 3.5(a), the edges  $e_1$ ,  $e_2$  and  $e_4$  cross one another and they also share subsections of curves in their intersections. The edge  $e_3$  intersects  $e_1$ ,  $e_2$  and  $e_4$  in a tangential fashion. All of the invalid intersections have been removed by the transformation, as may be seen in Figure 3.5(b).



(a)  $e_1$ ,  $e_2$  and  $e_4$  cross one another and  $e_3$  is involved in tangential intersections.

(b) All invalid intersections have been removed.

Figure 3.5: An illustration of the general technique for the removal of invalid edge intersections.

The appeal of graph drawings in normal form with the additional requirements (i) and (ii) is that, for all drawings achieving the minimum total number of crossings of their graphs, these properties do, in fact, hold. Furthermore, from a computational point of view, they reduce the number of operations required to find or approximate the crossing number of a graph (since it is not necessary to keep track of multiple crossings between pairs of edges).

Thus, again a definition from Székely [Szé04]:

**Definition 3.1.4** *A drawing  $\phi$  of a graph  $\mathcal{G}$  is said to be in single-cross normal form (nice in Székely’s terminology) if, in addition to being in normal form, it satisfies the following properties for any two edges  $e, f \in E(\mathcal{G})$ :*

1.  $|\phi(\tilde{e}) \cap \phi(\tilde{f})| = 0$  if  $e, f \in E(\mathcal{G})$  are adjacent — Adjacent edges never cross.
2.  $|\phi(\tilde{e}) \cap \phi(\tilde{f})| \leq 1$  for all  $e, f \in E(\mathcal{G})$  — Edges cross each other at most once. ■

Multiple crossings between two non-adjacent edges may be removed using the following procedure:

Suppose two edges  $e$  and  $f$ , that do not share an end vertex, cross each other  $t$  times. View each edge as the union of  $t + 1$  curves, where each curve is a section between two crossing points, or between a vertex and a crossing point. Hence  $e = a_1 \cup a_2 \cup a_3 \cup \dots \cup a_t \cup a_{t+1}$  and  $f = b_1 \cup b_2 \cup b_3 \dots \cup b_t \cup b_{t+1}$ . By exchanging every second curve between  $e$  and  $f$  (leaving  $a_{t+1}$  and  $b_{t+1}$  intact, since these curves are incident to different vertices) to obtain  $e = a_1 \cup b_2 \cup a_3 \cup \dots \cup b_t \cup a_{t+1}$

and  $f = b_1 \cup a_2 \cup b_3 \cup \dots \cup a_t \cup b_{t+1}$  when  $t$  is even and  $e = a_1 \cup b_2 \cup a_3 \cup \dots \cup a_t \cup a_{t+1}$  and  $f = b_1 \cup a_2 \cup b_3 \cup \dots \cup b_t \cup b_{t+1}$  when  $t$  is odd, and by ensuring that tangential intersections are removed, the total number of crossings between  $e$  and  $f$  will be reduced to 0 and 1 respectively. This is true, because the configurations  $a_k \cup a_{k+1}$  and  $b_k \cup b_{k+1}$  cause a crossing, which is removed by swapping either  $a_k$  and  $b_k$ , or  $a_{k+1}$  and  $b_{k+1}$ .

A visual representation of this method is shown in Figure 3.6. In part (a) of the figure there are three crossings between  $e$  and  $f$ , and this is reduced to a single crossing in part (b).

If  $e$  and  $f$  share an end vertex, then all mutual crossings may be removed, since, unlike the previous case, the curves  $a_{t+1}$  and  $b_{t+1}$  may also be swapped if necessary, since they share an end vertex.

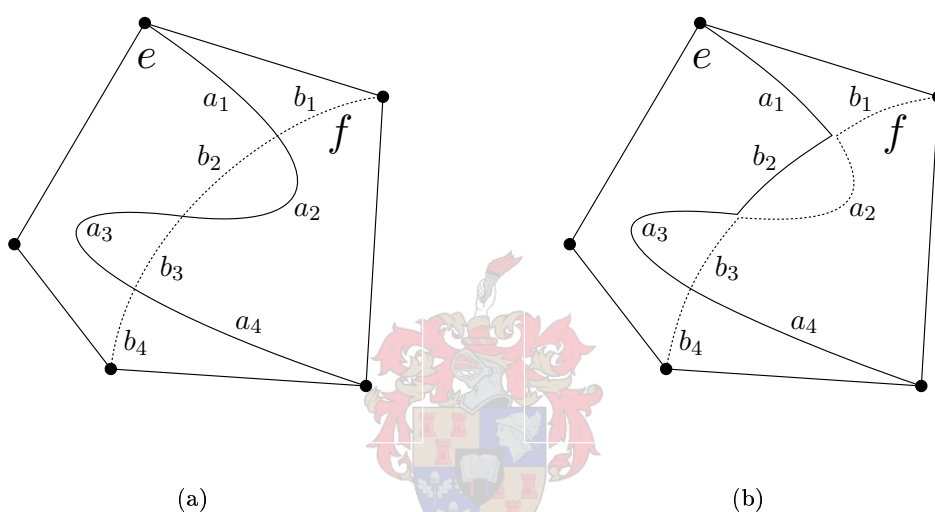


Figure 3.6: A procedure for reducing the number of mutual crossings between a pair of non-adjacent edges. The edges  $e$  and  $f$  cross each other multiple times in (a). The reassignment of subsections of the curves representing  $e$  and  $f$  results in a situation where  $e$  and  $f$  cross one another at most once, as shown in (b).

This procedure may be applied repeatedly, until there are no more opportunities for improvement. It is guaranteed to converge, since it always reduces the total number of crossings, and so the end result will always be a drawing in single-cross normal form.

### 3.1.3 Important variants of drawings

Besides drawings in normal form and in single-cross normal form, which still permit a large degree of freedom with respect to the placement of vertices and formation of edges, a number of other, restricted types of drawings occur in the literature.

#### 3.1.3.1 Book drawings

An  $n$ -page book is the union of  $n$  half-planes. The half-planes intersect exactly on their finite boundaries, and this line is called the “spine” of the book. The half-planes are the “pages” of the book.

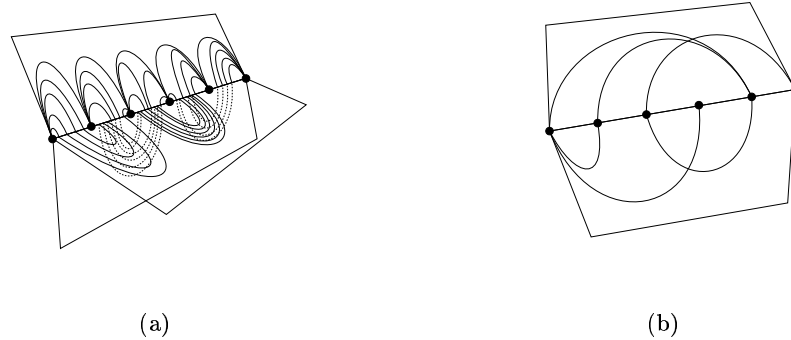
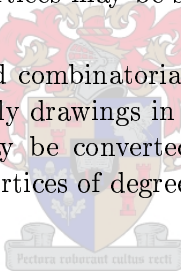


Figure 3.7: A topological and a combinatorial book embedding.

There are two variants of book drawings — combinatorial and topological book drawings. In both, the vertices of a graph are placed on the spine of a book. In a topological book drawing, edges are allowed to cross over the spine (although they may not be tangential with the spine), but in combinatorial book drawings, each edge must be drawn wholly on a single page of the book. Both topological and combinatorial book drawings are in normal form. A three–page topological book drawing of  $\mathcal{K}_6$  may be seen Figure 3.7(a), and a two–page combinatorial book drawing of an arbitrary graph on six vertices may be seen in part (b) of the figure.

It follows that one–page topological and combinatorial book drawings are equivalent, and that two–page topological drawings are merely drawings in normal form in  $\mathbb{R}^2$ . Furthermore, a topological book drawing of a graph  $\mathcal{G}$  may be converted to a combinatorial book drawing of a subdivision of  $\mathcal{G}$  by inserting artificial vertices of degree two into edges where they cross over the spine.



The appeal of combinatorial book drawings stems from the fact that books provide a simple combinatorial framework for computing the crossing number of a graph, but what is more compelling, is that it is very easy to enumerate the different book drawings of a graph.

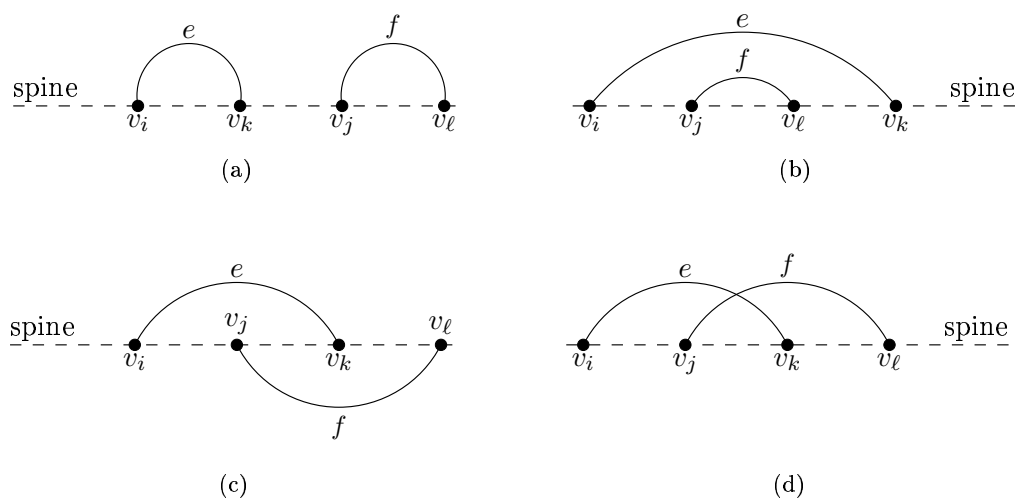


Figure 3.8: The possible configurations of two edges in a book.

### Computation of the crossing number for a given drawing

For a graph  $\mathcal{G}$ , and a vertex  $v_i \in V(\mathcal{G})$ , let  $i$  be the index of  $v_i$ , and let the vertices be placed in increasing order of their indices on the spine of a book. Any four vertices  $v_i, v_j, v_k$  and  $v_\ell$  are said to be *alternating* on the spine if  $i < j < k < \ell$ , and they are said to be *non-alternating* otherwise.

For two edges  $e = \{v_i, v_k\}, f = \{v_j, v_\ell\} \in E(\mathcal{G})$ , where  $i < k, j < \ell$  and  $i < j$ , the crossing possibilities are shown in Figure 3.8. In parts (a) and (b) of the figure, the four vertices are not alternating, and it is clear that the edges  $e$  and  $f$  can never cross. In part (c), although they alternate, the edges  $e$  and  $f$  are on different pages, and consequently, they cannot cross. However, if the vertices are alternating, and the edges  $e$  and  $f$  are on the same page, then a crossing must occur, and this configuration is shown in part (d) of the figure. As a matter of convenience, when the vertices of  $e$  alternate the vertices of  $f$ , the edges are themselves said to be alternating.

### Enumeration of different book drawings for a given graph

Clearly, the only factors that have an impact on whether two edges cross, are:

1. the relative orderings of their end vertices — alternating or not,
2. and the pages on which the edges are drawn.

Thus, by permuting the vertices on the spine, and by choosing different pages for edges, different drawings may be generated easily.

Since no such benefit is provided by the topological variant, only combinatorial book drawings will be considered in this thesis. Hence,  $n$ -page combinatorial book drawings will simply be called  $n$ -page book drawings, as the distinction is no longer important. It should be noted that combinatorial drawings are always in single-cross normal form, but that the topological drawing arrived at by removing the artificial subdivision vertices (that were inserted to convert a topological drawing to a combinatorial drawing) from a graph is not necessarily in single-cross normal form, since each subdivision edge may be crossed, resulting in possibly more than a single crossing on the unsubdivided edge.

Some interesting results regarding the approximation of the crossing number of a graph in  $\mathbb{R}^2$ , in terms of one-page layouts have been found, and these results have also been improved for two-page layouts (as will be discussed in the next chapter). Therefore, book drawings play an important role in the approximation of the crossing number of a graph in  $\mathbb{R}^2$ .

#### 3.1.3.2 Circular drawings

Circular drawings (the author's terminology) are sometimes used in the literature, and it is important to note that such drawings are equivalent to two-page book layouts. To illustrate the equivalence, in the former, vertices are drawn around the length of the circumference of a circle in the order given by the spine of a book layout, and edges are drawn either wholly inside the circle, or wholly outside of it (the inside of the circle corresponds, say to the upper page, and the outside to the lower page). It is shown in Figure 3.9 how this equivalence may be seen by “folding” out a circular arrangement. A point worth observing is the irrelevance of the side around which an edge on the outside of the circle is drawn (*i.e.* whether it passed around the left

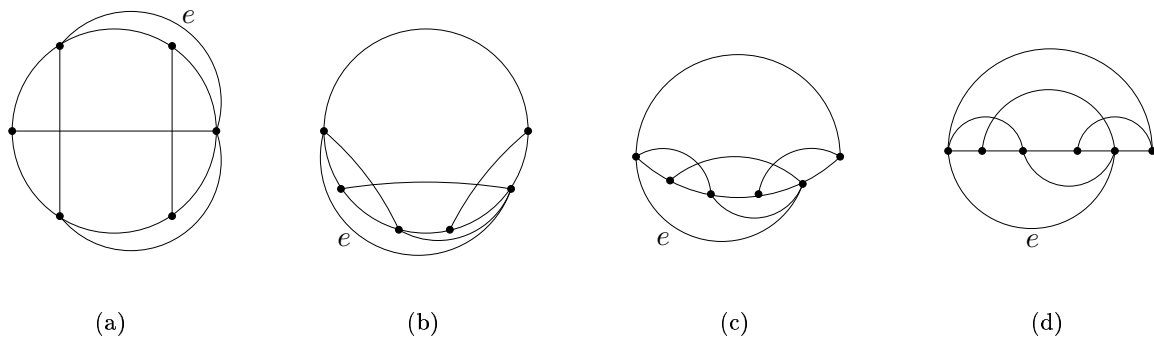


Figure 3.9: A book layout on two pages is equivalent to a circular layout.

or around the right of the circle) — this is true, because logically, two alternating edges must cross, and two non-alternating edges need never cross. However, when this “folding” out occurs, all edges which are drawn on the outside of the circle need to be drawn underneath the circle, since by the “folding” operation, they become the edges on the lower page. Hence, the edge  $e$  in Figure 3.9(a) is drawn around the other side of the circle in Figure 3.9(b), where it remains for the rest of the transformation.

### 3.1.3.3 Embeddings

An embedding is simply a drawing of a graph in which no pair of edges contains any mutual intersections. The simplest example is a plane drawing of a planar graph. The theory of embeddings is important, since it provides a vocabulary within which to describe certain types of drawings, or qualities of drawings.

It is often convenient, for example, to refer to regions of a graph drawing, as defined in Definition 3.1.2, although strictly, a region is bounded only by vertices and edges, and not by intersections of edges. However, since any drawing of a graph in normal form can be converted to an embedding (of another graph), by inserting artificial vertices of degree four at intersections of edges, it is convenient to loosen the definition of a region in this way.

Therefore, embeddings are not studied in their own right in this thesis, but they will be used for convenience when necessary, without explicit mentioning.

### 3.1.3.4 Other types of drawings

Other types of restricted drawings are considered in the literature. These drawings lead to new crossing number problems, which have produced large bodies of work. For this reason they are mentioned, but not discussed at length, so as to remain within the scope of this thesis.

*Rectilinear drawings* require that all drawings of edges are straight lines. This gives the crossing number problem a more combinatorial flavour, making it in some cases more manageable, and may lead to easily implementable algorithms (be they exact or heuristic). Since all edges are required to be straight, it is only possible for a pair of edges to have a single mutual crossing. Therefore, rectilinear drawings are necessarily drawings in single-cross normal form.

*Layered drawings*, or more specifically, *k-layered drawings* are such that all vertices may be placed on  $k$  distinct levels, where the vertices on level  $i$  are only adjacent to vertices on layers  $i - 1$  and  $i + 1$ . Edges may only be drawn as straight lines. Of course, a graph requires a certain



structure to be drawn in a layered fashion, and it is called a  $k$ -layered graph if it permits a  $k$ -layered drawing. According to this definition, bipartite graphs are 2-layered (2-layered drawings are sometimes called *bipartite drawings*). An example of a 3-layered drawing, of a graph with 15 vertices, and 5 vertices on each layer, may be seen in Figure 3.10. However, there is no requirement that the same number of vertices be located on each layer.

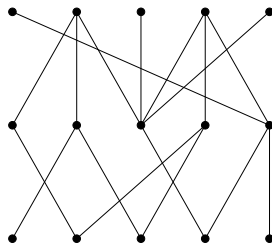


Figure 3.10: A 3-layered drawing of a graph on 15 vertices.

## 3.2 Defining the crossing number of a graph

The crossing number of a graph in  $\mathbb{R}^2$  is the smallest number of crossings with which the graph can be drawn, when considering all drawings in single-cross normal form of the graph in  $\mathbb{R}^2$ . Formally, this is defined as follows.

**Definition 3.2.1** For any drawing  $\phi$  in single-cross normal form, of a graph  $\mathcal{G}$ , define an indicator function

$$\chi(e, f, \phi) = \begin{cases} 1 & \text{if } |\phi(\tilde{e}) \cap \phi(\tilde{f})| = 1 \text{ and } e \neq f, \\ 0 & \text{otherwise.} \end{cases}$$

The crossing number of  $\mathcal{G}$ , realized by the drawing  $\phi$ , is

$$\nu_\phi(\mathcal{G}) = \frac{1}{2} \sum_{e, f \in E(\mathcal{G})} \chi(e, f, \phi), \quad (3.1)$$

whilst the crossing number of  $\mathcal{G}$  in  $\mathbb{R}^2$  is defined as

$$\nu_{\mathbb{R}^2}(\mathcal{G}) = \min_{\phi(\mathcal{G})} \{ \nu_\phi(\mathcal{G}) \}. \quad \blacksquare$$

The factor of  $1/2$  in (3.1) is due to the fact that the sum is taken over all ordered pairs of edges, and therefore each is counted twice.

Again, since the focus in this thesis is on the crossing number of a graph in  $\mathbb{R}^2$ , the crossing number of  $\mathcal{G}$  in  $\mathbb{R}^2$ ,  $\nu_{\mathbb{R}^2}(\mathcal{G})$  will be denoted simply by  $\nu(\mathcal{G})$ . This is consistent with the notation in much of the literature on crossing numbers, although some authors use the notation  $\text{cr}(\mathcal{G})$  instead of  $\nu(\mathcal{G})$ .

### 3.2.1 Crossing numbers based on other types of drawings

Pach and Tóth [PT98] have noted that the crossing number of a graph  $\mathcal{G}$  is often just defined as “the minimum number of edge crossings in any drawing of  $\mathcal{G}$  in the plane” ([BL84]). As Pach and Tóth emphasise, this may be interpreted in a number of ways. Clearly, it does not stipulate whether adjacent edges may cross each other. Even worse, it may be interpreted to mean either:

- The minimum number of pairs of edges that are involved in crossings at all, although the number of times two edges cross each other is irrelevant, or
- The minimum number of pairs of edges that are involved in crossings, where a pair of edges may cross each other at most once.

At first, these objections seem to be without merit, since the method discussed in § 3.1.2 may be used to transform any drawing in normal form to a drawing that is in single-cross normal form. However, this method does not guarantee that edges which did not cross before the operation, will not cross afterwards (although, of course, edges that had no crossings to start with cannot have any crossings after the operation). This implies that the number of crossings counted after the operation might be more than when the number of pairwise edge crossings are counted without multiplicity before the operation. Of course, this should only be a problem in cases where some researchers have actually counted in this way. Székely [Szé04] demonstrated how such different interpretations may have been made.

In [PT98], Pach and Tóth define two new crossing numbers<sup>2</sup>. The *pairwise crossing number*<sup>3</sup>,  $\nu^{(p)}(\mathcal{G})$ , counts the total number of pairs of edges which cross each other in a graph drawing, without considering the multiplicity of the crossings between a pair of edges. The *odd crossing number*<sup>4</sup>,  $\nu^{(o)}(\mathcal{G})$ , is the total number of pairs of edges which cross each other an odd number of times (again without consideration for the multiplicity of crossings between any two edges).

**Definition 3.2.2** For a graph  $\mathcal{G}$ ,

1. define an indicator function for a pair of edges  $e, f \in E(\mathcal{G})$  and a drawing  $\phi(\mathcal{G})$  of  $\mathcal{G}$  in normal form as

$$\chi^{(p)}(e, f, \phi) = \begin{cases} 1 & \text{if } |\phi(\tilde{e}) \cap \phi(\tilde{f})| > 0 \text{ and } e \neq f, \\ 0 & \text{otherwise.} \end{cases}$$

Then the pairwise crossing number of  $\mathcal{G}$ , realized by  $\phi$ , is

$$\nu_{\phi}^{(p)}(\mathcal{G}) = \frac{1}{2} \sum_{e, f \in E(\mathcal{G})} \chi^{(p)}(e, f, \phi),$$

whilst the pairwise crossing number of  $\mathcal{G}$  is

$$\nu^{(p)}(\mathcal{G}) = \min_{\phi(\mathcal{G})} \{ \nu_{\phi}^{(p)}(\mathcal{G}) \}.$$

2. define an indicator function for a pair of edges  $e, f \in E(\mathcal{G})$  and a drawing  $\phi$  of  $\mathcal{G}$  in normal form as

$$\chi^{(o)}(e, f, \phi) = \begin{cases} 1 & \text{if } |\phi(\tilde{e}) \cap \phi(\tilde{f})| \text{ is odd and } e \neq f, \\ 0 & \text{otherwise.} \end{cases}$$

<sup>2</sup>Recently, Pach and Tóth [PT00] defined three rules which may be applied to any crossing number definition so as to obtain a variant crossing number definition. The first rule, denoted by a subscript plus sign requires that drawings be in single-cross normal form. The second rule, denoted by a subscript zero, stipulates that the crossings of adjacent edges in normal drawings must be counted, and the third rule, denoted by a subscript minus sign, stipulates that crossings between adjacent edges are not counted. It is a straightforward task to adapt the crossing number definitions in this chapter for these rules.

<sup>3</sup>Pach & Tóth [PT98] use the notation  $CR - PAIR(\mathcal{G})$  instead of  $\nu^{(p)}(\mathcal{G})$ .

<sup>4</sup>Pach & Tóth [PT98] use the notation  $CR - ODD(\mathcal{G})$  instead of  $\nu^{(o)}(\mathcal{G})$ .

Then the odd crossing number of  $\mathcal{G}$ , realized by  $\phi$ , is

$$\nu_{\phi}^{(o)}(\mathcal{G}) = \frac{1}{2} \sum_{e, f \in E(\mathcal{G})} \chi^{(o)}(e, f, \phi),$$

whilst the odd crossing number of  $\mathcal{G}$  is

$$\nu^{(o)}(\mathcal{G}) = \min_{\phi(\mathcal{G})} \{ \nu_{\phi}^{(o)}(\mathcal{G}) \}. \quad \blacksquare$$

Székely notes further in [Szé04] that another definition of the crossing number of a graph is implicitly present in Tutte’s algebraic theory of graph crossing configurations [Tut70]. This crossing number is called the Tutte crossing number, and is defined in the next section. Székely defined a restricted version of the Tutte crossing number, called the *independent-odd crossing number*<sup>5</sup>,  $\nu^{(i)}(\mathcal{G})$ . This parameter is the same as the odd crossing number, except that adjacent edges are assumed not to cross, and so only crossings between non-adjacent edges are counted.

The motivation for this definition arises from the fact that Tutte quite mysteriously claims that “[he is] taking the view that crossings of adjacent edges are trivial, and easily got rid of” (p. 47, [Tut70]). Székely’s view that “[he interprets the] sentence as a philosophical view and not a mathematical claim” (p. 341, [Szé04]) is also the preferred view in this thesis, since the problem of removing crossings between adjacent edges, is just a special case of the removal of crossings between non-adjacent edges.

**Definition 3.2.3** For a graph  $\mathcal{G}$ , define an indicator function for a pair of edges  $e, f \in E(\mathcal{G})$  and a drawing  $\phi$  of  $\mathcal{G}$  in normal form as

$$\chi^{(i)}(e, f, \phi) = \begin{cases} 1 & \text{if } |\phi(\tilde{e}) \cap \phi(\tilde{f})| \text{ is odd and if } e \text{ is not adjacent to } f \text{ and } e \neq f, \\ 0 & \text{otherwise.} \end{cases}$$

Then the independent-odd crossing number of  $\mathcal{G}$  realized by  $\phi$  is

$$\nu_{\phi}^{(i)}(\mathcal{G}) = \frac{1}{2} \sum_{e, f \in E(\mathcal{G})} \chi^{(i)}(e, f, \phi),$$

whilst the independent-odd crossing number of  $\mathcal{G}$  is

$$\nu^{(i)}(\mathcal{G}) = \min_{\phi(\mathcal{G})} \{ \nu_{\phi}^{(i)}(\mathcal{G}) \}. \quad \blacksquare$$

These definitions imply the inequality chain

$$\nu^{(i)}(\mathcal{G}) \leq \nu^{(o)}(\mathcal{G}) \leq \nu^{(p)}(\mathcal{G}) \leq \nu(\mathcal{G}). \quad (3.2)$$

It is very interesting, yet at the same time somewhat disquieting to note that a fair number of lower bounds to  $\nu(\mathcal{G})$  are also lower bounds to  $\nu^{(i)}(\mathcal{G})$ . Is it possible that some researchers might have had these alternative crossing number definitions in mind when carrying out their work? Pach and Tóth [PT97] seem to think so, and Székely (p. 339, [Szé04]) gave a plausible explanation: “the conjectured optimal drawings are usually [in normal form] and [in single-cross normal form], and the lower bounds . . . usually also apply for all kinds of crossing numbers.”

<sup>5</sup>Székely [Szé04] introduces  $\nu^{(i)}(\mathcal{G})$  as  $CR - IODD(\mathcal{G})$ .

It is, of course, quite possible that

$$\nu^{(i)}(\mathcal{G}) = \nu^{(o)}(\mathcal{G}) = \nu^{(p)}(\mathcal{G}) = \nu(\mathcal{G}). \quad (3.3)$$

Pach considers this one of the most exciting open problems in the area (p. 271 [Pac00]). This claim is certainly strengthened by the fact that Pach and Tóth [PT98] give a clever combinatorial rephrasing of  $\nu^{(o)}$  in the form of three linear programs<sup>6</sup>. The algorithm is discussed later in this thesis. Székely [Szé04] also applies a combinatorial approach to the determination of  $\nu^{(i)}$ , which turns out, quite interestingly, to operate very similarly to the two–page layout algorithms presented in this thesis — so much so, that an hybrid algorithm based on Székely’s ideas and two–page layouts was developed for this thesis, and will be demonstrated later.

The following theorems present strides in the direction of proving equality for (3.2). The first remarkable theorem, was proved independently by Tutte [Tut70] and Hanani (alias Chojnacki) [CAH34].

**Theorem 3.2.1** *If a graph  $\mathcal{G}$  may be drawn in  $\mathbb{R}^2$ , so that any two non-adjacent edges cross an even number of times, then  $\mathcal{G}$  is planar.* ■

Pach and Tóth proved the following result in their article in which they introduced the concepts of the pairwise crossing number and the odd crossing number [PT98].

**Theorem 3.2.2** *For any graph  $\mathcal{G}$ ,  $\nu(\mathcal{G}) \leq 2[\nu^{(o)}(\mathcal{G})]^2$ .* ■

Recently, Kolman and Matoušek [KM04] established a fairly tight relationship between the pairwise crossing number and the crossing number of a graph.

**Theorem 3.2.3** *For any graph  $\mathcal{G}$ ,*

$$\nu(\mathcal{G}) = O\left([\log n]^2\left(\nu^{(p)}(\mathcal{G}) + \sum_{v \in V(\mathcal{G})} [\deg_{\mathcal{G}}(v)]^2\right)\right). \quad \blacksquare$$

### 3.2.2 The book crossing number

Using the results from § 3.1.3.1, the definition of the crossing number of a graph in  $\mathbb{R}^2$  can easily be amended for this special case, by replacing the indicator function  $\chi$ , with a more easily computed combinatorial variant  $\chi^{(B)}$ .

It is implicit that drawings are generated by the method described in § 3.1.3.1, but it is important to keep this in mind, as it makes the problem far more tractable: A book drawing  $\phi$  of a graph  $\mathcal{G}$  simply contains information on the order of the vertices of  $\mathcal{G}$  on the spine of a book, and on which edges are drawn on which pages.

**Definition 3.2.4** *For any combinatorial book drawing  $\phi$  of a graph  $\mathcal{G}$ , define an indicator function*

$$\chi^{(B)}(e, f, \phi) = \begin{cases} 1 & \text{if } e \text{ and } f \text{ are alternating} \\ & \text{and if } e \text{ is on the same page as } f \text{ in } \phi, \\ 0 & \text{otherwise.} \end{cases}$$

---

<sup>6</sup>This is contained within their proof of the NP-completeness of computing  $\nu^{(o)}$ .

Then the  $n$ -page book crossing number of  $\mathcal{G}$ , realized by the drawing  $\phi$ , is

$$\nu_{\phi,n}(\mathcal{G}) = \sum_{e,f \in E(\mathcal{G})} \chi^{(B)}(e, f, \phi), \quad (3.4)$$

whilst the  $n$ -page crossing number of  $\mathcal{G}$  is defined as

$$\nu_n(\mathcal{G}) = \frac{1}{2} \min_{\phi(\mathcal{G})} \{ \nu_{\phi,n}(\mathcal{G}) \}. \quad \blacksquare$$

It trivially follows that  $\nu(\mathcal{G}) \leq \nu_2(\mathcal{G})$ , and it is shown later in this thesis that there exists a subdivision  $\mathcal{H}$  of any graph  $\mathcal{G}$  so that  $\nu(\mathcal{G}) = \nu_2(\mathcal{H})$ .

### 3.2.3 Variants of crossing number parameters

There are a number of interesting variations on the plane crossing number of a graph. The first class of variations deals with crossing number results for restricted types of drawings. Of these types, book drawings have already been described in § 3.1.3.1, and the book crossing number in § 3.2.2. Crossing number parameters have also been defined for rectilinear and layered drawings (§ 3.1.3.4). The definitions of the crossing number parameters for these types of drawings are almost the same as that of the plane crossing number defined in § 3.2; the only difference being that the drawing types are restricted to the drawing type under consideration. Because of the fact that both of these types of drawings require edges to be drawn as straight lines, they are not subject to the possible confusion in interpreting the minimum number of crossings that was noted by Pach and Tóth [PT98] (§ 3.2.1).

The second class of variations generalizes the crossing number problem to surfaces other than the plane. The most studied surfaces have been the orientable and non-orientable compact 2-manifolds, which are natural generalizations of the plane. These surfaces have been studied in [Ru96, SSSV, SSSV96c, SSSV94]. Because these spaces are locally 2-dimensional, Definition 3.1.2 holds if  $\mathbb{R}^2$  is replaced by the space in question. Thus, the classical crossing number definition (§ 3.2) holds for locally 2-dimensional spaces. Due to the generality, the crossing number definitions for these spaces are subject to Pach and Tóth's alternative interpretations of the crossing number (§ 3.2.1).

The third and final class contains miscellaneous other variations. A graph  $\mathcal{G}$  is said to be *biplanar*, if it may be partitioned into two subgraphs  $\mathcal{G}_1$  and  $\mathcal{G}_2$ , such that both  $\mathcal{G}_1$  and  $\mathcal{G}_2$  are planar. This parameter is closely related to the *graph thickness* parameter, denoted  $\theta(\mathcal{H})$  for a graph  $\mathcal{H}$ . It follows that  $\theta(\mathcal{G}) \leq 2$  for a biplanar graph  $\mathcal{G}$ . The *biplanar crossing number* is to the concept of biplanarity as the crossing number is to planarity. The biplanar crossing number, denoted<sup>7</sup>  $\nu_2^{(B)}(\mathcal{G})$  for a graph  $\mathcal{G}$ , is defined as

$$\nu_2^{(B)}(\mathcal{G}) = \min_{\mathcal{G}=\mathcal{G}_1 \cup \mathcal{G}_2} \{ \nu(\mathcal{G}_1) + \nu(\mathcal{G}_2) \}.$$

It has already been noted in § 3.2.2 that  $\nu(\mathcal{G}) \leq \nu_2(\mathcal{G})$  for a graph  $\mathcal{G}$  — that is, the crossing number of a graph  $\mathcal{G}$  is at most the total number of crossings counted in a two-page book layout of  $\mathcal{G}$ . Now, a four-page book layout of  $\mathcal{G}$  is equivalent to two, separate two-page layouts of two

---

<sup>7</sup>Czabarka, Sýkora, Székely and Vrto [CSSV04] use the notation  $\text{cr}_2(\mathcal{G})$  to denote the biplanar crossing number of a graph  $\mathcal{G}$ , which when written in the notation of this thesis would be  $\nu_2(\mathcal{G})$ . Since this notation is already used to denote the book crossing number of  $\mathcal{G}$  in a two-page book, the superscript (B) was added in this thesis to the notation used for the biplanar crossing number.

subgraphs  $\mathcal{G}_1$  and  $\mathcal{G}_2$  where  $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2$ . These two facts together imply that  $\nu_2^{(B)}(\mathcal{G}) \leq \nu_4(\mathcal{G})$ . A survey of results for the biplanar crossing number has been done by Czaparka, Sýkora, Székely and Vrto [CSSV04].

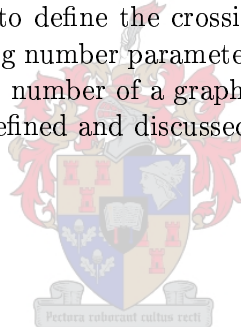
The concept of biplanarity, and the biplanar crossing number readily generalize to cases beyond the partitioning of a graph into two sets. A graph  $\mathcal{G}$  may be said to be  $n$ -planar if it can be partitioned into  $n$  planar subgraphs. The  $n$ -planar crossing number of a graph, denoted  $\nu_n^{(B)}(\mathcal{G})$ , is then defined as

$$\nu_n^{(B)}(\mathcal{G}) = \min_{\mathcal{G}=\mathcal{G}_1 \cup \mathcal{G}_2 \cup \dots \cup \mathcal{G}_n} \{\nu(\mathcal{G}_1) + \nu(\mathcal{G}_2) + \dots + \nu(\mathcal{G}_n)\}.$$

The  $n$ -planar crossing number of a graph  $\mathcal{G}$  is related to the book crossing number of  $\mathcal{G}$  on  $2n$  pages, so that  $\nu_n^{(B)}(\mathcal{G}) \leq \nu_{2n}(\mathcal{G})$ .

### 3.3 Chapter summary

The concept of a graph drawing in  $\mathbb{R}^2$  was formalized in this chapter, and it was shown that all graph drawings in  $\mathbb{R}^2$  may be transformed to drawings in so-called normal form. Important restricted variants of drawings in normal form were discussed, and finally the drawing definitions that have been developed were used to define the crossing number of a graph in the plane, as well as variants of this classical crossing number parameter. The definitions of drawings are dealt with in § 3.1.1 – § 3.1.3. The crossing number of a graph in the plane is defined in § 3.2, whilst variants of the crossing number are defined and discussed in § 3.2.1 – § 3.2.3.



# Chapter 4

## Literature review

In most sciences one generation tears down what another has built and what one has established another undoes. In mathematics alone each generation adds a new story to the old structure.

— *Hermann H. Hankel (1839–1873)*

This chapter provides an overview of parameters that are related to the crossing number, to analytical results and techniques from the literature, and to existing algorithms for finding an exact or approximate solution to the crossing number problem for a graph.

The crossing number problem itself has a number of variations (as mentioned in Chapter 3), all of which are interesting in their own right. However, this thesis concerns itself only with the crossing number in the plane, and with methods that aid in the determination of bounds on this crossing number. Restrictions of the crossing number problem (for example to layered drawings, or to rectilinear drawings) are ignored, since these are well studied topics that deserve to be studied further in their own right.

### 4.1 Parameters related to the crossing number

Since the crossing number of a graph is a measure of its non-planarity, one may ask whether other non-planarity measures might be of aid when studying the crossing number problem. A number of planarity related parameters have been defined, and the most well-known are dealt with here. Incidentally, the problems of determining any of these parameters for a general graph are in **NP**. The interested reader is referred to a complete survey by Liebers [Lie01].

#### 4.1.1 The maximum induced planar subgraph problem

The *max. induced planar subgraph problem* is considered first.

**Definition 4.1.1** *Given a graph  $\mathcal{G} = (V, E)$  and a positive integer  $k \leq |V|$ , the maximum induced planar subgraph problem may be formulated as: “Is there a subset  $V' \subseteq V$  so that  $|V'| \geq k$ , with the property that  $\langle V' \rangle$  is planar?” For a lack of agreed upon notation, the largest value of  $k$  for which the maximum induced planar subgraph problem may be answered in the affirmative for the graph  $\mathcal{G}$  is denoted  $m_v(\mathcal{G})$ . ■*

The maximum induced planar subgraph problem has not received much attention, and this is probably because for many graphs, induced planar subgraphs are very small. For example, the maximum planar subgraph for  $\mathcal{K}_n$ ,  $n > 4$  is  $\mathcal{K}_4$ . and consequently  $m_v(\mathcal{K}_n) = 4$ ,  $n > 4$ . The best general lower bound (see [WB78]) for the crossing number  $\nu(\mathcal{K}_n)$  is

$$\nu(\mathcal{K}_n) \geq \frac{n(n-1)(n-2)(n-3)}{20(n-6)(n-7)} \left\lfloor \frac{(n-6)}{2} \right\rfloor \left\lfloor \frac{(n-7)}{2} \right\rfloor,$$

which clearly suggests that there is no general relationship between the maximum induced planar subgraph problem and the graph crossing number problem.

### 4.1.2 The skewness of a graph

The skewness of a graph  $\mathcal{G}$  gives a measure of the minimum number of edges that would have to be removed from  $\mathcal{G}$  in order to obtain a planar graph.

**Definition 4.1.2** *A maximum planar subgraph of a graph  $\mathcal{G} = (V, E)$ , is a planar graph  $\mathcal{G}_2 = (V, E_2)$  with the property that  $E_2 \subseteq E$ , and such there exists no planar subgraph graph  $\mathcal{G}_3 = (V, E_3)$  of  $\mathcal{G}$  such that  $|E_2| < |E_3|$ . The number  $|E| - |E_2|$  is known as the skewness of  $\mathcal{G}$ , and for a lack of standard notation, is denoted here as  $s(\mathcal{G})$ . ■*

The maximum planar subgraph problem is related to the crossing number problem in the sense that, by removing appropriate edges from a drawing realizing the crossing number of a graph, one obtains a *maximal* planar subgraph — that is, a graph that will become non-planar with the addition of any removed edges. Unfortunately, it is not known whether drawings realizing the crossing number for a graph necessarily contain a maximum planar subgraph at the removal of the appropriate edges. If this were true, then the maximum planar subgraph problem would make it possible to determine the crossing number for a graph  $\mathcal{G}$ , by first finding a maximum planar subgraph  $\mathcal{H}$  of  $\mathcal{G}$ , and then by attempting all possible insertions of remaining edges  $E(\mathcal{G}) \setminus E(\mathcal{H})$  into  $\mathcal{H}$ . It trivially follows that  $s(\mathcal{G}) \leq \nu(\mathcal{G})$ , and for dense graphs such as  $\mathcal{K}_n$ ,  $\mathcal{K}_{m,n}$  and  $\overline{\mathcal{C}}_n$ , that  $s(\mathcal{G}) \ll \nu(\mathcal{G})$ , since at most  $O(|E(\mathcal{G})|)$  edges can be removed from  $\mathcal{G}$ , whilst the crossing numbers for such graphs are typically  $O(|V(\mathcal{K}_n)|^4) = O(|E(\mathcal{K}_n)|^2)$ .

### 4.1.3 The vertex splitting number

The operation of vertex splitting is a dual (in a very loose sense) of the operations of edge contraction and edge deletion, which are performed when constructing a minor of a graph.

**Definition 4.1.3** *A vertex splitting is an operation on a graph  $\mathcal{G} = (V, E)$ , where a vertex  $v \in V(\mathcal{G})$  is replaced by two vertices  $v_1, v_2$ , such that, for all edges  $e = \{v, u\} \in E(\mathcal{G})$ , there must at least be one edge  $\{v_1, u\}$  or  $\{v_2, u\}$ , although both may exist. Further, the edge  $\{v_1, v_2\}$  may be present. Clearly the vertex splitting creates a new graph  $\mathcal{G}' = (V', E')$ , and the following holds:*

$$\begin{aligned} V &= (V' \setminus \{v_1, v_2\}) \cup \{v\} \\ E &= (E' \setminus \{\{u, v_1\}, \{u, v_2\} : u \in V' \text{ and } \{u, v_1\} \in E' \text{ or } \{u, v_2\} \in E'\}) \\ &\quad \cup \{\{u, v\} : u \in V \setminus \{v\} \text{ and } \{u, v_1\} \in E' \text{ or } \{u, v_2\} \in E'\} \end{aligned} \quad \blacksquare$$

Application of the correct splitting operations to a non-planar graph may be used to obtain a new planar graph.



**Definition 4.1.4** *The splitting number  $\sigma(\mathcal{G})$  of a graph  $\mathcal{G}$  is the smallest number of vertex splittings, starting with  $\mathcal{G}$ , that will produce a planar graph  $\mathcal{G}'$ . ■*

Vertex splitting is interesting in its own right, but it is difficult to see whether it may be related to the crossing number in any practical way. It has generated a considerable amount of research, and some impressive analytical results have been obtained. For example, Jackson and Ringel [JR85, JR84] showed that

$$\sigma(\mathcal{K}_{m,n}) = \left\lceil \frac{(m-2)(n-2)}{2} \right\rceil,$$

and Hartsfield, Jackson and Ringel [HJR85] showed that

$$\sigma(\mathcal{K}_n) = \begin{cases} \left\lceil \frac{(n-3)(n-4)}{6} \right\rceil & \text{for } n \geq 3 \text{ and } n \notin \{6, 7, 9\} \\ \left\lceil \frac{(n-3)(n-4)}{6} \right\rceil + 1 & \text{for } n \in \{6, 7, 9\}. \end{cases}$$

#### 4.1.4 The genus of a graph

As mentioned in Chapter 3, the crossing number problem has been studied in the context of general compact 2-manifolds. Given a graph  $\mathcal{G}$ , another question that may be asked about compact 2-manifolds, is what the minimum genus for an orientable or non-orientable 2-manifold  $\mathcal{S}$  should be, in order that  $\mathcal{G}$  may be embedded (*i.e.*, drawn such that no edges are crossed) in  $\mathcal{S}$ .

**Definition 4.1.5** *The orientable genus<sup>1</sup> of a graph  $\mathcal{G}$  may be defined as*

$$\Upsilon(\mathcal{G}) = \min_h \{h : h \text{ is the genus of an orientable surface in which } \mathcal{G} \text{ has an embedding}\},$$

*and similarly, the non-orientable genus of  $\mathcal{G}$  may be defined as*

$$\hat{\Upsilon}(\mathcal{G}) = \min_h \{h : h \text{ is the genus of a non-orientable surface in which } \mathcal{G} \text{ has an embedding}\}. \blacksquare$$

All planar graphs have orientable and non-orientable genera zero, since they may be embedded on the sphere. Ringel and Youngs [RY68] proved that

$$\Upsilon(\mathcal{K}_n) = \frac{1}{12}(n-3)(n-4),$$

whilst Ringel [Rin65] showed that

$$\Upsilon(\mathcal{K}_{m,n}) = \frac{1}{4}(m-2)(n-2).$$

White and Beineke [WB78] give several results for the orientable and non-orientable genera of graphs.

There is a tenuous connection between the genus of a graph and the crossing number, since there exist classes of graphs with a bounded genus, but an unbounded crossing number. For example, the grids on the torus,  $\mathcal{C}_m \times \mathcal{C}_n$ , all have  $\Upsilon(\mathcal{C}_m \times \mathcal{C}_n) = 1$  but  $\nu(\mathcal{C}_m \times \mathcal{C}_n) \geq (1/2)(m-2)n$ , where  $m \leq n$  [JS01]. In general, for a graph  $\mathcal{G}$ , it is true that  $\Upsilon(\mathcal{G}) \leq \nu(\mathcal{G})$  and  $\hat{\Upsilon}(\mathcal{G}) \leq \nu(\mathcal{G})$ , since every crossing may be eliminated by the addition of a handle (in the orientable case) or a crosscap (in the non-orientable case) that acts as a bridge, allowing one edge to pass over another, thereby avoiding an intersection between the edges.

<sup>1</sup>The genus of a  $\mathcal{G}$  is often denoted  $\gamma(\mathcal{G})$ , but since this symbol is far more commonly used to denote the so-called (lower) domination number of a graph (see [Har69] for a definition),  $\Upsilon$  will be used instead.

### 4.1.5 The thickness of a graph

Like the crossing number problem of the graph, the graph thickness problem also has applications in the design of electronic circuits. In the graph thickness problem, a graph has to be subdivided into as few as possible partitions, each of which is planar. In this way, an electronic circuit may be designed to be laid out on multiple layers, which are stacked upon each other. Mutzel, Odenthal and Scharbrodt [MOS98] have compiled a thorough survey of results on the graph thickness problem.

**Definition 4.1.6** *The thickness  $\theta(\mathcal{G})$  (outerplanar thickness  $\theta^*(\mathcal{G})$ ) of a graph  $\mathcal{G}$  is the least number of sets into which  $E(\mathcal{G}) = E_1 \cup E_2 \cup \dots \cup E_n$  may be partitioned such that each  $\langle E_i \rangle$   $i \in 1, \dots, n$  is planar (outerplanar). ■*

It trivially follows that for any non-planar graph that  $\theta^*(\mathcal{G}) \geq \theta(\mathcal{G}) > 1$ . The thickness problem has been solved for some restricted classes of graphs. Beineke and Harary [BH65] found the thickness for  $\mathcal{K}_n$  in most cases. Alekseev and Gončakov [AG76] completed their results and showed that

$$\theta(\mathcal{K}_n) = \left\lceil \frac{n+7}{6} \right\rceil, \text{ for } n \neq 9, 10 \quad \text{and} \quad \theta(\mathcal{K}_9) = \theta(\mathcal{K}_{10}) = 3.$$

Beineke, Harary and Moon [BHM64] largely solved the problem for the bipartite graphs. They proved that

$$\theta(\mathcal{K}_{m,n}) = \left\lceil \frac{mn}{2(m+n-2)} \right\rceil,$$

except if  $m$  and  $n$  are both odd,  $m \leq n$ , and if there exists an integer  $k$  satisfying

$$n = \left\lfloor \frac{2k(m-2)}{m-2k} \right\rfloor.$$

Finally, Jünger, Mutzel, Odenthal and Scharbrodt [JMOS98] proved the very interesting result that if a graph  $\mathcal{G}$  is without  $\mathcal{K}_5$ -minors, then  $\theta(\mathcal{G}) \leq 2$ . This is vaguely reminiscent of Kuratowski's theorem (§ 2.1.5).

### 4.1.6 The page number of a graph

The page number problem (or book thickness problem), which is the problem of determining the minimum number of book pages required for a crossing-free book drawing of a graph, has applications in the field of electronic circuit design — see [CLR87]. This problem is analogous to the problem of determining the minimum genus required in a 2-manifold  $\mathcal{S}$ , so that a graph embedding may be realized in  $\mathcal{S}$ .

**Definition 4.1.7** *The page number, also called the pageness or book thickness,  $p(\mathcal{G})$  of a graph  $\mathcal{G}$  is the fewest number of pages that a book drawing of  $\mathcal{G}$  can have such that no edge crossings are present on any page. ■*

Bernhart and Kainen [BK79] proved that

$$p(\mathcal{K}_n) = \left\lceil \frac{m}{2} \right\rceil \quad \text{and} \quad p(\mathcal{K}_{m,n}) = m, \quad m \leq n, \quad n \geq m^2 - m + 1.$$

A book embedding of a graph  $\mathcal{G}$  realizing  $p(\mathcal{G})$  induces embeddings of disjoint subgraphs of  $\mathcal{G}$  on each of the  $p(\mathcal{G})$  pages. This is equivalent to outerplanar drawings of these subgraphs on  $p(\mathcal{G})$  separate planes such that the order of the vertices in each plane drawing corresponds to the order of the vertices on the spine of the book. Therefore, this parameter is more restrictive with regards to allowable embeddings than the outerplanar thickness  $\theta^*$ . Using this observation, it trivially follows that  $\theta(\mathcal{G}) \leq \theta^*(\mathcal{G}) \leq p(\mathcal{G})$  for any graph  $\mathcal{G}$ .

### 4.1.7 The coarseness of a graph

Liebers [Lie01] provides only a passing mention of the coarseness of a graph in her survey, and notes that according to Harary’s book [Har69] (p. 121), Paul Erdős introduced the notion of graph coarseness by accident.

**Definition 4.1.8** *The coarseness  $\xi(\mathcal{G})$  of a graph  $\mathcal{G}$ , is the largest number of pairwise edge disjoint non-planar subgraphs contained within  $\mathcal{G}$ . ■*

It is interesting to note that where, for the thickness of a graph, one seeks to minimize the number of edge disjoint planar subgraphs; one does almost the opposite for the coarseness of a graph.

## 4.2 Analytical results for the crossing number problem

Some ingenious analytical techniques for studying the crossing number of a graph have been developed. In spite of this, few exact results for the crossing number have been found, and where for some of the other parameters mentioned in the previous section, the cases relating to the complete graph, and sometimes the bipartite graph have been resolved, no such success has been achieved with the crossing number problem.

### 4.2.1 Tutte’s algebraic formulation of crossings in graph drawings

A rather thorough overview of Tutte’s algebraic theory of crossings in graph drawings is given in this section. This is necessary, as the algorithms of Pach and Tóth [PT98] and of Székely [Szé04], presented later in this chapter, both use ideas from the theory.

In [Tut70], Tutte developed an elegant algebraic structure which may be used to represent graph drawings. He defined a general transformation on graph drawings, expressible as the sum of elements in the algebraic structure, which may best be described (informally) as the process whereby edges are “pulled” over vertices — in Figure 4.1(a) the edge  $\{v_i, v_j\}$  is “pulled” over the vertex  $v_k$  in this fashion.

For the purposes of the theory, edges are considered as arcs (*i.e.*, edges are assigned orientations). Thus, arbitrary orientations for the edges of a given graph are chosen, and fixed. Any point on an arc  $e_i$  has a “left” and a “right” side. Another arc  $e_j$ , which passes through a point in  $e_i$ , does so either from left to right, or from right to left. If  $e_j$  crosses  $e_i$  from left to right, then  $e_i$  crosses  $e_j$  from right to left. An illustration of arc crossings is given in Figure 4.1(b).

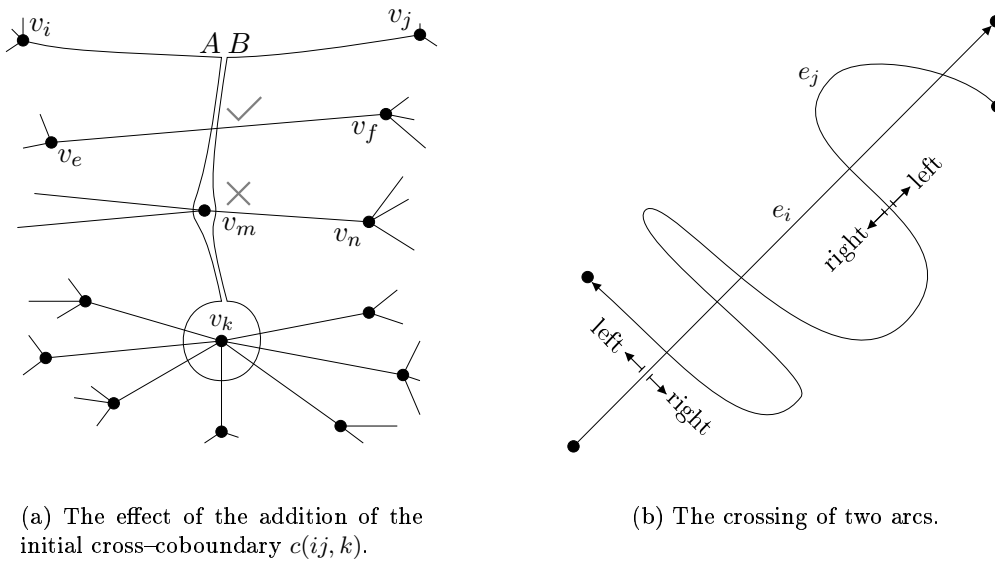


Figure 4.1: Illustrations of ideas in Tutte's theory.

The following important parameter was defined by Tutte (it should also be clear from this definition that his focus was on drawings of graphs in normal form):

$$\lambda(ij, k\ell) = \begin{cases} m - m' & \text{if } (v_i, v_j) \text{ crosses } (v_k, v_\ell) \text{ } m \text{ times from the left} \\ & \text{and } m' \text{ times from the right,} \\ 0 & \text{if } (v_i, v_j) \text{ or } (v_k, v_\ell) \text{ does not exist.} \end{cases}$$

The value of  $\lambda(ij, k\ell)$  is clearly not necessarily the same as the number of crossings between  $(v_i, v_j)$  and  $(v_k, v_\ell)$ , but it is congruent to the number of crossings between these edges modulo 2. In other words,  $\lambda(ij, k\ell)$  gives an indication of whether  $(v_i, v_j)$  and  $(v_k, v_\ell)$  cross an odd or even number of times. It follows from the definition of  $\lambda(ij, k\ell)$ , that its value may also be expressed as

$$\lambda(k\ell, ji), \lambda(\ell k, ij), \lambda(ji, \ell k), -\lambda(\ell k, ji), -\lambda(ij, \ell k), -\lambda(k\ell, ij) \text{ or } -\lambda(ji, k\ell). \quad (4.1)$$

#### 4.2.1.1 The algebraic structure — crossing chains

For a graph  $\mathcal{G}$ , define a set  $Q(\mathcal{G})$  of symbols  $[ij, k\ell]$ :

$$Q(\mathcal{G}) = \{[ij, k\ell] : 1 \leq i < j \leq |V(\mathcal{G})|, 1 \leq i < k < \ell \leq |V(\mathcal{G})|\}. \quad (4.2)$$

A new structure, called a *chain* is defined. It contains symbols from  $Q(\mathcal{G})$ , and a chain  $C$  has the form

$$C = \sum_{Q(\mathcal{G})} N_{ijkl} [ij, k\ell], \quad (4.3)$$

where  $N_{ijkl} \in \mathbb{N}$ . The symbols  $[ij, k\ell]$  are treated as indeterminates. Chains may be added, subtracted and multiplied by integers in exactly the same way as polynomials, and they therefore constitute the group  $R(\mathcal{G})$  of all chains. As with polynomials, when  $N_{ijkl} = 1$ , the coefficient in (4.3) is omitted altogether, and one only writes  $[ij, k\ell]$ .

A drawing  $\phi$  of a graph  $\mathcal{G}$  in the plane may be used to construct a corresponding chain. The integers  $\lambda(ij, k\ell)$  satisfy the requirements for the symbols from  $Q(\mathcal{G})$  (as long as the indices  $i, j, k, \ell$

are in the correct order, but application of the identities (4.1) always renders this possible). This prompts the following definition.

**Definition 4.2.1** *The crossing chain<sup>2</sup>  $x(\phi)$  corresponding to a drawing  $\phi$  of a graph  $\mathcal{G}$  is*

$$x(\phi) = \sum_{Q(\mathcal{G})} \lambda(ij, k\ell)[ij, k\ell]. \quad \blacksquare$$

In the same vein as the identities for  $\lambda(ij, k\ell)$ , it is useful to write  $[ij, k\ell]$  also as

$$[k\ell, ji], [\ell k, ij], [ji, \ell k], -[\ell k, ji], -[ij, \ell k], -[k\ell, ij], -[ji, k\ell]$$

so that the requirements in (4.2) for the indices  $i, j, k, \ell$  in the symbol  $[ij, k\ell]$  are met. For example  $\lambda(ij, k\ell)[ij, k\ell]$  may be rewritten as  $\lambda(k\ell, ij)[k\ell, ij]$  if  $k < \ell$  and  $k < i < j$ . When  $(v_i, v_j) \notin E(\mathcal{G})$  or  $(v_k, v_\ell) \notin E(\mathcal{G})$  or when  $i, j, k, \ell$  are not distinct, it is convenient to write  $[ij, k\ell] = 0$ .

#### 4.2.1.2 Transformations on drawings by cross-coboundaries

Referring to Figure 4.1(a), if the edge  $e = (v_i, v_j)$  is to be “pulled” across the vertex  $v_k$ , then an arbitrarily small gap is made at some point along  $(v_i, v_j)$ , which in the figure is the gap between  $A$  and  $B$ ; a line is drawn from  $A$  to the vicinity of  $v_k$  so that it does not intersect the drawings of any vertices; the line is circled around  $v_k$ , and returns along the same path to  $B$ , crossing exactly the same edges it crossed on its way from  $A$  to  $v_k$ , again not intersecting any vertices.

It is possible that an edge, like  $(v_e, v_f)$  may be crossed twice, but since it is crossed once from the left, and once from the right, the crossings cancel in the crossing chain. The crossing situation around the vertex  $v_m$  is disallowed, according to the transformation. Thus, only edges incident to  $v_k$  will contribute to changes in the crossing chain.

Formally, the crossings introduced by the transformation are accounted for in the chain  $c(ij, k)$ , defined as

$$c(ij, k) = \sum_{\ell=1}^{|V(\mathcal{G})|} [ij, k\ell].$$

According to the definitions in the previous subsection, it follows that

$$c(ij, k) = -c(ji, k).$$

Tutte called the chains  $c(ij, k)$  *initial cross-coboundaries*<sup>3</sup>, and chains expressible as a finite sum of initial cross-coboundaries, *general cross-coboundaries*. Clearly cross-coboundaries form a group, which is denoted  $R_c(\mathcal{G})$ , and it is also a subgroup of  $R(\mathcal{G})$ . Note that  $R_c(\mathcal{G})$  obviously contains the zero chain, which just leaves the graph as it is.

If  $x(\mathcal{G})$  is the crossing chain of a drawing before the transformation, then the crossing chain of the drawing after the transformation is either  $x(\mathcal{G}) + c(ij, k)$  or  $x(\mathcal{G}) - c(ij, j)$ , depending on the orientation of  $(v_i, v_j)$ .

<sup>2</sup>Tutte chose to denote a crossing chain by  $\chi$  in [Tut70], but this symbol already denotes the indicator function used to calculate the crossing number for a graph (§ 3.2) in this thesis.

<sup>3</sup>Tutte denoted an initial cross-coboundary  $c(ij, k)$  as  $K(ij, k)$ , but considering that  $K$ , or even  $k$  is too easily interpreted as the complete graph, the symbol  $c$  is used in this thesis.

### 4.2.1.3 A lower bound to the crossing number

Tutte noted that the sum of the absolute values of the parameters  $\lambda(ij, k\ell)$  is less than or equal to the total number of crossings in the corresponding drawing of a graph. It is not necessarily less than or equal to the crossing number of a graph (as defined in § 3.2), since the drawing under consideration need not be in single-cross normal form. But if one can show that  $|\lambda(ij, k\ell)| \leq 1$ , one arrives at Székely’s definition of the independent-odd crossing number (§ 3.2.1), which is certainly at most equal to the crossing number of a given graph.

**Definition 4.2.2** For a graph  $\mathcal{G}$ , the Tutte crossing number is defined as

$$\nu^{(t)}(\mathcal{G}) = \min_{\substack{\text{drawings in} \\ \text{normal form}}} \sum_{\substack{i < j, k < \ell \\ i < k}} |\lambda(ij, k\ell)|. \quad (4.4)$$

In the case where  $|\lambda(ij, k\ell)| \leq 1$ , the right-hand side of (4.4) is simply the independent-odd crossing number, and so the chain of inequalities  $\nu^{(i)}(\mathcal{G}) \leq \nu^{(t)}(\mathcal{G}) \leq \nu(\mathcal{G})$  is evident. Székely asserts that “in Tutte’s work, another kind of crossing number is implicit: the independent-odd crossing number...” (p. 333, [Szé04]). This is true under the particular interpretation that Székely makes (*i.e.*, the independent-odd crossing number, § 3.2.1), although he is aware of this, as he states in § 7.1 of [Szé04], when he concludes that his own algorithm for the computation of  $\nu^{(i)}$  uses a “mod 2 version of Tutte’s theory” (*ibid.*). That is, the  $\lambda$  coefficients are assumed to be either 0 or 1 for the independent-odd crossing number.

Székely’s interpretation is quite valid: for any sequence of cross-coboundaries  $c_1, c_2, \dots, c_t$ , that transform a crossing chain  $x(\phi)$  it follows, for any term  $\lambda(ij, k\ell)$  in  $c_1 + c_2 + \dots + c_t$ , that  $|\lambda(ij, k\ell)| \leq 1$ . This is true, since if an edge is “pulled” across a vertex  $v_m$ , it crosses all edges incident to  $v_m$  in directions that are opposite to the directions in which these edges were crossed before the “pull.” Moreover, since the original chain  $x(\phi)$  may be seen as a drawing where certain edges are initially “pulled” over particular vertices, by the same reasoning it follows that, if for every term  $\lambda(ij, k\ell)$  in  $x(\phi)$ , it holds that  $|\lambda(ij, k\ell)| \leq 1$ , then  $|\lambda(ij, k\ell)| \leq 1$  in  $x(\phi) + c_1 + c_2 + \dots + c_t$ .

It is always possible to construct a drawing  $\phi$  of a graph  $\mathcal{G}$  so that  $|\lambda(ij, k\ell)| \leq 1$  for all unordered pairs of edges  $(v_i, v_j), (v_k, v_\ell) \in E(\mathcal{G})$  where  $(v_i, v_j) \neq (v_k, v_\ell)$  — simply place the vertices of  $\mathcal{G}$  equi-spaced on an imaginary circle, and connect the vertices by straight lines (if three or more lines meet at a common point, this may be corrected by the method described in § 3.1.2). Clearly the total number of crossings between any pair of edges is at most one in such a drawing.

### 4.2.1.4 Tutte’s main result

Clearly, for a graph  $\mathcal{G}$  and a drawing  $\phi$ ,  $R_c(\mathcal{G}) + x(\phi)$  is a coset of  $R_c(\mathcal{G})$ . Tutte showed, if every  $\lambda$  term in the crossing chain  $x(\phi)$  of a drawing is even, that  $x(\phi) \in R_c(\mathcal{G})$ . Thus, since the zero chain is in  $R_c(\mathcal{G})$ , it is possible to find elements in  $R_c(\mathcal{G})$  so as to remove all crossings in  $\phi$ , meaning that  $\mathcal{G}$  must be planar.

## 4.2.2 Bounding techniques

The most obvious approach towards finding bounds on the crossing number of a particular graph, is to use an analytical lower bound (which is typically quite weak, since it is likely to be a general

bound), and to find a drawing of the graph with as few crossings as possible to bound the crossing number from above. This strategy is quite limited, and consequently some other methods have been used with some success — for example, the graph-to-graph embedding method described in this section has been used to determine bounds for the crossing numbers of hypercubes (defined later in the chapter) so that the upper and lower bounds are asymptotically equal.

#### 4.2.2.1 Standard counting method

Given a graph  $\mathcal{G}$ , the standard counting method for bounding  $\nu(\mathcal{G})$  may be described as finding a number of distinct subgraphs  $S = \{\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_t\}$ , so that  $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2 \cup \dots \cup \mathcal{G}_t$ , and such that for any  $\mathcal{G}_i, \mathcal{G}_j \in S, \mathcal{G}_i \cap \mathcal{G}_j \notin S$ ; and adding up the best available lower bounds on the crossing numbers of each graph in  $S$ . The subgraphs need not be pairwise disjoint with respect to their vertex sets nor with respect to their edge sets — in fact, better results are obtained when the subgraphs are chosen as large as possible. If the subgraphs are not pairwise disjoint in their edge sets, then some crossings will be counted more than once wherever two subgraphs intersect. This must be accounted for and subtracted from the final result.

In the simplest case, which is typically how this method is applied in the literature, the graph  $\mathcal{G}$  is highly symmetric, and the subgraphs are all of the same size. For example, for  $\mathcal{G} \cong \mathcal{K}_n$ , the subgraphs may be chosen to be isomorphic to  $\mathcal{K}_{n-1}$ . The procedure described here is greatly simplified: There are  $n$  copies of  $\mathcal{K}_{n-1}$  in  $\mathcal{K}_n$ . Every crossing is between two edges, and therefore four distinct vertices (the standard definition of the crossing number requires single crossing drawings in single-cross normal form, as defined in § 3.1.2) are involved in every crossing. For a given set of four vertices, it is found that  $\binom{n-4}{n-3} = n - 4$  subgraphs contain the four vertices — therefore, every crossing is counted  $n - 4$  times, and the lower bound

$$\nu(\mathcal{K}_n) \geq \frac{n}{n-4} \nu(\mathcal{K}_{n-1}) \quad (4.5)$$

is obtained. Application of this bound to  $\mathcal{K}_6$  yields  $\nu(\mathcal{K}_6) \geq (6/2)\nu(\mathcal{K}_5) = 3$ . A drawing of  $\mathcal{K}_6$  is shown in Figure 4.2(a) (this also shows that  $\nu(\mathcal{K}_6) \leq 3$ , and consequently, the two bounds imply that  $\nu(\mathcal{K}_6) = 3$ ), and the six different subgraphs which are isomorphic to  $\mathcal{K}_5$  are shown in Figures 4.2(b)–(g). It may be seen that the same crossing will be counted in the constructions in Figures 4.2(b) and (c). This is also true for the two constructions in Figures 4.2(d) and (e), and finally also for the pair of constructions in Figures 4.2(f) and (g).

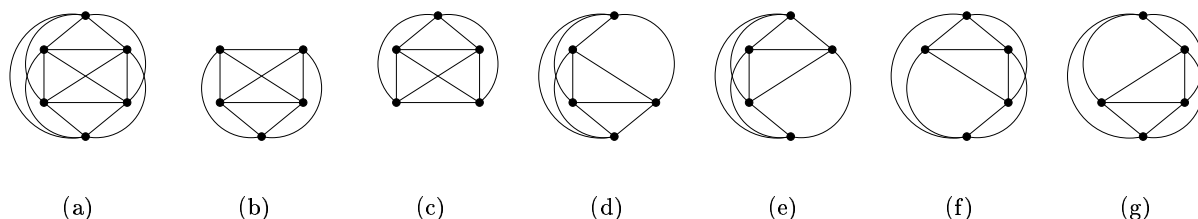


Figure 4.2:  $\mathcal{K}_6$  has 6 distinct subgraphs isomorphic to  $\mathcal{K}_5$ , where each crossing in  $\mathcal{K}_6$  is shared by two subgraphs.

### 4.2.2.2 Graph-to-graph embedding

Leighton [Lei83] introduced the ingenious method of *graph-to-graph embedding*<sup>4</sup> for deriving both upper and lower bounds on the crossing number of a graph. Shahrokhi, Sýkora, Székely and Vrto include graph-to-graph embedding in their survey [SSSV97a], which includes much of their own work on the subject. Many ideas presented in this section also derive from their work.

**Definition 4.2.3** For two graphs  $\mathcal{G}$  and  $\mathcal{H}$  where  $|V(\mathcal{G})| \leq |V(\mathcal{H})|$ , a graph-to-graph embedding  $\psi(\mathcal{G}) = (\psi^{(v)}, \psi^{(e)})$  of  $\mathcal{G}$  into  $\mathcal{H}$  is a pair of injections

$$\psi^{(v)} : V(\mathcal{G}) \rightarrow V(\mathcal{H}) \text{ and} \tag{4.6}$$

$$\psi^{(e)} : E(\mathcal{G}) \rightarrow \{p : p \text{ is a path in } \mathcal{H}\}, \tag{4.7}$$

where edge mappings  $\psi^{(e)}(e)$ ,  $e \in E(\mathcal{G})$ , are subject to the constraint that

$$\psi^{(e)}(e) = \psi^{(e)}(\{v_i, v_j\}) = p \in \{\text{all } \psi^{(v)}(v_i) - \psi^{(v)}(v_j) \text{ paths in } \mathcal{H}\}. \quad \blacksquare$$

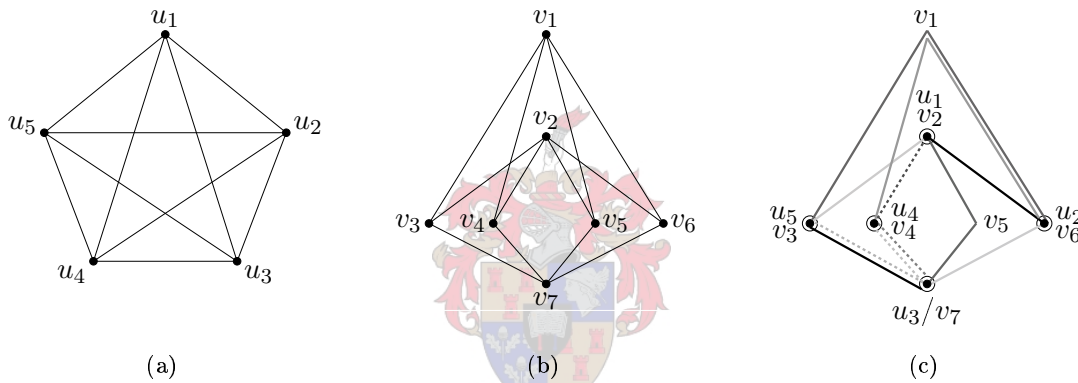


Figure 4.3: A graph-to-graph embedding of  $\mathcal{K}_5$  into  $\mathcal{K}_{4,3}$ .

As an example, consider a graph-to-graph embedding of  $\mathcal{K}_5$  into  $\mathcal{K}_{3,4}$ . The vertices of  $\mathcal{K}_5$  are labelled as  $V(\mathcal{K}_5) = \{u_1, \dots, u_5\}$  and the two partite sets  $V_1 \cup V_2 = V(\mathcal{K}_{3,4})$  of  $\mathcal{K}_{3,4}$  are labelled as  $V_1 = \{v_1, v_2, v_7\}$  and  $V_2 = \{v_3, v_4, v_5, v_6\}$ . Define the graph-to-graph embedding  $\psi'(\mathcal{K}_5)$  by mapping the vertices as

$$\psi^{(v)}(u_1) = v_2, \quad \psi^{(v)}(u_2) = v_6, \quad \psi^{(v)}(u_3) = v_7, \quad \psi^{(v)}(u_4) = v_4, \quad \psi^{(v)}(u_5) = v_3,$$

and the edges as

$$\begin{aligned} \psi^{(e)}(\{u_1, u_2\}) &= [v_2, v_6], & \psi^{(e)}(\{u_1, u_3\}) &= [v_2, v_5, v_7] \\ \psi^{(e)}(\{u_1, u_4\}) &= [v_2, v_4], & \psi^{(e)}(\{u_1, u_5\}) &= [v_2, v_3] \\ \psi^{(e)}(\{u_2, u_3\}) &= [v_6, v_7], & \psi^{(e)}(\{u_2, u_4\}) &= [v_6, v_1, v_4] \\ \psi^{(e)}(\{u_2, u_5\}) &= [v_6, v_1, v_3] \\ \psi^{(e)}(\{u_3, u_4\}) &= [v_4, v_7], & \psi^{(e)}(\{u_3, u_5\}) &= [v_3, v_7] \\ \psi^{(e)}(\{u_4, u_5\}) &= [v_3, v_7, v_4]. \end{aligned}$$

From the example it may be seen that paths in  $\mathcal{K}_{3,4}$  corresponding to mappings of edges from  $\mathcal{K}_5$  may intersect, so that a single vertex or edge in  $\mathcal{K}_{3,4}$  may “route” a number of edges from  $\mathcal{K}_5$ , as is the case for  $v_1 \in V(\mathcal{K}_{3,4})$ .

<sup>4</sup>In the crossing number literature, this is typically only called *graph embedding*, but in order to avoid confusion of this method with graph drawings which are themselves embeddings, the given terminology will be used.



### Obtaining upper bounds

In order to obtain an upper bound on the crossing number of a graph  $\mathcal{G}$ , using a graph-to-graph embedding  $\psi$  of  $\mathcal{G}$  into  $\mathcal{H}$ , a drawing  $\phi$  of  $\mathcal{H}$  is required. To construct a drawing  $\phi_2$  for  $\mathcal{G}$ , the following simple procedure is followed:

1. Draw the vertices of  $\mathcal{G}$  onto the positions of the vertices in  $\mathcal{H}$  corresponding to their mappings.
2. For every edge  $e \in E(\mathcal{G})$ , draw a curve along the path  $p = \psi^{(e)}(e)$ , so that it does not pass through any intermediate vertex on  $p$  and so that it does not intersect any other drawings of edges along  $p$ , unless the intersection is within a distance  $\varepsilon > 0$  from a vertex, where  $\varepsilon$  is an arbitrarily small positive real number. The drawings of edges are “looped” around vertices (this is similar to the transformation which ensures that every crossing point involves only two edges, as described in § 3.1.2), where the “loop” is within distance  $\varepsilon$  of the vertex, and where consecutive edges are “looped” outwards around the vertex.

This action may be seen in Figure 4.4(b), which is a section of a drawing for an embedding obtained from the drawing in Figure 4.4(a).

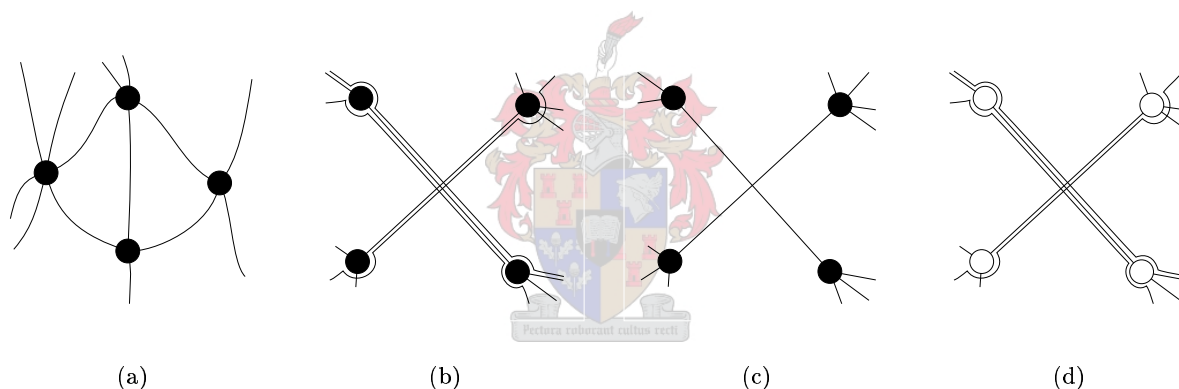


Figure 4.4: The first two figures show how graph-to-graph embeddings may cause crossings at vertices, whilst the last two figures show that multiple crossings may be caused at a single crossing in the graph into which the embedding has been done.

By constructing a drawing  $\phi_2$  of  $\mathcal{G}$  in this way, it is straightforward to count the number of crossings in  $\phi_2$ . Crossings can only occur either at

1. *crossings in  $\phi$*  — that is, if edges from  $\mathcal{G}$  map to paths in  $\mathcal{H}$ , which contain edges that cross in  $\phi$ , then of course the drawing  $\phi_2$  will contain crossings at those points (there may be multiple crossings in  $\phi_2$ , since multiple paths may be routed through an edge which contains a crossing, as is shown in Figure 4.4(d), where the original drawing is shown in Figure 4.4(c)),
2. *vertices*, or more correctly, the vicinities within a distance  $\varepsilon$  of vertices — if two paths  $p_1$  and  $p_2$  (corresponding to edges in  $\mathcal{G}$ ) both contain a vertex  $v \in V(\mathcal{H})$ , then it is possible that a crossing will be caused in  $\phi_2$ . Denote the two edges from  $p_1$  adjacent to  $v$  by  $e_a$  and  $e_b$ , and the two edges from  $p_2$  adjacent to  $v$  by  $e_c$  and  $e_d$ . If, in the drawing  $\phi$ , it is found that  $e_a$  is not followed or preceded by  $e_b$  when considering a clockwise ordering of edges around  $v$ , it means that they are separated by  $e_c$  and  $e_d$  ( $e_c$  and  $e_d$  are also separated by  $e_a$

and  $e_b$ ) — this forces a crossing, as shown in Figure 4.5(c). If this is not the case, then a crossing can be avoided, as shown in Figures 4.5(a) and (b). Of course, if  $p_1$  and  $p_2$  share a vertex  $v$  only by virtue of both ending at  $v$ , then no crossing needs to occur between them.

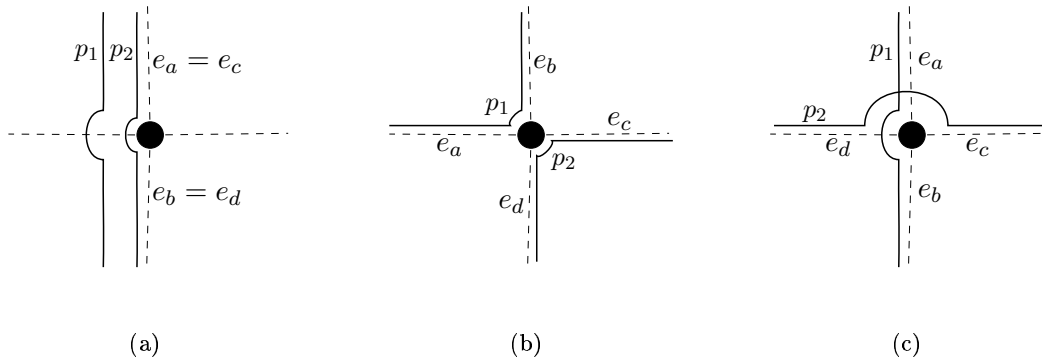


Figure 4.5: Graph-to-graph embeddings may cause crossings at vertices.

Since the total number of crossings in  $\phi_2$  depend on the number of edges that are mapped to particular edges or vertices in  $\mathcal{H}$ , two new quantities (both of which are defined in terms of the edge mapping  $\psi^{(e)}$ ), called respectively the *edge congestion* of an edge and the *vertex congestion* of a vertex in  $\mathcal{H}$  are defined.

**Definition 4.2.4** For two graphs  $\mathcal{G}$  and  $\mathcal{H}$  and a graph-to-graph embedding  $\psi$  of  $\mathcal{G}$  into  $\mathcal{H}$ , define

1. the edge congestion  $c^{(e)}(e', \psi)$  of an edge  $e' \in E(\mathcal{H})$  as

$$c^{(e)}(e', \psi) = |\{f \in E(\mathcal{G}) : e' \in \psi^{(e)}(f)\}|,$$

2. the vertex congestion  $c^{(v)}(v', \psi)$  of a vertex  $v' \in V(\mathcal{H})$  as

$$c^{(v)}(v', \psi) = |\{f \in E(\mathcal{G}) : v' \in \psi^{(e)}(f)\}|. \quad \blacksquare$$

At each point where there is a crossing between two edges  $e_i, e_j \in E(\mathcal{H})$ , it is found that there are  $c^{(e)}(e_i, \psi) \times c^{(e)}(e_j, \psi)$  crossings in the drawing  $\phi_2$ . This is represented graphically in Figure 4.4(d), where multiple edges are mapped through the two original edges in Figure 4.4(c).

Shahrokhi, Sýkora, Székely and Vrto [SSSV97a] made the conservative assumption that at every vertex  $v \in V(\mathcal{H})$ , every path  $p$  that is routed through  $v$  may cross every other path that is so routed. This renders the upper bound<sup>5</sup>  $\binom{c^{(v)}(v, \psi)}{2}$  for the number of crossings that occur at a vertex  $v$ .

Combining these two types of crossings, it follows, for a particular embedding  $\psi$  and a particular drawing  $\phi$  of  $\mathcal{H}$ , that

$$\nu(\mathcal{G}) \leq \sum_{\substack{e_i, e_j \in E(\mathcal{H}) \\ e_i \text{ crosses } e_j \text{ in } \phi}} c^{(e)}(e_i, \psi) \times c^{(e)}(e_j, \psi) + \sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}. \quad (4.8)$$

<sup>5</sup>Shahrokhi, Sýkora, Székely and Vrto used a factor of  $[c^{(v)}(v, \psi)]^2/2$  in [SSSV96a, SSSV94] which, although correct, could render a slightly larger upper bound.

### Obtaining lower bounds

Shahrokhi, Sýkora, Székely and Vrto's [SSSV96a, SSSV94] innovation was to turn the upper bound into a lower bound. They considered how the right-hand side of (4.8) could be made independent of a particular drawing of  $\mathcal{H}$  and, in particular, how it could be expressed in terms of  $\nu(\mathcal{H})$ . If the first term,

$$\sum_{\substack{e_i, e_j \in E(\mathcal{H}) \\ e_i \text{ crosses } e_j \text{ in } \phi}} c^{(e)}(e_i, \psi) \times c^{(e)}(e_j, \psi),$$

in (4.8) is to be attained for any drawing  $\phi$  of  $\mathcal{H}$ , then the possibility exists that the edge  $e_h$  with the highest edge congestion will be crossed by every other edge in  $\mathcal{H}$ . By letting

$$\begin{aligned} c^{(e)}(\psi) &= \max_{e \in E(\mathcal{H})} \{c^{(e)}(e, \psi)\}, \\ c^{(v)}(\psi) &= \max_{v \in V(\mathcal{H})} \{c^{(v)}(v, \psi)\}, \end{aligned}$$

the first term of (4.8) may be approximated as

$$c^{(e)}(\psi) \sum_{\substack{e_i \in E(\mathcal{H}) \\ e_i \text{ is crossed in } \phi}} c^{(e)}(e_i, \psi). \quad (4.9)$$

However, it is not known which edges in  $\mathcal{H}$  will cross  $e_h$  — the route that Shahrokhi, Sýkora, Székely and Vrto took was to assume that each crossing occurs between two edges with the highest edge congestion  $c^{(e)}(\psi)$  (this is a very strict assumption, and a more relaxed, albeit more complex assumption is given in later in this thesis), which simplifies the summation (4.9) to  $[c^{(e)}(\psi)]^2 \nu_\phi(\mathcal{H})$ , since there are  $\nu_\phi(\mathcal{H})$  terms within the summation (one for each crossing in  $\phi$ ). If this is to be realised for any drawing  $\phi$  of  $\mathcal{H}$ , it must also be realised for optimal drawings of  $\mathcal{H}$ , leading to the final form of  $[c^{(e)}(\psi)]^2 \nu(\mathcal{H})$ .

Due to Shahrokhi, Sýkora, Székely and Vrto's conservative assumption regarding crossings caused at vertices, the second factor is attained for any drawing of  $\mathcal{H}$ . For their lower bound method however, they replaced the term

$$\sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}, \text{ by } |V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2},$$

which results in a potentially somewhat weaker upperbound. From the foregoing discussion, the inequality

$$\nu(\mathcal{G}) \leq \nu(\mathcal{H}) [c^{(e)}(\psi)]^2 + |V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2},$$

is evident, which when rewritten to make  $\nu(\mathcal{H})$  the subject, yields

$$\nu(\mathcal{H}) \geq \frac{\nu(\mathcal{G}) - |V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2}}{[c^{(e)}(\psi)]^2}, \quad (4.10)$$

giving a method to compute a lower bound to  $\nu(\mathcal{H})$ . Obviously the quality of the bound depends on the maximum values of the edge and vertex congestion values.

### Graph-to-graph embeddings into $\mathcal{K}_n$

In a restricted version of the graph-to-graph embedding method, which has been used by Székely, Shahrokhi, Sýkora and Vrřo, the edge mappings  $\psi^{(e)}$  of a graph-to-graph embedding  $\psi$  from a graph  $\mathcal{G}$  to a graph  $\mathcal{H}$  maps edges from  $\mathcal{G}$  only to edges in  $\mathcal{H}$ , and not to paths in  $\mathcal{H}$  as with the general version. This requirement has the effect that edge mappings  $\psi^{(e)}$  are automatically determined by the vertex mappings  $\psi^{(v)}$ , since an edge  $e = \{u, v\} \in E(\mathcal{G})$  maps to the edge  $f = \{\psi^{(v)}(u), \psi^{(v)}(v)\} \in E(\mathcal{H})$ . In this restricted version, it is also required that  $|V(\mathcal{G})| = |V(\mathcal{H})|$ , which means that  $\psi^{(v)}$  is a bijection — this restriction is not strictly necessary, but this is how the graph-to-graph embedding method was studied by Shahrokhi, Székely, Sýkora and Vrřo. For the purposes of this thesis, this restricted graph-to-graph embedding is called a *single-edge graph-to-graph embedding*. Also, since a single-edge graph-to-graph embedding  $\psi = (\psi^{(v)}, \psi^{(e)})$  is entirely determined by the bijection  $\psi^{(v)}$  of the vertices from  $\mathcal{G}$  to the vertices of  $\mathcal{K}_n$ , a single-edge graph-to-graph embedding is simply denoted  $\bar{\psi}$ .

The only way to ensure that an edge  $f = \{\psi^{(v)}(u), \psi^{(v)}(v)\} \in E(\mathcal{H})$  exists, for an edge  $\{u, v\} \in E(\mathcal{G})$ , and for any mapping  $\psi$ , is to choose the graph  $\mathcal{H}$  to be isomorphic to the complete graph  $\mathcal{K}_n$ , where  $|V(\mathcal{H})| = n$ . This restriction may be relaxed, but this will place constraints on how  $\psi$  can be chosen. For the purposes of this thesis,  $\mathcal{H} \cong \mathcal{K}_n$  will be assumed.

Since edges may no longer be routed through vertices, crossings can only occur at crossings in  $\mathcal{K}_n$ . Also the edge congestion can only be equal to one in this scheme, and it follows thus that

$$\nu(\mathcal{G}) \leq \sum_{\substack{e_i, e_j \in E(\mathcal{H}) \\ e_i \text{ crosses } e_j \text{ in } \phi(\mathcal{K}_n)}} c^{(e)}(e_i, \psi) \times c^{(e)}(e_j, \psi).$$

The point of single-edge graph-to-graph embeddings, is that for a random mapping  $\psi$  of the vertices of a graph  $\mathcal{G}$  to the vertices of  $\mathcal{K}_n$ , the expected number of crossings that will be present in the drawing  $\phi_2(\mathcal{G})$  obtained from a drawing  $\phi$  of  $\mathcal{K}_n$  and the graph-to-graph embedding  $\psi$ , can be determined. This property is exploited by Shahrokhi, Székely, Sýkora and Vrřo’s probabilistic embedding algorithm, which is described later in this chapter.

#### 4.2.2.3 Using graph minors

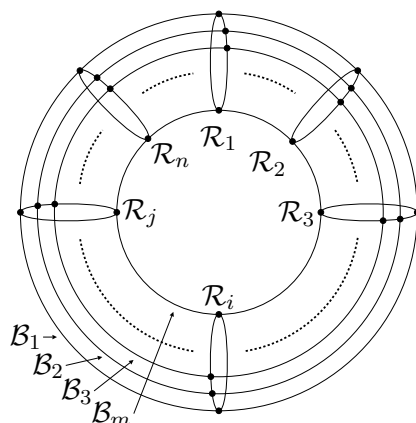
Garcia–Moreno and Salazar [GMS01] proved that the following result holds for the crossing number of a graph  $\mathcal{G}$  in a compact 2–manifold (§ 4.1.4). It is stated here only in the form of the plane crossing number.

**Theorem 4.2.1** *For a graph  $\mathcal{G}$ , and a minor  $\mathcal{M}$  of  $\mathcal{G}$  with  $\Delta(\mathcal{M}) \leq 4$ ,*

$$\nu(\mathcal{G}) \geq (1/4)\nu(\mathcal{M}).$$

■

No significant bounds for the crossing number of any graph in the plane have yet been obtained by using this method (although Garcia–Moreno and Salazar successfully obtained lower bounds for all graphs with respect to their so called “representativity” on the Klein bottle in [GMS01]), and in the cases where graphs are symmetrical, the standard counting method and the graph-to-graph embedding method seem to fare better.

Figure 4.6:  $\mathcal{C}_m \times \mathcal{C}_n$ .

#### 4.2.2.4 The method of graph bisection

The *bisection width*  $b(\mathcal{G})$  of a graph  $\mathcal{G}$  is the minimum number of edges whose removal partitions the vertex set  $V(\mathcal{G})$  into two sets  $V_1(\mathcal{G})$  and  $V_2(\mathcal{G})$  such that  $|V_1(\mathcal{G})|, |V_2(\mathcal{G})| \geq (1/3)|V(\mathcal{G})|$ . The problem of determining the bisection width of a graph is known to be **NP**-complete [GJS76]. Leighton [Lei83] showed that

$$\nu(\mathcal{G}) + n = \Omega(b(\mathcal{G})^2). \quad (4.11)$$

This result in itself is not immediately useful. However, it may be used recursively to establish a provably good upper bound to the crossing number of a graph, as will be discussed later in this chapter.

A more general version of this result, which is known by researchers in the field of VLSI design ([SSSV97a], p.10), and which is immediately applicable, is

$$\nu(\mathcal{G}) \geq \frac{1}{16} \left( \frac{b(\mathcal{G})}{1.58} \right)^2 - \frac{1}{16} \sum_{v \in V(\mathcal{G})} [d_{\mathcal{G}}(v)]^2, \quad (4.12)$$

for which proofs may be found in [PSS96] and in [SV94].

This method is only useful for classes of graphs where the bisection width is a function of the number of vertices. This is, for example, not true of the product of cycles  $\mathcal{C}_m \times \mathcal{C}_n$  and it is easy to show that  $b(\mathcal{C}_m \times \mathcal{C}_n) \leq 2m$ , when  $m \leq n$ . A drawing construction for the graph  $\mathcal{C}_m \times \mathcal{C}_n$ ,  $m \leq n$ , is shown in Figure 4.6. The  $m$   $n$ -cycles are the large circles, labelled  $\mathcal{B}_1$  through  $\mathcal{B}_m$ , and the  $n$   $m$ -cycles are the smaller ellipses, which are distributed around the circumference of the larger  $n$ -cycles. The  $m$ -cycles are labelled  $\mathcal{R}_1$  through  $\mathcal{R}_n$ . The removal of the  $m$  edges, belonging to the  $m$   $n$ -cycles, between any pair of adjacent  $m$ -cycles, say  $\mathcal{R}_i$  and  $\mathcal{R}_{i+1}$ , will break the  $n$ -cycles. If the  $m$  edges between another pair of adjacent  $m$ -cycles (of which neither is adjacent either to  $\mathcal{R}_i$  nor to  $\mathcal{R}_{i+1}$ ), say  $\mathcal{R}_j$  and  $\mathcal{R}_{j+1}$  is removed, then two graph components will result. Thus, for a fixed  $m \leq n$ , the bisection width is fixed. If  $n$  is therefore large enough, the right-hand side of (4.12) will be negative.

Another important aspect that prevents this method from being applied easily, is that the bisection width of the graph in question needs to be known, and unless this quantity is easily computed, other bounding techniques should rather be considered.

### 4.2.2.5 Edge set partitioning

Strictly speaking, the *edge set partitioning* method is used in the literature as a bounding technique only insofar as it may be used to prove that for an upper bound  $U(\mathcal{G})$  to the crossing number  $\nu(\mathcal{G})$  of a graph  $\mathcal{G}$ ,  $\nu(\mathcal{G}) \geq U(\mathcal{G})$ . This is normally achieved by the method of *proof by contradiction*. The upper bound value  $U(\mathcal{G})$  may as well be replaced by any value  $x$ , and therefore, this method may be used to prove the validity of lower bounds to the crossing number of a graph. In order to understand how to apply this method, some notation is necessary.

**Definition 4.2.5** For a graph  $\mathcal{G}$ , let  $A, B \subseteq E(\mathcal{G})$ , then for a drawing  $\phi$  of  $\mathcal{G}$ , let

$$\nu_\phi(A, B) = \sum_{\substack{a \in A \\ b \in B}} |\phi(\tilde{a}) \cap \phi(\tilde{b})|.$$

Also, let  $\nu_\phi(A, A) = \nu_\phi(A)$ . ■

Informally,  $\nu_\phi(A, B)$  denotes the number of crossings between every pair of edges where one edge is in  $A$ , and the other in  $B$ .

For three mutually disjoint subsets  $A, B, C \subset E(\mathcal{G})$ , the identities

$$\nu_\phi(A \cup B) = \nu_\phi(A) + \nu_\phi(B) + \nu_\phi(A, B) \text{ and} \tag{4.13}$$

$$\nu_\phi(A, B \cup C) = \nu_\phi(A, B) + \nu_\phi(B, C) \tag{4.14}$$

are noted.

### Applying the edge partition method to find the crossing number of a graph

To show that an upper bound  $U(\mathcal{G})$  is equal to the crossing number,  $\nu(\mathcal{G})$ , of  $\mathcal{G}$ , the edge set  $E(\mathcal{G})$  may be partitioned so that  $A \cup B = E(\mathcal{G})$ , followed by an assumption that  $\nu(\mathcal{G}) < U(\mathcal{G})$ . Since, for a drawing  $\phi$ ,

$$\nu_\phi(\mathcal{G}) = \nu_\phi(A) + \nu_\phi(B) + \nu_\phi(A, B),$$

it follows that if  $\phi$  is an optimal drawing of  $\mathcal{G}$  (*i.e.*, realizes the crossing number of  $\mathcal{G}$ ), then

$$\nu_\phi(A, B) < U(\mathcal{G}) - \nu_\phi(A) - \nu_\phi(B). \tag{4.15}$$

To complete the proof by contradiction, it remains to be shown that  $\nu_\phi(A, B) \geq U(\mathcal{G}) - \nu_\phi(A) - \nu_\phi(B)$  for every optimal drawing  $\phi$  of  $\mathcal{G}$ .

Unfortunately, the set of optimal drawings is unknown (since otherwise the crossing number problem would have been solved), which makes the problem of proving the converse of (4.15) as difficult as the original problem. If  $\Phi(A)$  denotes the set of all drawings of  $A$ , and  $\Phi(B)$  the set of all drawings of  $B$ , then as many as  $|\Phi(A)| \times |\Phi(B)|$  different right-hand sides in (4.15) might have to be considered, which is clearly impractical.

To reduce the number of cases to a handful, it is normally assumed that the subgraph  $\langle A \rangle$  is large and varies in size as  $\mathcal{G}$  varies, and that the subgraph  $\langle B \rangle$  is small and constant in size (an upper bound drawing construction which realises  $U(\mathcal{G})$  is typically a good guide as to how to choose  $A$  and  $B$ ). For example, in Asano's [Asa86] proof for the crossing number of  $\mathcal{K}_{1,3,n}$ , he chose  $\langle A \rangle \cong \mathcal{K}_{4,n}$  and  $\langle B \rangle \cong \mathcal{K}_{1,3}$ . In the proof for the crossing number of  $\mathcal{K}_{1,1,1,n}$  which will be presented later in this thesis, the subsets were chosen so that  $\langle A \rangle \cong \mathcal{K}_{4,n}$  and  $\langle B \rangle \cong \mathcal{K}_4$ . This, in

itself, does not change the nature of the problem, but if  $\nu(A)$  is known, then, since  $\nu(A) \leq \nu_\phi(A)$  for any drawing  $\phi$  of  $\langle A \rangle$ , it follows that

$$\nu_\phi(A, B) < U(\mathcal{G}) - \nu(A) - \nu_\phi(B), \quad (4.16)$$

where the drawing of  $\langle A \rangle$  is unspecified, and is determined by the choices of drawings of  $\langle B \rangle$  and the ways in which edges from  $A$  and  $B$  cross when considering the term  $\nu_\phi(A, B)$ .

This reduces the problem to enumeration of drawings of  $\langle B \rangle$  and showing that (4.16) is violated for every such drawing. Since  $\langle B \rangle$  is small,  $|\Phi(B)|$  should hopefully be relatively small. Of course, for a given drawing of  $\langle B \rangle$ , it might still be very difficult to prove that  $\nu_\phi(A, B) \geq U(\mathcal{G}) - \nu(A) - \nu_\phi(B)$ , since it might be difficult to enumerate the different ways in which edges from  $A$  cross edges from  $B$  when considering the term  $\nu_\phi(A, B)$ . When this is the case a method which seems first to have been employed by Asano [Asa86] may be useful, as described in the next section.

### Using vertex set partitioning to ease proof by contradiction

The inequality (4.16) constrains the way in which edges from  $A$  may be added to a drawing  $\phi$  of  $\langle B \rangle$ . If the subgraph  $\langle B \rangle$  has the property that  $\langle V(\langle B \rangle) \rangle \cong \langle B \rangle$  — that is, if the vertex set of  $\langle B \rangle$  induces the subgraph  $\langle B \rangle$  itself — the identity (4.14) may be used, so that one may write

$$\nu_\phi(A, B) \geq \sum_{v \in V(\mathcal{G}) \setminus V(\langle B \rangle)} \nu_\phi(B, E_{\langle B \rangle}(v)), \quad (4.17)$$

where  $E_U(v) = \{e : e \text{ joins } v \text{ with some } u \in V(U)\}$ .

In the sum on the right-hand side of (4.17), edges joining pairs of vertices that are both in  $V(\mathcal{G}) \setminus V(\langle B \rangle)$  are not considered. It is worth noting that in the proof for the crossing number of  $\mathcal{K}_{1,1,1,1,n}$ , presented later in this thesis, it does not occur that there are such edges, since the set  $V(\mathcal{G}) \setminus \langle B \rangle$  in that case is exactly the partite set with  $n$  vertices. Therefore, equality in (4.17) is attained, since all of the edges in  $A$  are considered. For the same reasons, equality in (4.17) is reached in both of Asano's proofs [Asa86]. Combination of (4.17) with (4.16), yields

$$\sum_{v \in V(\mathcal{G}) \setminus V(\langle B \rangle)} \nu_\phi(B, E_{\langle B \rangle}(v)) < U(\mathcal{G}) - \nu(A) - \nu_\phi(B).$$

For  $n = |V(\langle A \rangle)|$ , the right-hand side typically has the form  $\alpha n + \beta$ , where  $\beta < n$ , and  $\alpha, \beta \in \mathbb{N}$ . Ideally, in every drawing  $\phi$  of  $\langle B \rangle$ , the addition of a single vertex  $v \in V(\mathcal{G}) \setminus V(\langle B \rangle)$  will render the inequality  $\nu_\phi(B, E_{\langle B \rangle}(v)) > \alpha$ , which immediately leads to the desired result.

If, for some drawing  $\phi(\langle B \rangle)$ , this is not the case, then there is at least one vertex  $v$ , whose addition will cause at most  $\alpha$  crossings. Every drawing obtainable from  $\phi(\langle B \rangle)$  by the addition of  $v$  may be considered. Let  $B_2 = B \cup E_{\langle B \rangle}(v)$  and  $A_2 = E(\mathcal{G}) \setminus B_2$ . Then,

$$\sum_{v \in V(\mathcal{G}) \setminus V(\langle B_2 \rangle)} \nu_\phi(B_2, E_{\langle B_2 \rangle}(v)) < U(\mathcal{G}) - \nu(A_2) - \nu_\phi(B_2).$$

Now, the right-hand side has the form  $\alpha_2(n-1) + \beta_2$ , and one proceeds by the same argument as before, by attempting to show that the addition of any vertex from  $V(\mathcal{G}) \setminus V(\langle B_2 \rangle)$  to every drawing of  $B_2$  will cause more than  $\alpha_2$  crossings. Clearly this method may be applied as many times as necessary.

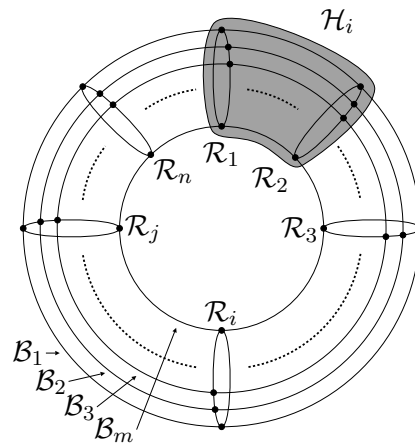


Figure 4.7: Construction for an upper bound to  $\nu(\mathcal{C}_m \times \mathcal{C}_n)$ .

#### 4.2.2.6 A sparse graph technique: bounding $\nu(\mathcal{C}_m \times \mathcal{C}_n)$ from below

From the drawing of  $\mathcal{C}_m \times \mathcal{C}_n$  in Figure 4.7, it may be seen that if the coordinate  $m$  is fixed, the number of smaller  $m$ -cycles (marked  $\mathcal{R}_1$  through  $\mathcal{R}_n$  in Figure 4.7) would vary with  $n$ , and conversely, if  $n$  is fixed, the number of larger cycles (marked  $\mathcal{B}_1$  through  $\mathcal{B}_n$  in Figure 4.7) would vary with  $m$ . These  $n$  small cycles and  $m$  large cycles are called the *principal cycles* of  $\mathcal{C}_m \times \mathcal{C}_n$ , and it is normally interactions between principal cycles that are studied when studying the crossing number properties of  $\mathcal{C}_m \times \mathcal{C}_n$ . Following the idea of Beineke and Ringelsen [BR80] to render the conceptual task of differentiating between the different types of principal cycles manageable, the edges of the  $n$  principal small cycles are coloured red, and the edges of the  $m$  principal large cycles are coloured blue. The naming of the labels in Figure 4.7 were chosen to reflect this colouring.

The fact that  $\mathcal{C}_m \times \mathcal{C}_n$  is 4-regular, and the fact that crossings in a graph are conveniently modelled as artificial vertices of degree 4, makes it possible to obtain a different perspective of a drawing  $\phi$  of  $\mathcal{C}_m \times \mathcal{C}_n$  —  $\phi$  is a drawing of two families of closed curves, say  $R$  and  $B$ , where  $R$  has  $n$  red curves, and  $B$  has  $m$  blue curves, and where each red curve in  $R$  crosses each blue curve in  $B$  at least once. In this perspective,  $mn$  of the crossings of the curves, are not, in fact, crossings, but correspond to vertices in  $\mathcal{C}_m \times \mathcal{C}_n$ . This was the idea of Richter and Thomassen [RT95], who introduced the notion of a *curve system*, and used a particular type of curve system, called an  $(m, n)$ -mesh to provide proofs for the crossing numbers of  $\mathcal{C}_4 \times \mathcal{C}_4$  and  $\mathcal{C}_5 \times \mathcal{C}_5$ .

**Definition 4.2.6** *An  $(m, n)$ -mesh is a pair  $(R, B)$  of families of respectively red and blue closed curves (i.e., the beginning of the curve meets its end), where  $R$  contains  $n$  such curves, and  $B$  contains  $m$  such curves, so that*

1. every curve in  $R$  intersects every curve in  $B$ ,
2. no point in the plane is covered thrice or more, although a curve may cover a single point twice. ■

The number of crossings in an  $(m, n)$ -mesh is denoted  $i^*(R, B)$ , and crossings are counted for

1. the number of crossings between  $R$  and  $B$ , where, for every two curves  $\mathcal{R}_i \in R$  and  $\mathcal{B}_j \in B$ ,  $|\mathcal{R}_i \cap \mathcal{B}_j|$  crossings are counted,



2. the number of crossings of curves over themselves, *i.e.*, where a single curve crosses a point twice.

From the preceding discussion,  $mn$  of the crossings in a drawing of an  $(m, n)$ -mesh, correspond to vertices in a drawing of  $\mathcal{C}_m \times \mathcal{C}_n$ . Thus, if  $i^*(R, B) \geq mn + \alpha$ , then  $\nu(\mathcal{C}_m \times \mathcal{C}_n) \geq \alpha$ .

The advantage gained from the curve system perspective is that vertices and edge crossings are treated uniformly. This allows for more elegant topological arguments in the proofs of the crossing numbers of  $\mathcal{C}_m \times \mathcal{C}_n$ . As an example of why this works, a common technique, which was first employed by Ringelsen and Beineke [RB78] in the context of toroidal grid graphs (although they considered graphs, and not curve systems), will be considered. Let  $\mathcal{H}_i$  be the subgraph of  $\mathcal{C}_m \times \mathcal{C}_n$  induced by the vertices of two red principal cycles  $\mathcal{R}_i$  and  $\mathcal{R}_{i+1}$  (indices are modulo  $n$ ), and define the *force*  $f(\mathcal{H}_i)$  of  $\mathcal{H}_i$  as the sum of crossings that are caused by

1. self intersections of  $\mathcal{R}_i$ ,
2. crossings between blue edges in  $\mathcal{H}_i$  with red edges in  $\mathcal{H}_i$ ,
3. crossings between blue edges in  $\mathcal{H}_i$ ,
4. crossings of blue edges in  $\mathcal{H}_i$  and blue edges in  $\mathcal{H}_{i+1}$ .

From this it may be seen that a crossing between a pair of edges is counted exactly once, and that  $\sum_{i=1}^n f(\mathcal{H}_i) \geq mn + \nu(\mathcal{C}_m \times \mathcal{C}_n)$ .

Shahrokhi, Sýkora, Székely and Vrto [SSSV] have applied this concept to obtain some good lower bounds for  $\mathcal{C}_m \times \mathcal{C}_n$  in the plane, the projective plane, and the Klein bottle.

#### 4.2.2.7 Obtaining bounds for other sparse graphs

Finding a general method of bounding sparse graph crossing numbers is quite difficult. The standard counting method (§ 4.2.2.1) mostly fails, because it is hard to find subgraphs with a high crossing number. The graph-to-graph embedding method (§ 4.2.2.2) may be considered, but a large vertex congestion would impact negatively on the quality of the lower bound. This problem would be mitigated by embedding a sparse graph into the sparse graph under consideration. The problem with application of the graph bisection method, is that a given class of sparse graphs typically has a constant bisection width with respect to variation of its size in certain ways — this was already discussed in § 4.2.2.4 for  $\mathcal{C}_m \times \mathcal{C}_n$ . A priori, there is no reason why the graph minor method cannot work for some cases, but of course, the crossing numbers of subgraphs may often provide better lower bounds than the minor method.

Although the idea of a mesh does not generalise well to other classes of sparse graphs, some ideas from the proofs of the crossing numbers of toroidal grid graphs have been applied successfully to finding the crossing numbers of other products of graphs. See, for example, the proof of Beineke and Ringelsen [BR80], showing that  $\nu(\mathcal{K}_4 \times \mathcal{C}_n) = 3n$ , and the technique that is described in the next section. For products of graphs, principal cycles are generalised to principal subgraphs ( $\mathcal{K}_4$  and  $\mathcal{C}_n$  in the case mentioned). For many other interesting sparse graphs, one finds that the maximum vertex degrees are larger than 4. One could, in principle, define a new type of mesh, say for graphs with maximum degree six, where up to three curves in the mesh may cover a particular point, since a vertex with degree six may be modelled as the intersection of three lines. It soon becomes apparent that this approach has the disadvantage that a lower bound for the total number of crossings in such a mesh might count only one crossing where three curves

intersect, although the particular intersection might correspond to the intersection of three edges (thus, three crossings should actually be counted), or it might correspond to an edge drawn over a vertex with degree four (thus, at least a single crossing, and at most four crossings are not counted).

#### 4.2.2.8 A typical proof technique demonstrated for the result $\nu(\mathcal{C}_m \times \mathcal{C}_n) = m(n - 2)$

Nothing more than a very quick sketch of the proof techniques used for the proofs of the crossing number results for the products of cycles is provided in this section.

For a fixed value of  $m$ , the main idea is to use induction on  $n$ , where the base case of  $\nu(\mathcal{C}_m \times \mathcal{C}_m)$  is known. From the drawing in Figure 4.7, it may be seen that every curve  $\mathcal{R}_i$  is crossed a total of  $m - 2$  times, and although this is an upper bound construction, the question of whether at least one red edge is crossed at least  $m - 2$  times is central in the proof technique.

By the induction hypothesis,  $\nu(\mathcal{C}_m \times \mathcal{C}_{n-1}) = (m - 2)(n - 1)$ , and if there exists at least one red cycle  $\mathcal{R}_i$  which is crossed at least  $m - 2$  times, then deletion of its edges from  $\mathcal{C}_m \times \mathcal{C}_n$  gives a subdivision of  $\mathcal{C}_m \times \mathcal{C}_{n-1}$ . Thus, it follows that the drawing of  $\mathcal{C}_m \times \mathcal{C}_n$  must have at least  $(m - 2)(n - 1) + m - 2 = (m - 2)n$  crossings.

If every red cycle is crossed fewer than  $m - 2$  times, it must be shown for such a drawing  $\phi$ , that  $\nu_\phi(\mathcal{C}_m \times \mathcal{C}_n) \geq (m - 2)n$ . First it is shown that if some red cycles are drawn inside other red cycles, or if some red cycles cross each other, that one of the red cycles involved will be crossed at least  $m - 2$  times. This implies that all the red cycles are disjoint in  $\phi$ . Now, the notion of the *force* of a the subgraph induced by the vertices of two consecutive (with respect to their indices) red cycles (as defined in § 4.2.2.6) may be used to perform that  $\nu_\phi(\mathcal{C}_m \times \mathcal{C}_n) \geq (m - 2)n$ .

A very useful result due to Juarez and Salazar [Sal99], allows one to avoid having to show the last part of the proof. A proper crossing in a principal cycle is a crossing not caused by two edges from the cycle itself.

**Theorem 4.2.2** (Juarez and Salazar [Sal99]) *Let  $m, n$  be integers such that  $m \geq n \geq 3$ . Then every drawing of  $\mathcal{C}_m \times \mathcal{C}_n$  such that either the principal  $n$   $m$ -cycles are pairwise disjoint or the principal  $m$   $n$ -cycles are pairwise disjoint, has at least  $(m - 2)n$  crossings. ■*

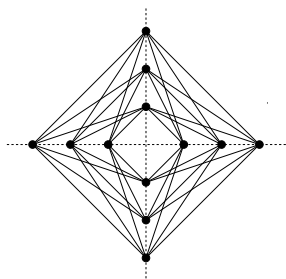
### 4.2.3 Crossing number bounds

This subsection contains bounds on the crossing numbers of various classes of graphs, expressed in terms of analytical functions.

#### 4.2.3.1 General crossing number bounds

In the functions provided here, the crossing number bounds for general graphs are expressed in terms of easily computable graph parameters. As such, the bounds cannot be expected to be very good.

$$\nu(\mathcal{G}) \geq \begin{cases} \frac{c-3}{c^3} |E(\mathcal{G})|^3 / |V(\mathcal{G})|^2 & \text{if } |E(\mathcal{G})| \geq c|V(\mathcal{G})|, \ c > 3 & (a) \\ \frac{1}{33.75} |E(\mathcal{G})|^3 / |V(\mathcal{G})|^2 & \text{if } |E(\mathcal{G})| \geq 4|V(\mathcal{G})| & (b) \\ |E(\mathcal{G})| - [g(\mathcal{G}) / (g(\mathcal{G}) - 2)] (|V(\mathcal{G})| - 2) & \text{where } g(\mathcal{G}) \text{ is the girth of } \mathcal{G} & (c) \\ 0 & \text{otherwise} & (d) \end{cases}$$

Figure 4.8:  $\mathcal{K}_{6,6}$  bipartite construction.

The lower bound (a) was independently proved by Leighton [Lei83] and by Ajtai, Chvátal, Newbron and Szemerédi [ACNS82]. They were unaware that this bound was already conjectured by Erdős and Guy [EG73]. The maximum factor of  $1/64$  in (a) is obtained by setting  $c = 4$ . Pach and Tóth [PT97] improved the coefficient for  $c = 4$  to  $1/33.75$  in (b).

The lower bound (c) is due to Kainen<sup>6</sup> [Kai72]. His result is a generalization of the case where  $g(\mathcal{G}) = 3$ , and where each crossing is viewed as an artificial vertex of degree 4.

As a general upper bound, it is obvious, for a graph  $\mathcal{G}$  on  $n$  vertices, that  $\nu(\mathcal{G}) \leq \nu(\mathcal{K}_n)$ . An upper bound for  $\mathcal{K}_n$  may be found in the following subsections.

#### 4.2.3.2 Multipartite graphs

##### Bipartite graphs

This problem probably has the longest and one of the most interesting histories of all the crossing number problem special cases. In fact, it is exactly a question posed by P. Turán (see § 1.1) about the crossing number of bipartite graphs that spawned the field of research on crossing numbers. Let

$$L(m, n) = \frac{1}{5}(m)(m-1) \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor, \quad (4.18)$$

$$L'(m, n) = \frac{1}{5}(m)(m-1) \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor + 9.9 \times 10^{-6} m^2 n^2 \text{ and} \quad (4.19)$$

$$U(m, n) = \left\lfloor \frac{m}{2} \right\rfloor \left\lfloor \frac{m-1}{2} \right\rfloor \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor. \quad (4.20)$$

Then

$$\max\{L(m, n), \nu(\mathcal{K}_{s,t})\} \leq \left. \begin{array}{l} L'(m, n) \leq \\ \nu(\mathcal{K}_{m,n}) \leq \end{array} \right\} \left\{ \begin{array}{ll} = 0 & m \leq 2 \text{ and } m \leq n \\ = U(m, n) & 3 \leq m \leq 6 \text{ and } m \leq n; 7 \leq m \leq n \leq 10 \\ \leq U(m, n) & \text{otherwise,} \end{array} \right.$$

where  $s$  and  $t$  on the left-hand side are values for which  $\nu(\mathcal{K}_{s,t})$  is known, and  $s \leq t$ ,  $s \leq m$  &  $t \leq n$ .

Zarankiewicz [Zar54] originally proved the case for  $m = 3$  as the base case for his inductive proof, to show equality to the upper bound in (4.20). An uncorrectable flaw was later discovered in the

<sup>6</sup>Kainen actually proved in [Kai72], that  $\nu_{\Upsilon}(\mathcal{G}) \geq |E(\mathcal{G})| - g(\mathcal{G})/(g(\mathcal{G}) - 2)(|V(\mathcal{G})| - 2 + 2\Upsilon)$  where  $\Upsilon$  is the genus (§ 4.1.4) of  $\mathcal{G}$ .

inductive part of his proof (which was still valid for the inductive base case where  $m = 3$ ) — this is discussed in Guy [Guy69]. The case for  $m = 4$  follows from the standard counting method when finding copies of  $\mathcal{K}_{3,n}$  in  $\mathcal{K}_{4,n}$  — the lower bound then meets the upper bound. The cases for  $m = 5, 6$  were proved by Kleitman [Kle70]. Woodall [Woo93] employed the help of computers to show that  $\nu(\mathcal{K}_{m,n})$  is equal to (4.20) for  $7 \leq m \leq n \leq 10$ .

The upper bound is due to Zarankiewicz [Zar54]. This upper bound may be realised as follows: For a bipartite graph  $\mathcal{K}_{m,n}$ , the vertices of the partite set with cardinality  $m$  are placed on the  $x$ -axis of a two-dimensional Cartesian system of axes, so that there are  $\lceil m/2 \rceil$  vertices on the positive side of the axis, and  $\lfloor m/2 \rfloor$  vertices on its negative side. The same is done for the other partite set (*i.e.*, the partite set with cardinality  $n$ ), but the vertices are placed on the  $y$ -axis. Finally, the edges are drawn as straight lines. An example of a drawing of  $\mathcal{K}_{6,6}$  using this method may be seen in Figure 4.8.

The first lower bound,  $L'(m, n)$ , is true if  $m$  and  $n$  are sufficiently large. This result is due to Nahas [Nah03], who did not provide bounds on  $m$  and  $n$ ; such bounds would be valuable for improving the lower bound on  $\nu(\mathcal{K}_{m,n})$ .

The second lower bound  $L(m, n)$ , which is only valid for  $m \geq 6$ , follows from the standard counting method (Kleitman's [Kle70] version is followed): There are  $m$  copies of  $\mathcal{K}_{m-1,n}$  in  $\mathcal{K}_{m,n}$ ,  $m \leq n$ . Each crossing is counted in  $m - 2$  copies of  $\mathcal{K}_{m-1,n}$ , since for each crossing there are two subgraphs isomorphic to  $\mathcal{K}_{m-1,n}$ , each of which will lack a vertex incident to an edge involved in the crossing. Thus it follows that

$$\begin{aligned} \nu(\mathcal{K}_{m,n}) &\geq \frac{m}{m-2} \nu(\mathcal{K}_{m-1,n}) \\ &\geq \frac{\prod_{i=7}^m i(i-2)}{\prod_{i=5}^{m-2} i(i-2)} \nu(\mathcal{K}_{6,n}) \\ &= \frac{m(m-1)}{6 \times 5} 6 \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \\ &= \frac{m(m-1)}{5} \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor. \end{aligned}$$

Kleitman also showed in [Kle70], that the crossing number of  $\mathcal{K}_{m+1,n}$  follows immediately if the crossing number of  $\mathcal{K}_{m,n}$  is known, for  $m$  odd. Thus, the minimum counter example for equality in (4.20) must have  $m$  odd.

De Klerk, Maharry, Pasechnik, Richter and Salazar [dKMP<sup>+</sup>04] have shown that for a fixed value of  $m \geq 9$ ,

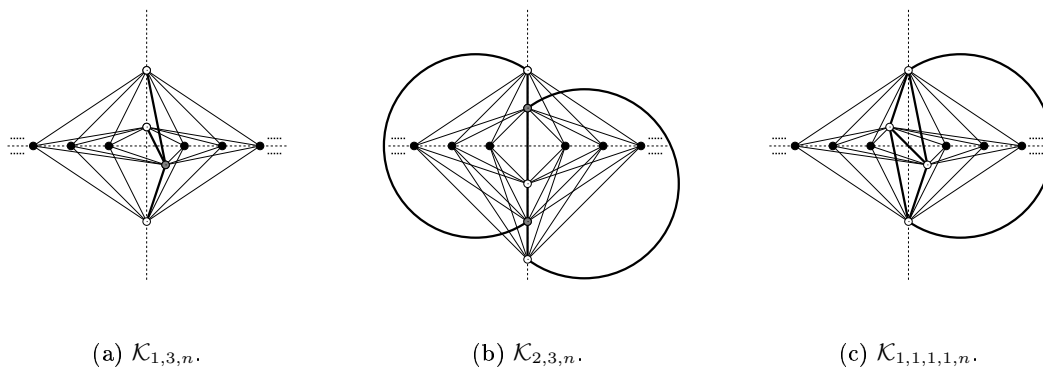
$$\lim_{n \rightarrow \infty} \frac{\nu(\mathcal{K}_{m,n})}{\left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \left\lfloor \frac{m}{2} \right\rfloor \left\lfloor \frac{m-1}{2} \right\rfloor} \geq \frac{0.83m}{m-1}. \tag{4.21}$$

It may be verified that if  $\nu(\mathcal{K}_{m,n})$  is replaced by  $L(m, n)$  in (4.21), that a smaller fraction than  $0.83m/(m-1)$  is obtained.

### Multipartite graphs, and derivative graphs

Relatively little is known about the crossing number of multipartite graphs. The only general results for non-bipartite multipartite graphs are due to Asano [Asa86], who showed that

$$\nu(\mathcal{K}_{1,3,n}) = \nu(\mathcal{K}_{4,n}) + \left\lfloor \frac{n}{2} \right\rfloor,$$

Figure 4.9: Optimal drawings of  $\mathcal{K}_{1,3,n}$ ,  $\mathcal{K}_{2,3,n}$  and  $\mathcal{K}_{1,1,1,1,n}$ .

and that

$$\nu(\mathcal{K}_{2,3,n}) = \nu(\mathcal{K}_{5,n}) + n.$$

He used the edge set partitioning method, and partitioned  $\mathcal{K}_{1,3,n}$  into  $\mathcal{K}_{1,3}$  &  $\mathcal{K}_{4,n}$ , and he partitioned  $\mathcal{K}_{2,3,n}$  into  $\mathcal{K}_{2,3}$  &  $\mathcal{K}_{5,n}$ . Drawing constructions of  $\mathcal{K}_{1,3,n}$  and  $\mathcal{K}_{2,3,n}$  which realize the crossing numbers of those graphs may be seen in Figures 4.9(a) and (b) respectively. The thick lines in Figure 4.9(a) belong to the subgraph  $\mathcal{K}_{1,3}$  — only one of the thick lines is crossed, and it is crossed by edges that are joined to the right half (with respect to the drawing, and the vertical dotted line) of the black vertices. It may be drawn to fall on the side with fewest black vertices (which is the right in this case), and therefore, an extra  $\lfloor \frac{n}{2} \rfloor$  crossings are incurred by its presence. The thick lines in Figure 4.9(b) are the edges which belong to the subgraph  $\mathcal{K}_{2,3}$ . A total of  $n$  edges which are joined to all  $n$  black vertices are crossed in total by the two thick curved edges. Therefore,  $n$  crossings are incurred by their presence. His method may be applied generally, but the number of cases to consider for larger graphs quickly become impractical to accommodate by hand — perhaps a computer implementation may be suitable for the larger cases. It will be shown later in this thesis that

$$\nu(\mathcal{K}_{1,1,1,1,n}) = \left\lfloor \frac{n^2 + 1}{2} \right\rfloor,$$

by the method used by Asano in [Asa86], where the graph  $\mathcal{K}_{1,1,1,1,n}$  is partitioned into  $\mathcal{K}_4$  and  $\mathcal{K}_{4,n}$ . The optimal drawing for this graph is shown in Figure 4.9(c), and the thick lines show the  $\mathcal{K}_4$ .

Klešč [Kle01b] found the crossing numbers of two graphs  $\mathcal{H}_1$  and  $\mathcal{H}_2$ , which are obtained from  $\mathcal{K}_{1,1,1,1,1,n}$  respectively by deletion of a single edge from the  $\mathcal{K}_5$  subgraph, and deletion of two adjacent edges from the  $\mathcal{K}_5$  subgraph. The result follows as:

$$\nu(\mathcal{H}_1) = \nu(\mathcal{H}_2) = \nu(\mathcal{K}_{5,n}) + 2 \left\lfloor \frac{n}{2} \right\rfloor.$$

#### 4.2.3.3 Results for $\mathcal{K}_n$

Let

$$L(n) = \frac{n(n-1)(n-2)(n-3)}{20(n-6)(n-7)} \left\lfloor \frac{(n-6)}{2} \right\rfloor \left\lfloor \frac{(n-7)}{2} \right\rfloor \quad \text{and} \quad (4.22)$$

$$U(n) = \frac{1}{4} \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \left\lfloor \frac{n-2}{2} \right\rfloor \left\lfloor \frac{n-3}{2} \right\rfloor. \quad (4.23)$$

Then,

$$\max\{L(n), U(10)\} \leq \nu(\mathcal{K}_n) \begin{cases} = 0 & n \leq 4 \\ = U(n) & 5 \leq n \leq 10 \\ \leq U(n) & \text{otherwise.} \end{cases} \quad (4.24)$$

Saaty [Saa69] showed that equality holds in (4.24) for  $6 \leq n \leq 10$ , using a series of *ad hoc* arguments. The upper bound derives from a well-known drawing construction of  $\mathcal{K}_n$  due to Blažek & Koman [BK64] and Guy [Guy69]. It seems rather uncertain as to whom actually originally found the upper bound, and it is probably folklore.

The upper bound construction is based on a drawing of  $\mathcal{K}_n$  on a cylinder, or a “soupcan” —  $\lfloor n/2 \rfloor$  vertices are placed, evenly spaced around the rim on one lid, and  $\lceil n/2 \rceil$  vertices on the other. All vertices on both lids are joined pairwise by means of straight lines. To join vertices from one lid to the other, the shortest distance around the perimeter of the soupcan is chosen. An example of such a drawing of  $\mathcal{K}_{10}$  may be seen in Figure 4.10(a), and a plane version where the “soupcan” has been folded flat is shown in Figure 4.10(b).

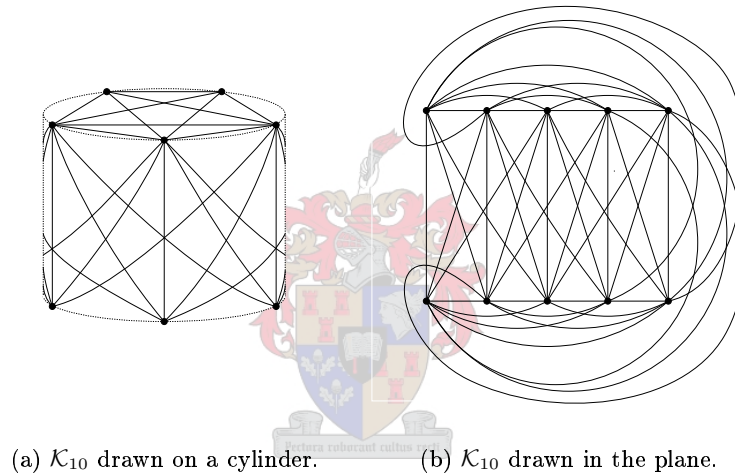


Figure 4.10: An upper bound construction for  $\nu(\mathcal{K}_n)$ .

The lower bound is due to White and Beineke [WB78]. They used the standard counting method by finding copies of  $\mathcal{K}_{6,n-6}$  in  $\mathcal{K}_n$ . They actually used a general method by finding copies of  $\mathcal{K}_{r,n-r}$ , but since  $r = 6$  is the best known exact result for the crossing number of  $\mathcal{K}_{r,p}$ , the lower bound is limited by it.

It follows that as  $n \rightarrow \infty$ ,  $L(n) \rightarrow \frac{1}{80}n^4$ ,  $U(n) \rightarrow \frac{1}{64}n^4$ , and therefore  $\frac{1}{80} \leq \lim_{n \rightarrow \infty} \nu(\mathcal{K}_n)/n^4 \leq \frac{1}{64}$ , if the limit exists. Kainen [Guy69] showed that if  $\nu(\mathcal{K}_{m,n})$  is known for all  $m, n \in \mathbb{N}$ , then  $\lim_{n \rightarrow \infty} \nu(\mathcal{K}_n)/n^4 = \frac{1}{64}$ . Recently, De Klerk, Maharry, Pasechnik, Richter and Salazar [dKMP<sup>+</sup>04] showed that

$$\lim_{n \rightarrow \infty} \frac{\nu(\mathcal{K}_n)}{\frac{1}{4} \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor} \geq 0.83,$$

which implies that  $\frac{1}{80} < 0.012968 \leq \lim_{n \rightarrow \infty} \nu(\mathcal{K}_n)/n^4$ .

#### 4.2.3.4 Progress on crossing numbers of products of graphs

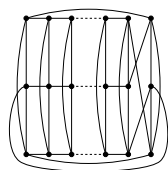
The best-known crossing number problem in the class of products of graphs, relates to the toroidal grid graph, or product of cycles. However, much research has also been done for products of small graphs with paths, cycles and stars.

### Bounds and exact results for $\mathcal{C}_m \times \mathcal{C}_n$ and variations of $\mathcal{C}_m \times \mathcal{C}_n$

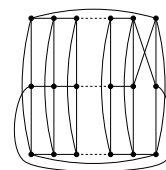
This particular class of graphs has received a fair amount of attention in the literature of the past few years. The fact that some of the most experienced researchers have not had much success in obtaining anything more than a few results, is testimony to the difficulty of the crossing number problem for even very special classes of graphs. Meyers [Mye98] gives an entertaining survey on this particular problem. The best known results for this problem are shown in Table 4.1.

$m$	$n$	$\nu(\mathcal{C}_m \times \mathcal{C}_n)$	Proved by
3	3	= 3	Ringeisen and Beineke [RB78]
3	$n$	= $n$	Ringeisen and Beineke [RB78]
4	4	= 8	Dean & Richter [DR95]; Eggleton & Guy [EG70]
4	$n$	= $2n$	Beineke & Ringeisen [BR80]
5	5	= 15	Richter and Thomassen [RT95]
5	$n$	= $3n$	Klešč, Richter, Stobert [KRS96]
6	6	= 24	Anderson, Richter and Rodney [ARR96]
6	$n$	= $4n$	Richter and Salazar [RS01]
7	7	= 35	Anderson, Richter and Rodney [ARR97]
$m$	$n$	= $m(n-2)$ $n \geq m(m+1)$	Glebsky and Salazar [SG04]
$m$	$n$	$\geq (1/2)m(n-2)$	Juarez and Salazar [JS01]
$m$	$n$	$\geq (0.8 - \varepsilon)mn$ $\varepsilon > 0$ $m$ sufficiently large	Salazar and Ugalde [SU04]
$m$	$n$	$\geq (3/5)mn$ if $n \leq (5/4)(m-1)$	Shahrokhi, Sýkora, Székely and Vrto [SSSV]
$m$	$n$	$\leq m(n-2)$	Harary, Kainen and Schwenk [HKS73]

Table 4.1: Crossing number results for the product of cycles  $\mathcal{C}_m \times \mathcal{C}_n$ .



(a) The twisted toroidal grid graph  $\mathcal{T}_{3,n}$ .



(b) The crossed toroidal grid graph  $\mathcal{X}_{3,n}$ .

Figure 4.11: Two variations on the toroidal grid graph  $\mathcal{C}_3 \times \mathcal{C}_n$ .

Two variations to the product of cycles  $\mathcal{C}_m \times \mathcal{C}_n$  are the so-called *twisted toroidal grid graph*  $\mathcal{T}_{m,n}$ , and the *crossed toroidal grid graph*  $\mathcal{X}_{m,n}$ . Analytical descriptions of these graphs would be rather cumbersome, and since the only known analytical results for the crossing numbers of these graphs are for  $\mathcal{T}_{3,n}$  and for  $\mathcal{X}_{3,n}$ , only drawings of these graphs are shown in Figure 4.11. Proofs for the results

$$\nu(\mathcal{T}_{3,n}) = n \quad \text{and} \quad \nu(\mathcal{X}_{3,n}) = n$$

are due to Foley, Krieger, Riskin and Stanton [FKRS02].

**Bounds and exact results for miscellaneous products**

$\mathcal{G}_j$		$\nu(\mathcal{G}_j \times \mathcal{P}_n)$		$\nu(\mathcal{G}_j \times \mathcal{C}_n)$		$\nu(\mathcal{G}_j \times \mathcal{S}_n)$	
$\mathcal{G}_1$		0	$n \geq 0$	0	$n \geq 0$	$2 \lfloor \frac{(n-1)^2}{4} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_2$		$n - 1$	$n \geq 1$ [JŠ82]	$n$	$n \geq 1$ [JŠ82]	$2 \lfloor \frac{(n-1)^2}{4} \rfloor + \lfloor \frac{n}{2} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_3$		$n - 1$	$n \geq 1$ [Kle94]	$n$	$n \geq 3$ [BR80]	$2 \lfloor \frac{(n-1)^2}{4} \rfloor + \lfloor \frac{n}{2} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_4$		0	$n \geq 0$	$2n$	$n \geq 4$ [BR80]	$2 \lfloor \frac{(n-1)^2}{4} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_5$		$n - 1$	$n \geq 1$ [Kle94]	$2n$	$n \geq 4$ [BR80]	$2 \lfloor \frac{(n-1)^2}{4} \rfloor + \lfloor \frac{n}{2} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_6$		$2n$	$n \geq 1$ [Kle94]	$3n$	$n \geq 3$ [BR80]	$2 \lfloor \frac{(n+1)^2}{4} \rfloor$	$n \geq 1$ [Kle94]
$\mathcal{G}_7$		0	$n \geq 0$	0	$n \geq 0$		
$\mathcal{G}_8$		$2(n - 1)$	$n \geq 0$ [Kle91]	$2n$	$n > 5$ [Kle91]		
$\mathcal{G}_9$		$n - 1$	$n \geq 0$ [Kle01a]	$n$	$n > 5$ [Kle01a]		
$\mathcal{G}_{10}$		$n - 1$	$n \geq 0$ [Kle01a]	$n$	$n \geq 0$ [Kle01a]		
$\mathcal{G}_{11}$		$n - 1$	$n \geq 0$ [Kle01a]	$n$	$n \geq 0$ [Kle01a]		
$\mathcal{G}_{12}$		$2(n - 1)$	$n \geq 0$ [Kle01a]	$2n$	$n > 5$ [Kle01a]		
$\mathcal{G}_{13}$		$n - 1$	$n \geq 0$ [Kle01a]	$2n$	$n > 3$ [Kle01a]		
$\mathcal{G}_{14}$		0	$n \geq 0$	$3n$	$n > 4$ [KRS96]		

Table 4.2: Crossing number results for products with small graphs.

The results in this section are numerous. Most of the results in Table 4.2 have relatively long, *ad hoc* proofs, and again, one is left with an impression of the difficulty of the crossing number problem.

Note that  $\mathcal{G}_1 \times \mathcal{P}_n$  and  $\mathcal{G}_7 \times \mathcal{P}_n$  are just grid graphs, and they are consequently planar. The graphs  $\mathcal{G}_1 \times \mathcal{C}_n$ ,  $\mathcal{G}_7 \times \mathcal{C}_n$ ,  $\mathcal{G}_4 \times \mathcal{P}_n$  and  $\mathcal{G}_{14} \times \mathcal{P}_n$  may be drawn with no crossings, by the construction shown in Figure 4.12. Furthermore, it should also be noted that the crossing number results for some graphs in the table follow directly from the crossing number results of subgraphs, if a drawing of the graph in question can be made from the subgraph, without causing any crossings — see for example  $\mathcal{G}_{22}$  and  $\mathcal{G}_{24}$ .
















$\mathcal{G}_j$		$\nu(\mathcal{G}_j \times \mathcal{P}_n)$		$\nu(\mathcal{G}_j \times \mathcal{C}_n)$		$\nu(\mathcal{G}_j \times \mathcal{S}_n)$	
$\mathcal{G}_{15}$		$2(n-1)$	$n \geq 0$	$2n$	$n > 5$ [Kle01a]		
$\mathcal{G}_{16}$		$2n$	$n \geq 0$ [Kle96]	$= 9, \quad n = 3$ $\leq 4n \quad n \geq 4$	$n \geq 4$ [Kle02]	$4 \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor + 2n$	$n \geq 0$ [Kle96]
$\mathcal{G}_{17}$		$2(n-1)$	$n \geq 0$ [Kle01a]			$n(n-1)$	$n \geq 0$ [Kle01b]
$\mathcal{G}_{18}$		$2(n-1)$	$n \geq 0$ [Kle95]	$2n$	$n \geq 0$ [Kle01a]		
$\mathcal{G}_{19}$		$n-1$	$n \geq 0$ [Kle01a]	$3n$	$n > 4$ [Kle01a]		
$\mathcal{G}_{20}$		$2(n-1)$	$n \geq 0$ [Kle01a]	$3n$	$n > 4$ [Kle01a]	$n(n-1)$	$n \geq 0$ [Kle01b]
$\mathcal{G}_{21}$		$3n-1$	$n \geq 0$ [Kle95]				
$\mathcal{G}_{22}$		$3n-1$	$n \geq 0$ [Kle99b]	$2(n + \lfloor \frac{n+1}{2} \rfloor)$	$n \geq 0$ [Kle99b]		
$\mathcal{G}_{23}$		$2n$	$n \geq 0$ [Kle01a]			$4 \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor + 2n$	$n \geq 0$ [Kle01a]
$\mathcal{G}_{24}$		$3n-1$	$n \geq 0$ [Kle95]				
$\mathcal{G}_{25}$		$3n-1$	$n \geq 0$ [Kle01a]				
$\mathcal{G}_{26}$		$4n$	$n \geq 0$ [Kle01a]				
$\mathcal{G}_{27}$		$6n$	$n \geq 0$ [Kle99a]				

Table 4.2 (continued): Crossing number results for products with small graphs.

#### 4.2.3.5 The hypercube and interconnection graphs

Parallel computation has become an important topic in recent times. The hypercube, which has been used as the basis of interconnection schemes, has a well understood structure — networks designed according to this structure have a high level of fault tolerance, since the hypercube is a highly connected graph, and because it has a highly symmetrical structure. Hypercube-like networks, and a host of other graphs which also offer recursive structures are of particular interest in parallel computation. The interested reader is referred to Leighton’s book [Lei92], and the proceedings on hypercube multicomputers [Hea87]. Cimikowski [Cim02] approximated the crossing numbers for a host of these interconnection graphs (amongst others) by means of his two-page layout algorithms, which are described later in this chapter.

The hypercube of dimension  $d$  is denoted  $\mathcal{Q}_d$ , and it is a  $d$ -regular graph such that  $|V(\mathcal{Q}_d)| = 2^d$  and  $|E(\mathcal{Q}_d)| = d2^{d-1}$ . Every vertex is labelled with a unique  $d$ -bit binary string, and two vertices are adjacent if their respective labels differ in a single bit. A more intuitive method of constructing the hypercube  $\mathcal{Q}_d$  is to “extrude” the hypercube  $\mathcal{Q}_{d-1}$ . This “extrusion” effect may be seen clearly for  $\mathcal{Q}_4$  in Figure 4.13(d), which is an extruded version of the cube  $\mathcal{Q}_3$  in Figure 4.13(c), which, in turn, is an extruded version of the square  $\mathcal{Q}_2$ .

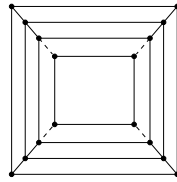


Figure 4.12:  $\mathcal{P}_m \times \mathcal{C}_n$  is planar.

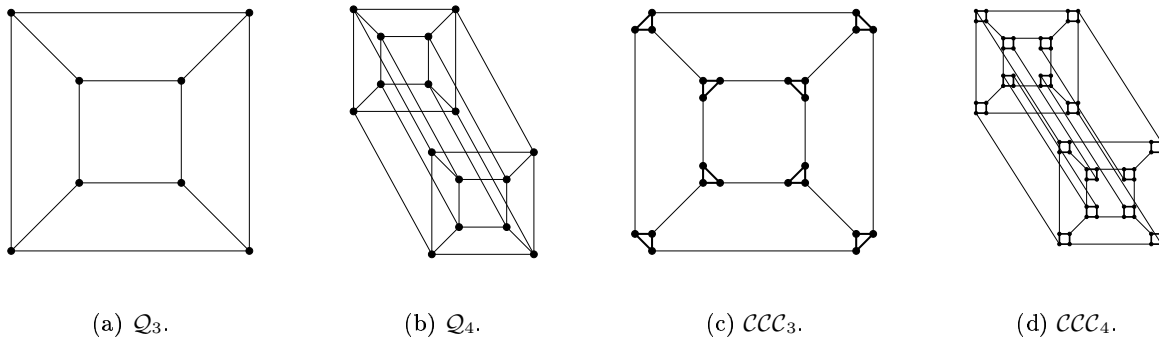


Figure 4.13: Drawings of hypercube and cube connected cycle networks.

Sýkora and Vrřo used the embedding technique in [SV93], to find a lower bound to  $\mathcal{Q}_d$ . The previously known best lower bound, due to Madej [Mad91], was  $\alpha 2^{d+0.215 \log^2 d} < \nu(\mathcal{Q}_d)$  for  $\alpha > 0$ . This improvement is a good example of the power of the graph-to-graph embedding method. The results known for the crossing number of the hypercube may be seen in Table 4.3.

### Other interconnection graphs

It is interesting to note that virtually all of the graphs presented in this section have a highly recursive structure. Only the first class, the so-called class of “cube-connected-cycle” graphs, is a derivative of the hypercube.

**Cube-connected-cycle graphs:** The cube-connected-cycles  $CCC_d$  of dimension  $d$  is obtained from the hypercube  $\mathcal{Q}_d$ , by replacing every vertex  $v \in V(\mathcal{Q}_d)$  with a cycle  $\mathcal{C}_{d-1}$  (*i.e.*, a cycle with  $d$  vertices), so that every vertex in the cycle is adjacent to a unique edge that was adjacent to  $v$ . If the vertices in one of the above-mentioned cycles (*i.e.*, the a cycle that corresponds to a vertex of  $\mathcal{Q}_d$ ) are indexed according to the order of their appearance in a walk along the cycle, then, if there is an edge joining a vertex in one cycle  $C$  to a vertex in another cycle  $C'$ ; the vertices have the same indices. Two examples,  $CCC_3$  and  $CCC_4$ , are shown respectively in Figures 4.13(c) and (d). These examples were chosen so that it may be seen, by comparing them to Figures 4.13(a) and (b), that they are derived from  $\mathcal{Q}_3$  and  $\mathcal{Q}_4$  respectively. Sýkora and Vrřo [SV93] proved the lower bound to  $\nu(CCC_d)$  in

$$\frac{1}{20}4^d - (9d + 1)2^{d-1} < \nu(CCC_d) < \frac{1}{20}4^d + 3d^2 2^{d-3} \tag{4.25}$$

by means of the graph-to-graph embedding method, whilst the upper bound in (4.25) was obtained from considering the maximum number of crossings that may be caused when replacing vertices in a drawing of  $\mathcal{Q}_d$  with cycles.

$d$	$\nu(Q_d)$	Proved by
1,2,3	= 0	This is easily seen from drawings
4	= 8	Harary, Hayes and Wu [HHW88]
5	$\leq$ 56	Harary, Hayes and Wu [HHW88]
$d \geq 6$	$\leq (165/1024)4^d - (2d^2 - 11d + 34)2^{d-3}$	Faria and Herrera de Figueiredo [FdF00]
$d$	$> (1/20)4^d - (d^2 + 1)2^{d-1}$	Sýkora and Vrto [SV93]
$d$	$\leq? (5/32)4^d - \lfloor (d^2 + 1)/2 \rfloor 2^{d-1}$	Conjectured by Eggleton and Guy [EG70]

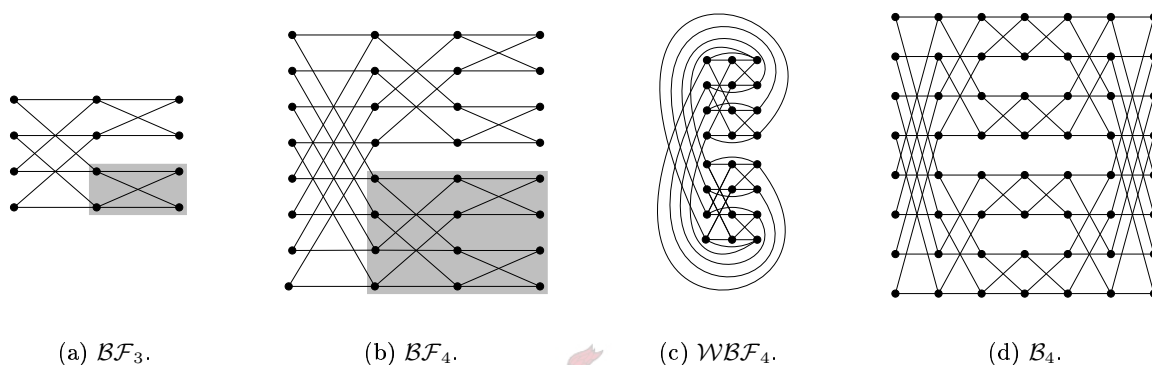
Table 4.3: Crossing number results for the hypercube  $Q_d$ 

Figure 4.14: Drawings of butterfly, wrapper butterfly and Beneš graphs.

**Butterfly graphs:** The butterfly graph is also referred to as the “FFT network,” because it was originally designed for the electronic implementation of Fast Fourier Transform algorithms. The butterfly graph of order  $n$  is denoted  $\mathcal{BF}_n$ . There is a recursive structure in that the graph  $\mathcal{BF}_{n+1}$  is built by attaching two  $\mathcal{BF}_n$  graphs by a “cross-switch.” For example, the  $\mathcal{BF}_3$  graph shown in Figure 4.14(a) is isomorphic to the subgraph highlighted by the gray block in Figure 4.14(b), which depicts the graph  $\mathcal{BF}_4$ . The “cross-switch” is constituted by the first two columns of vertices in Figure 4.14(b), and all the lines between these vertices. It may also be seen in Figure 4.14(a) that  $\mathcal{BF}_3$  is constructed by the attachment of two  $\mathcal{BF}_2$  graphs via a “cross-switch” — one of the  $\mathcal{BF}_2$  subgraphs is highlighted by the gray block. Cimikowski [Cim96] showed that

$$\nu(\mathcal{BF}_n) \leq \frac{3 \times 4^n}{2} - 3 \times 2^n - n2^n + 1.$$

**Wrapped Butterfly graphs:** With reference to the Figure 4.14(b), the wrapped butterfly graph of order  $n$  is constructed from the butterfly graph of order  $n$  by merging the first column (*i.e.*, the leftmost column) of vertices with the last column (*i.e.*, the rightmost) of vertices. The wrapped butterfly graph of order  $n$  is denoted  $\mathcal{WBF}_n$ , and an order 3 wrapped butterfly graph is shown in Figure 4.14(c). It was shown by Cimikowski [Cim96] that

$$\nu(\mathcal{WBF}_n) \leq \frac{3 \times 4^n}{2} - 3 \times 2^n - n2^n.$$

**Beneš graphs:** Like the wrapped butterfly graph, the Beneš graph may be defined in terms of the butterfly graph. The Beneš graph of order  $n$ , denoted  $\mathcal{B}_n$ , is formed by placing two copies of the butterfly graph of order  $n$ ,  $\mathcal{BF}_n$ , back to back. This is illustrated in Figure 4.14(d), where it may be seen that a horizontal mirror image of the butterfly graph (which falls in the right-hand

side of the figure) in Figure 4.14(b), is attached to another (unmirrored) version of  $\mathcal{BF}_4$  (which falls on the left-hand side). Cimikowski [Cim96] found the upper bound

$$\nu(\mathcal{B}_n) \leq 3 \times 4^n - 5 \times 2^n - 2n2^n + 2.$$

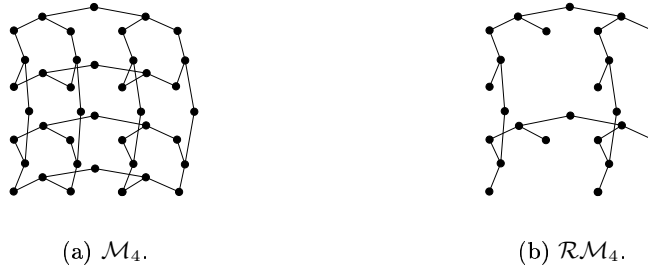


Figure 4.15: The mesh of trees, and the reduced mesh of trees.

**Mesh of trees, and the reduced mesh of trees:** The  $n \times n$  mesh of trees, denoted  $\mathcal{M}_n$ , has its vertices placed in a regular  $n \times n$  grid. In each row, there are  $n$  vertices, and these form the leaves of a complete, balanced binary tree. The same is true for each column. Therefore, there are a total of  $2n$  complete binary trees. The mesh of trees is only defined for powers of two (since otherwise not all of the binary trees would be balanced). An example of  $\mathcal{M}_4$  may be seen in Figure 4.15(a).

The reduced  $n \times n$  mesh of trees, denoted  $\mathcal{RM}_n$  is defined as the mesh of trees, except that only  $\log_2 n$  of the rows, and  $\log_2 n$  of the columns are the leaves of binary trees. Specifically, binary trees are added to the  $(i \log_2 n + 1)$ th rows and columns respectively, where  $0 \leq i < n / \log_2 n$ . The reduced  $4 \times 4$  mesh of trees is shown in Figure 4.15(b).

Cimikowski [Cim96] showed that

$$\nu(\mathcal{M}_n) \leq (n - 2)^2 \quad \text{and} \quad \nu(\mathcal{RM}_n) \leq \frac{(n - 2)^2}{\log_2 n}.$$

#### 4.2.3.6 Petersen graphs

The Petersen graph is famous for having served as the counter example to numerous propositions. This gives it a certain appeal for use in the crossing number problem, and some good results have been obtained for the general version of this graph.

**Definition 4.2.7** *The generalised Petersen graph  $\mathcal{P}(n, k) = (V, E)$  has the vertex and edge sets*

$$\begin{aligned} V &= \{x_i, y_i : i \in \mathbb{Z}_n\}, \\ E &= \{\{x_i, x_{i+1}\}, \{x_i, y_i\}, \{y_i, y_{i+k}\} : x_i, x_{i+1}, y_i, y_{i+k} \in V\}. \end{aligned} \quad \blacksquare$$

Fiorini [Fio86] originally provided bounds and exact results for  $\nu(\mathcal{P}(3n, 3))$  and  $\nu(\mathcal{P}(4n, k))$ . Unfortunately a flaw was found in his paper (according to Richter and Salazar [RS02]), which invalidated his results. The known results for the Petersen graph are shown in Table 4.4.

$n$	$k$	$\nu(\mathcal{P}(n, k))$	Proved by
$2n, 3$	2	= 0	Exoo, Harary and Kabell [EHK81]
5	2	= 2	Exoo, Harary and Kabell [EHK81]
$n$	2	= 3 if $n \geq 7$ and $n$ odd	Exoo, Harary and Kabell [EHK81]
10	3	$\geq$ 5 $\leq$ 6	McQuillan and Richter [MR92]
10	4	= 4	Lovrečič Saražin [LS97]
$3n$	3	= $n$	Richter and Salazar [RS02]
$3n + 1$	3	= $n + 3$	
$3n + 2$	3	= $n + 2$	
$3n$	$k$	= $k$	Fiorini and Gauci [FG02]
$n$	$k$	$\leq (2 - 2/k)n + k^2/2 + k/2 + 1$	Salazar [Sal04]
$n$	$k$	$\geq 2/5[(1 - 4/k)(n - k^4)] + 4k^2 + 1 - k^3$	Salazar [Sal04]

Table 4.4: Crossing number bounds for generalized Petersen graphs.

#### 4.2.3.7 Complements of cycles

The complement of a cycle  $\mathcal{C}_n$  is equivalent to the complete graph, with a cycle deleted. It is denoted  $\overline{\mathcal{C}}_n$ . Guy and Hill [GH73] proved the results

$$\binom{n}{5}(n-15)(n-17)/4 \binom{n-4}{3} \leq \nu(\overline{\mathcal{C}}_n) \begin{cases} = 0 & 3 \leq n \leq 6 \\ = \frac{1}{64}(n-3)^2(n-5)^2 & n = 7, 9 \\ = \frac{1}{64}n(n-4)(n-6)^2 & n = 8, 10 \\ \leq \frac{1}{64}(n-3)^2(n-5)^2 & n \text{ odd}, n > 9 \\ \leq \frac{1}{64}n(n-4)(n-6)^2 & n \text{ even}, n > 10 \end{cases}$$

using relatively *ad hoc* arguments for the exact cases, which gives little hope for a general method of finding general values. The upper bounds follow from drawings, and the lower bound is derived by using the standard counting method to find copies of  $\mathcal{K}_{5,n}$  in  $\overline{\mathcal{C}}_n$ .

### 4.3 Computational methods

Although some of the analytical results and techniques discussed in § 4.2 may be applied to find or approximate the crossing number of an arbitrary graph, the task becomes unmanageable for large graphs which do not share enough structure with the graphs for which analytical results are known. Furthermore, graphs that are highly asymmetrical may pose problems to the techniques in the foregoing section. This gives impetus to the case for using computer algorithms to find or approximate the crossing number of a graph.

As with most **NP**-complete graph parameters, algorithms for the crossing number problem come in two flavours: brute force exact algorithms, and heuristic approximation algorithms. The former is typically of theoretical importance, and for application to small graphs, whilst the latter is the only viable solution for all but the smallest graphs.

### 4.3.1 Brute force (exact) algorithms

Only two brute force algorithms have been devised to date. For both algorithms, finding the asymptotic running time has proven to be very difficult indeed — it is likely that this running time depends not only on the number of vertices and/or edges of the input graph, but on the structure of the graph itself.

#### 4.3.1.1 The Garey–Johnson algorithm

This algorithm derives its name from Garey and Johnson who demonstrated that determining the crossing number for a bipartite graph in the plane is an **NP**–complete problem [GJ83]. It follows that, since most graphs contain relatively large bipartite subgraphs, determining their plane crossing numbers is, in general, an **NP**–complete problem. Their algorithm is implicit in their article, and it is rather simple as is typically the case with brute force algorithms.

In order to test whether  $\nu(\mathcal{G}) \leq k$  for a graph  $\mathcal{G}$  and a number  $k \in \mathbb{N}$ , the algorithm enumerates every set  $P$  of  $k$  unordered *pairs* of edges in  $\mathcal{G}$ . An unordered pair of edges corresponds to a crossing between the two edges. When considering a set  $P$  of pairs of edges, the algorithm must also enumerate all possible permutations  $X_e$  of crossings with each edge  $e$ . For example, Figure 4.16 represents the case where in a particular choice of edge pairs, the pairs  $\{e_s, e_a\}$ ,  $\{e_s, e_b\}$  and  $\{e_s, e_c\}$  are present. Two permutations  $X_e^{(1)}$  and  $X_e^{(2)}$  of the order of  $e_a, e_b$  and  $e_c$  over  $e_s$  are shown, corresponding to the orderings  $(e_a, e_b, e_c)$  and  $(e_c, e_a, e_b)$  in Figures 4.16 (a) and (b) respectively.

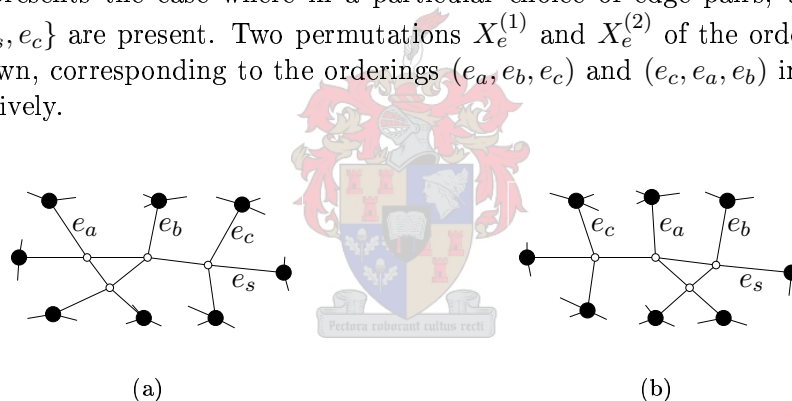


Figure 4.16: Two different crossing configurations where  $e_a, e_b, e_c$  cross  $e_s$ .

To determine whether a chosen set  $P$  of edge pairs, and a set of permutations  $\mathcal{X} = \{X_e : e \text{ is an edge involved in some crossings}\}$ , correspond to a graph drawing with at most  $k$  crossings, the crossings are modelled as vertices of degree four. If the resulting graph is planar, then a planar drawing of the resulting graph is a drawing of  $\mathcal{G}$  with at most  $k$  crossings. If this occurs, then the algorithm returns successfully, otherwise if no set  $P$  produces a planar configuration, the algorithm returns a failure, indicating that  $\nu(\mathcal{G}) > k$ .

There is a subtlety here: it is easy to assume that the Garey–Johnson algorithm could be sped up by having it avoid cases where adjacent edges cross — that is, by considering only drawings in single-cross normal form. Archdeacon and Richter [AR88] showed that  $\nu(\mathcal{K}_n) \equiv \nu(\mathcal{K}_{p,q}) \equiv 1 \pmod{2}$  where  $n, p, q$  are all odd, when all drawings of these graphs are considered to be in single-cross normal form. Therefore, the Garey–Johnson algorithm would incorrectly report, that the crossing number is, for example, larger than or equal to  $k$ , when  $k$  is even and  $n$  is odd, for the graph  $\mathcal{K}_q$ , when it is the case that  $\nu(\mathcal{K}_n) < k$ . This is because it would not be able to find a drawing in single-cross normal form with an even number of crossings. That said, there is no reason why the algorithm couldn't be made to avoid adjacent edge crossings, as long as these details are kept in mind.

Garey and Johnson provided no details on how to implement this algorithm, but it is a relatively simple exercise, and an implementation is given later in this thesis. Given a set of edge pairs  $P$ , where the edge permutations are given by  $X_e^{(1)}, X_e^{(2)}, \dots, X_e^{(\ell)}$ , the time required to enumerate all such permutations is  $O(X_e^{(1)} \times X_e^{(2)} \times \dots \times X_e^{(\ell)})$ . However, the total running time of the algorithm is difficult to estimate, since there seems to be no simple way of determining an expression that will represent the cumulative running time of all edge sets  $P$ . In the worst case, all  $k$  crossings occur on a single edge, resulting in a running time of  $O(k!)$  for enumeration of all configurations in a set  $P$ . Now,  $k$  pairs of edges are chosen for each set  $P$ , and there are therefore  $\binom{q}{k}$  such choices, where  $q = \binom{|E(\mathcal{G})|}{2}$ . The running time of the algorithm may then be expressed as  $O(k! \binom{q}{k})$  for the worst case. This does give a slightly distorted view of the real algorithmic performance, since the average running time for the enumeration of crossing configurations in a set  $P$  might be much smaller than  $k!$ , although it will almost certainly still be of an exponential order.

#### 4.3.1.2 The Harris–Harris algorithm

This algorithm is named after F.C. Harris Jr. and C.R. Harris, for their proposal [HH99] of a brute force method for finding the crossing number of a graph. Their idea was to choose a clockwise ordering  $\Pi(\mathcal{G}) = (\pi_1, \pi_2, \dots, \pi_{|V(\mathcal{G})|})$  of the edges around the vertices of a graph  $\mathcal{G}$ , where  $\pi_i = (e_a, e_b, \dots, e_t)$  corresponds to the ordering of the edges incident to  $v_i \in V(\mathcal{G})$ .

Due to a theorem by Heffter [Hef91], there exists an orientable surface  $S$  (see § 4.1.4), so that  $\mathcal{G}$  has an embedding in  $S$  for which the clockwise ordering of the edges of  $\mathcal{G}$  around its vertices in  $S$  is given by the orderings in  $\Pi$ . The embeddings of subgraphs of  $\mathcal{G}$  are of course also determined by  $\Pi$  (or technically by subsets of  $\Pi$ , or subsets of the constituent sets  $\pi_i \in \Pi$ ), and there certainly exists a spanning subgraph  $\mathcal{H}$  of  $\mathcal{G}$ , so that  $\mathcal{H}$  may be embedded in the plane whilst satisfying  $\Pi$  (one may say that  $\mathcal{H}$  is a planar spanning subgraph of  $\mathcal{G}$  with respect to  $\Pi$ ).

Suppose that, in a drawing  $\phi$  of  $\mathcal{G}$  which realises  $\nu(\mathcal{G})$ , the orderings of the edges of  $\mathcal{G}$  around its vertices is given by  $\Pi'$ . Then, without any knowledge of  $\phi$ , one can draw a planar spanning subgraph  $\mathcal{H}$  of  $\mathcal{G}$  with respect to  $\Pi'$ , so that it matches the drawing of  $\mathcal{H}$  in  $\phi$ . Unfortunately, there is no easy way to insert the remaining edges  $C = E(\mathcal{G}) \setminus E(\mathcal{H})$  into  $\mathcal{H}$  to arrive at  $\phi$  — in order to achieve this, every possible way of inserting an edge  $e \in C$  must be attempted. However, one at least knows the order of  $e$  on its incident vertices, which constrains the way in which  $e$  may be drawn.

Of course, it is in general also not possible to determine  $\Pi'$ , since this would require a drawing realising the crossing number of  $\mathcal{G}$ , which is not known. This means that the algorithm must painstakingly enumerate all possible orderings  $\Pi'$ .

#### Description of the algorithm

The algorithm is divided into two parts. For each set of edge orderings  $\Pi$  of a graph  $\mathcal{G}$ , the algorithm

1. draws  $\mathcal{G}$  as far as possible according to  $\Pi$ , without causing any crossings; then it
2. attempts to insert all remaining edges into the drawing of  $\mathcal{G}$  by attempting each possible way of inserting each edge.

It remains to be described how the algorithm accomplishes each step. The solution to the first part is to use the theory of *rotational embedding schemes*, which provides a way of constructing a drawing of a graph in an orientable surface from the ordering  $\Pi$  of its edges around its vertices. The solution to the second part is a branch and bound search.

### Rotational embedding schemes

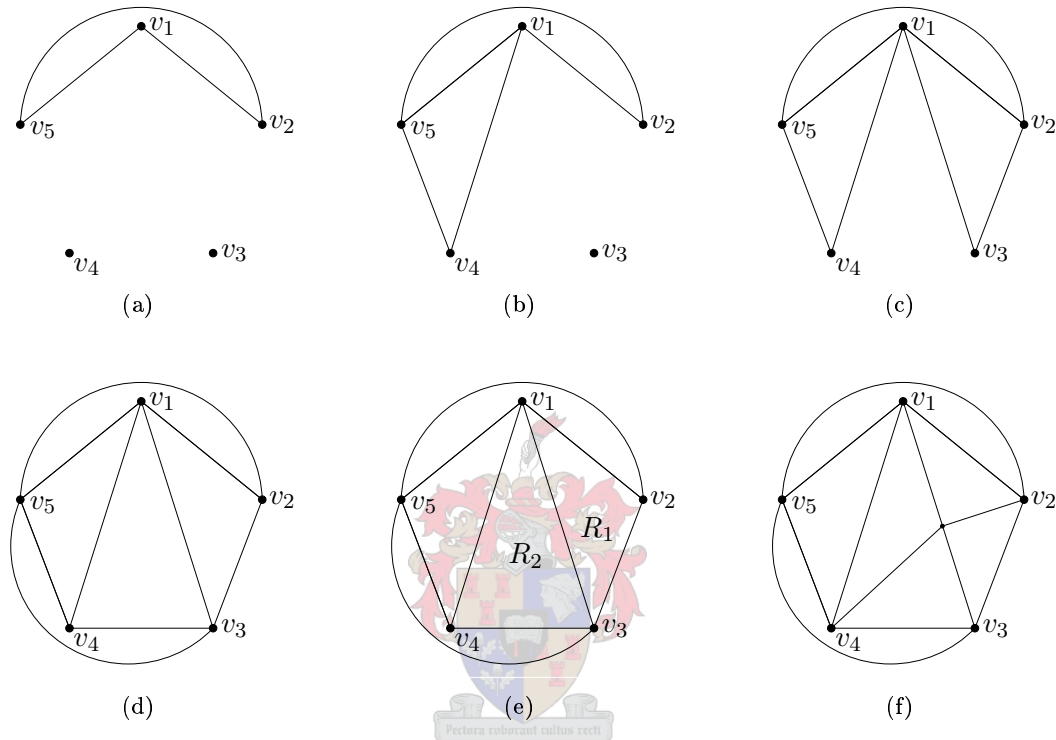


Figure 4.17: A step by step example of how the Harris–Harris algorithm proceeds with  $\mathcal{K}_5$ .

Heffter [Hef91] introduced the notion of a *rotational embedding scheme* for the algebraic description of graph embeddings in orientable surfaces. The concept was extended to non-orientable surfaces by Stahl [Sta78]. White and Beineke’s description of this notion will be adopted here [WB78].

For a graph  $\mathcal{G}$ , and a vertex  $v_i \in V(\mathcal{G})$ , let  $E(v_i)$  denote (as before) the edges adjacent to  $v_i$ . Consider the edges in  $E(v_i)$  as being directed away from  $v_i$ . Let  $\pi_i$  be a cyclic permutation of the edges in  $E(v_i)$ . Hence, every edge  $e = \{v_i, v_j\}$  results in the two arcs  $(v_i, v_j) \in E(v_i)$  and  $(v_j, v_i) \in E(v_j)$ . Also, define  $\pi_i$  as an operator, so that  $\pi_i(v_i, v_j) = (v_i, v_k)$ , where  $(v_i, v_j)$  is followed by  $(v_i, v_k)$  in  $\pi_i$ .

An embedding of a graph into a surface is said to be 2-cell if it has the property that every region may be deformed to a disc. For example, in the embedding of  $\mathcal{K}_4$  into the torus in Figure 4.18(a), all three regions,  $R_1$ ,  $R_2$  and  $R_3$  may be deformed to discs; this embedding is therefore a 2-cell embedding. However, in the embedding of  $\mathcal{K}_4$  in Figure 4.18(b), the region labelled  $R_4$  cannot be deformed to a disc; consequently, this embedding is not a 2-cell embedding. Embeddings of connected<sup>7</sup> graphs into the sphere are always 2-cell. This is also the only surface to which the Harris–Harris algorithm is applied.

<sup>7</sup>If a graph is disconnected, then the region into which all of the components are drawn will not be deformable to a disc.



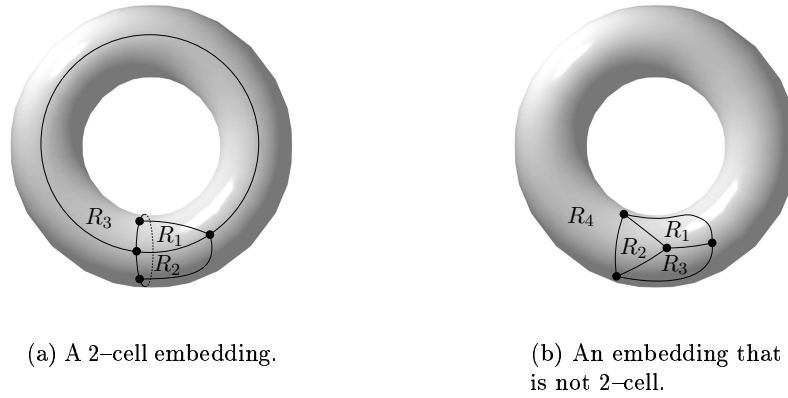


Figure 4.18: An illustration of two embeddings of  $\mathcal{K}_4$  into the torus. The first embedding is 2-cell, whilst this is not true of the second embedding, since the region  $R_4$  cannot be deformed to a disc.

The following theorem shows that, for an arbitrary graph  $\mathcal{G}$  and for any ordering  $\Pi$  of the edges of  $\mathcal{G}$  around its vertices, an embedding of  $\mathcal{G}$  may be found on some orientable surface, such that the orderings of the edges around the vertices of  $\mathcal{G}$  correspond to  $\Pi$ . A proof of the theorem, by White and Beineke, may be found in [WB78].

**Theorem 4.3.1** *Each choice of permutations  $\Pi(\mathcal{G}) = (\pi_1, \pi_2, \dots, \pi_{|V(\mathcal{G})|})$  uniquely determines a 2-cell embedding of  $\mathcal{G}$  in an orientable surface  $\mathcal{S}$ , which itself is specified by the embedding;  $\mathcal{S}$  may be so oriented that, at every vertex  $v_i$ , the arc  $(v_i, v_j)$  is followed by  $\pi_i(v_i, v_j) = (v_i, v_k)$ . Conversely, for any 2-cell embedding of  $\mathcal{G}$  in an orientable surface  $\mathcal{S}'$ , there exists a unique corresponding set of permutations  $(\pi_1, \pi_2, \dots, \pi_{|V(\mathcal{G})|})$  which yields the embedding. ■*

As a running example for the description of the Harris–Harris algorithm, an embedding of  $\mathcal{K}_5$  will be considered, where the edge orderings  $\Pi$  correspond to a drawing of  $\mathcal{K}_5$  realising its crossing number. The vertices are numbered  $v_1$  through  $v_5$ . The edge orderings  $\Pi = (\pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$  are given as:

$$\begin{aligned}
 \pi_1 & : (v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5) \sim (v_2, v_3, v_4, v_5) \\
 \pi_2 & : (v_2, v_1), (v_2, v_5), (v_2, v_3), (v_2, v_4) \sim (v_1, v_5, v_3, v_4) \\
 \pi_3 & : (v_3, v_1), (v_3, v_2), (v_3, v_5), (v_3, v_4) \sim (v_1, v_2, v_5, v_4) \\
 \pi_4 & : (v_4, v_1), (v_4, v_2), (v_4, v_3), (v_4, v_5) \sim (v_1, v_2, v_3, v_5) \\
 \pi_5 & : (v_5, v_1), (v_5, v_4), (v_5, v_3), (v_5, v_2) \sim (v_1, v_4, v_3, v_2)
 \end{aligned}$$

For a finite set  $S$ , let  $f$  be a bijection  $f : S \rightarrow S$ . Now for an element  $x \in \mathcal{B}$ , a sequence  $x, f(x), f(f(x)), \dots$  of successive applications of  $f$  to its previous applications must eventually arrive at an answer equal to  $x$ , due to the fact that  $f$  is a bijection. Such a sequence is closed under the application of  $f$ , and is called an *orbit* of  $f$  in  $S$ . A bijection  $f$  is said to *generate* its orbits, and the orbits of  $f$  partition  $S$ .

Define the mapping on directed edges  $\Pi^*(v_i, v_j) = \pi_j(v_j, v_i)$ . The orbits of  $\Pi^*(v_i, v_j) = \pi_j(v_j, v_i)$  define the different regions. Using this result, the regions for  $\mathcal{K}_5$  may be filled in one by one.

The region shown in Figure 4.17(a) may be mapped out in the following way. Starting with  $(v_1, v_2)$ , it is seen to be followed by  $\Pi^*(v_1, v_2) = (v_2, v_5)$ , which in turn is followed by  $\Pi^*(v_2, v_5) =$

$(v_5, v_1)$ . Since  $\Pi^*(v_5, v_1) = (v_1, v_2)$ , it may be seen that the orbit of  $\Pi^*(v_1, v_2)$  corresponds to the mapped out region.

In the same fashion, the regions in Figures 4.17(b)–(d) are mapped out respectively by the orbits of  $\Pi^*(v_1, v_5)$ ,  $\Pi^*(v_2, v_1)$  and  $\Pi^*(v_4, v_5)$ .

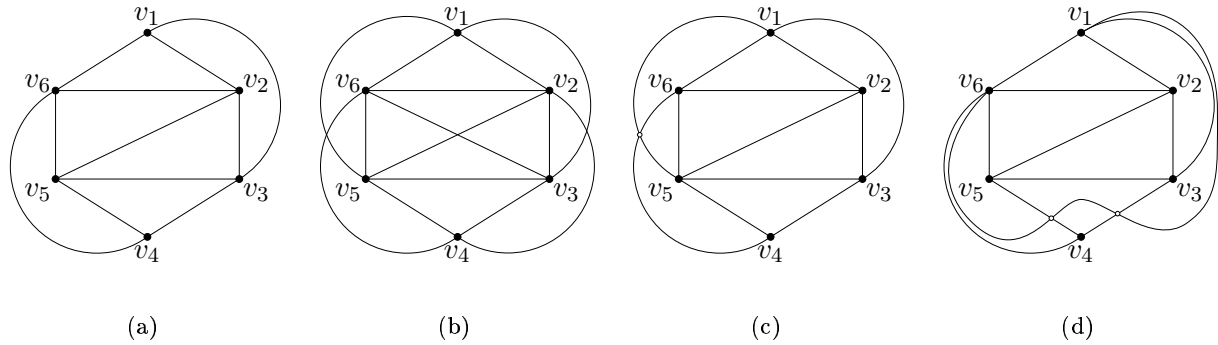


Figure 4.19: In general, there are multiple ways to insert crossing edges.

### Insertion of edges

All edges which have to be inserted into  $\mathcal{G}$  after the first part will cause crossings in  $\mathcal{G}$ , and for the purposes of this section, they are called crossing edges of  $\Pi$ .

From Figure 4.17(e), it may be seen that one cannot draw  $\mathcal{K}_5$  any further without incurring a crossed edge. According to  $\Pi$ , the crossing edge  $\{v_2, v_4\}$  must be drawn so that it has one end in the region marked  $R_1$  and its other end in the region marked  $R_2$ . In this case  $\{v_2, v_4\}$  must cross  $\{v_1, v_3\}$  in order to leave  $R_1$ , since it is adjacent to both  $\{v_1, v_2\}$  and  $\{v_2, v_3\}$ , and drawings are required to be in single cross normal form. Also, it cannot leave  $R_2$  other than by crossing  $\{v_1, v_3\}$ , for the same reason. In this case, the way that  $\{v_2, v_4\}$  should be drawn is completely specified, and Figure 4.17(f) shows the action of inserting the edge.

In general, however, the way in which the crossing edges have to be inserted is not specified to such an extent as for the previous example. Consider a partial drawing of  $\mathcal{K}_6$  shown in Figure 4.19(a), which is a drawing of a largest planar subgraph of  $\mathcal{K}_6$  with respect to the ordering  $\Pi(\mathcal{K}_6)$  given by the full drawing of  $\mathcal{K}_6$  in Figure 4.19(b). In this partial drawing, the edge  $\{v_1, v_5\}$  may be inserted in a number of ways, two of which are shown in Figures 4.19(c)–(d). Now, for each of the possible ways that  $\{v_1, v_5\}$  may be inserted, the total number of ways in which  $\{v_1, v_3\}$  and subsequently  $\{v_3, v_6\}$  may be inserted must also be enumerated. At each insertion of a crossing edge, the total number of regions increases, and therefore the remaining edges have an increasing number of placement options.

The insertion of crossing edges is easily modelled as a branch and bound algorithm, since the branches of the search tree correspond to the different ways in which crossing edges may be inserted. The bounding is achieved by storing the minimum number of crossings found so far, and terminating a search on a branch of the search tree when it becomes apparent that the current configuration will result in more crossings than the best configuration found so far.

### General remarks

Although the first part of the algorithm, which creates planar drawings corresponding to a given ordering  $\Pi$ , reduces the total amount of work that the algorithm must do, by and large, most

work is done in the branch and bound phase with the insertion of crossing edges. It should not be underestimated that this leads to a combinatorial explosion of the worst case complexity, even for small graphs like  $\mathcal{K}_{10}$ .

Furthermore, the procedure has to be repeated for every possible ordering  $\Pi$ . The total number of these permutations is of the order  $O(|V(\mathcal{G})||V(\mathcal{G})!)$  for complete and bipartite graphs, since the degree of each vertex in such a graph is of order  $O(|V(\mathcal{G})|)$ , meaning that there are  $O(|V(\mathcal{G})!)$  orderings of edges around each vertex. Doubtless, many of these orderings will be equivalent under symmetry considerations, but this is hardly an easy observation to exploit in general.

Thus, the Harris–Harris algorithm has the problem of worst case complexity explosions at two levels, and it seems, at least at a superficial level, that the Garey–Johnson algorithm does less work. The Garey–Johnson algorithm also lends itself to some easily exploitable symmetry considerations as will be demonstrated later (although there are also many cases for the Garey–Johnson algorithm where it would take more work to discover symmetries than just to let the algorithm run its course).

#### 4.3.1.3 The Pach & Toth odd crossing number algorithm

Pach and Tóth [PT98] developed an algorithm for determining the odd crossing number of a graph as part of their proof that the odd crossing number problem is **NP**–complete.

The algorithm rests heavily on Tutte’s crossing algebra discussed in § 4.2.1. Pach and Tóth considered two types of transformations on graph drawings. The first transformation is Tutte’s edge “pulling” transformation, but for the second transformation, they noted that since Tutte’s transformations do not account for any crossings between adjacent edges, a transformation was needed to facilitate this.

They defined a transformation which only alters drawings of adjacent edges where the edges are in regions that are very close to the vertices of a graph. In these regions, the order of the edges as they leave the vertex may be permuted to force crossings between pairs of edges incident to the vertex. This is illustrated in Figure 4.20, and is described in more detail later in the section.

As with Tutte, Pach and Tóth took an algebraic view of crossing configurations and the transformations thereupon. Firstly, they adopted a seemingly superficial shift from Tutte’s approach, by viewing his crossing chains as vectors (this is achieved in the same way that a polynomial would be represented in vector form). More importantly, since only the parity of the number of crossings between a pair of edges is of relevance for the odd crossing number problem, they considered all operations between vectors corresponding to crossing configurations and transformations to occur modulo 2. This converted Tutte’s crossing chain groups, and the groups resulting from Pach and Tóth’s own transformation, into subspaces of the vector space  $\mathcal{V}(q, 2)$ , where  $q$  denotes the total number of pairs of edges.

The algorithm commences with a vector  $\bar{x}$  representing the odd crossing configuration of a given drawing. Transformations are applied by the addition of vectors (corresponding to crossing chains) to  $\bar{x}$ , modulo 2. The resulting odd crossing configuration completely specifies a drawing, and therefore, the algorithm only considers vector operations, and explicit drawings do not have to be taken into consideration.

Of course, it still remains to show how the algorithm enumerates all possible crossing configurations. Pach and Tóth’s important result was to show that the odd crossing configuration of any drawing of a graph in normal form is expressible as the sum of any other crossing configuration and the appropriate transformations on that configuration (or in the terminology of vector

spaces, any odd crossing configuration of a graph is in the span of all transformations applied to any other crossing configuration).

### Crossing transformations

As has been stated, the approach of Pach and Tóth is very close to that of Tutte [Tut70], except that they viewed crossing chains (§ 4.2.1.1) as vectors, where the vector components correspond to the  $\lambda(ij, k\ell)$  coefficients in the crossing chains (this is virtually the same method by which a polynomial is represented as a vector). Therefore, for a graph  $\mathcal{G}$ , and a drawing  $\phi$  of  $\mathcal{G}$ , the crossing chain

$$x(\phi) = \sum_{Q(\mathcal{G})} \lambda(ij, k\ell)[ij, k\ell] \tag{4.26}$$

corresponds to a vector

$$\bar{x}(\phi) = [\lambda_1 \ \lambda_2 \ \dots \ \lambda_q],$$

where  $q = \binom{|E(\mathcal{G})|}{2}$ , and where the terms  $\lambda_i$ ,  $1 \leq i \leq q$  are the coefficients of the terms in (4.26). By the same method, a crossing coboundary  $c(ij, k)$  (§ 4.2.1.2) may be written as a vector  $\bar{c}(ij, k)$ . Clearly, using this notation, the crossing chain  $x(\phi) + c(ij, k)$  may be interpreted as the vector  $\bar{x}(\phi) + \bar{c}(ij, k)$  under vector addition.

For the odd crossing number problem, it is only relevant whether a pair of edges cross an even or odd number of times. Therefore Pach and Tóth take the vectors  $\bar{x}(\phi)$  and  $\bar{c}(ij, k)$  modulo 2, to form, respectively, the vectors  $\bar{X}_\phi$  and  $\bar{Y}_{v_k, \{v_i, v_j\}}$  (using their notation). Hence

$$\bar{X}_\phi = \bar{x}(\phi) \pmod{2} \text{ and} \tag{4.27}$$

$$\bar{Y}_{v_k, \{v_i, v_j\}} = \bar{c}(ij, k) \pmod{2}. \tag{4.28}$$

Denote the set of all crossing coboundary vectors  $\bar{Y}_{v,e}$  as  $\bar{\mathcal{Y}}_{\mathcal{G}}$ , that is

$$\bar{\mathcal{Y}}_{\mathcal{G}} = \{\bar{Y}_{v,e} : v \in V(\mathcal{G}), e \in E(\mathcal{G}) \text{ and } v \text{ is not adjacent to } e\}.$$

The set of vectors  $\bar{\mathcal{Y}}_{\mathcal{G}}$  is a vector space under addition and scalar multiplication modulo 2. This vector space is a subspace of the vector space of vectors with length  $\binom{|E(\mathcal{G})|}{2}$ , and vector operations modulo 2, and it describes all possible transformations that are due to the action of “pulling” edges over vertices. It cannot, however, change the number of crossings that occur between two adjacent edges.

### Crossings between adjacent edges

To change the number of crossings between a pair of adjacent edges, Pach and Tóth define a transformation, which may best be described as “twisting” the edges around each other in the vicinity of their shared vertex. In a drawing  $\phi$  of a graph  $\mathcal{G}$ , for a given vertex  $v \in V(\mathcal{G})$ , the edges leaving  $v$  induce a natural clockwise ordering — for example, in Figure 4.20 (a), the ordering is  $e_1, e_2, e_3, e_4, e_5$  and  $e_6$  (or any cyclic shift of this ordering). Should one wish to change the crossing situation between two edges, this change should not affect the way in which edges that are not incident to  $v$ , are crossed. For this reason, any changes should occur within a distance  $\varepsilon$  from  $v$ , leaving the edges unchanged beyond the distance  $\varepsilon$ , where  $\varepsilon$  is a small real, positive constant — this ensures that the original clockwise ordering of the edges remain intact for distances greater than  $\varepsilon$ .

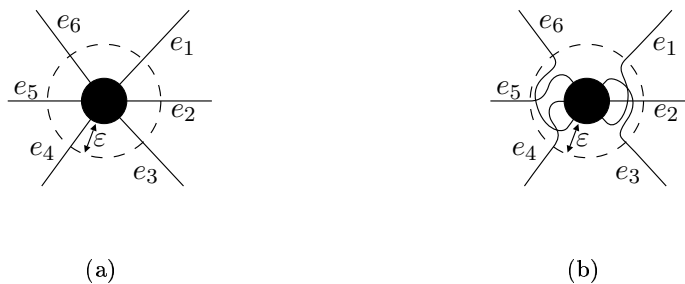


Figure 4.20: “Twisting” a pair of adjacent edges, causes a crossing between the pair.

The only way to change the number of crossings between two edges that are adjacent to  $v$ , is to change their ordering, relative to one another, as they leave  $v$ . For example, the original relative order had  $e_1$  followed by  $e_2$ , but in Figure 4.20(b),  $e_2$  is followed by  $e_1$ , which forces a crossing between the two edges. In the same way, crossings are caused for the pairs  $e_1$  &  $e_3$ ,  $e_2$  &  $e_3$ ,  $e_4$  &  $e_6$  and for  $e_5$  &  $e_6$ . Since the relative order of  $e_4$  and  $e_5$  did not change, there is no crossing between these edges.

For a vertex  $v_i \in V(\mathcal{G})$ , with degree  $d$ , the edges incident to  $v_i$  are labelled as  $\{e_1^i, e_2^i, \dots, e_d^i\}$ , where the labels are assigned in increasing clockwise order of the edges around  $v_i$  (i.e., the first edge is  $e_1^i$ , the second is  $e_2^i$ , and so on until the last, which is labelled  $e_d^i$ ). Now, for a particular permutation  $\sigma_i$  of the labels of the edges around  $v_i$ , it is found, if the labels of the two edges, labelled  $e_a^i$  and  $e_b^i$ ,  $a < b$  are swapped relative to one another by  $\sigma_i$ , that  $\sigma_i(a) > \sigma_i(b)$ , as otherwise  $\sigma_i(a) < \sigma_i(b)$ . Therefore it can easily be determined whether a pair of adjacent edges  $e, f$  with common vertex  $v_i$ , cross each other an odd number of times. The indicator function  $z(e, f, i)$  is defined as 1 when  $e$  crosses  $f$  an odd number of times due to  $\sigma_i$ , and 0 otherwise. That is

$$z(e, f, i) = \begin{cases} 1 & \text{if } e = e_\alpha^i, f = e_\beta^i \text{ and } (\alpha - \beta)(\sigma_i(\alpha) - \sigma_i(\beta)) < 0 \\ 0 & \text{otherwise.} \end{cases}$$

The total change that a permutation  $\sigma_i$  has on the odd crossing configuration is expressed by the vector  $\bar{Z}_{v_i, \sigma_i}$  (Pach and Tóth’s notation), where the components are the indicator functions  $z(e, f, i)$  for all  $e, f \in E(\mathcal{G})$ ,  $e \neq f$ . Using standard vector notation, one may write

$$\bar{Z}_{v_i, \sigma_i} = (z(e, f))_{e, f \in E(\mathcal{G}), e \neq f}.$$

Denote the set of all “edge twist” transformations  $\bar{Z}_{v_i, \sigma_i}$  as  $\bar{Z}_{\mathcal{G}}$ , that is

$$\bar{Z}_{\mathcal{G}} = \{\bar{Z}_{v_i, \sigma_i} : v_i \in V(\mathcal{G}) \text{ and any permutation } \sigma_i \text{ of edges incident to } v_i\}.$$

### Relating the odd crossing configurations of different drawings

From the preceding discussions, the vector space which describes all distinct odd crossing configurations obtainable from the given drawing  $\phi$ , is the space

$$\Psi = \bar{X}_\phi + \bar{Y}_{\mathcal{G}} + \bar{Z}_{\mathcal{G}}.$$

In order for Pach and Tóth’s algorithm to compute the odd crossing number for a graph  $\mathcal{G}$ , it must be shown, for any drawing  $\phi'$  of  $\mathcal{G}$  distinct from  $\phi$ , that  $\bar{X}_{\phi'} \in \Psi$ . This is true, because the algorithm will commence with the drawing  $\phi$ , and it must be able to generate the odd crossing configuration of any other drawing, if it is to find the odd crossing number of  $\mathcal{G}$ .

Pach and Tóth showed that the odd crossing configuration of any drawing of  $\mathcal{G}$  may be related to the odd crossing configuration of a so-called convex drawing of  $\mathcal{G}$ . A convex drawing in this context is a drawing of a graph where all of its vertices are placed equi-spaced on a unit circle, and all edges are drawn as straight lines (except for where three or more edges cross at a single point, which is corrected by the procedure described in § 3.1.2). An example of a convex drawing of  $\mathcal{K}_5$  is shown in Figure 4.21(f).

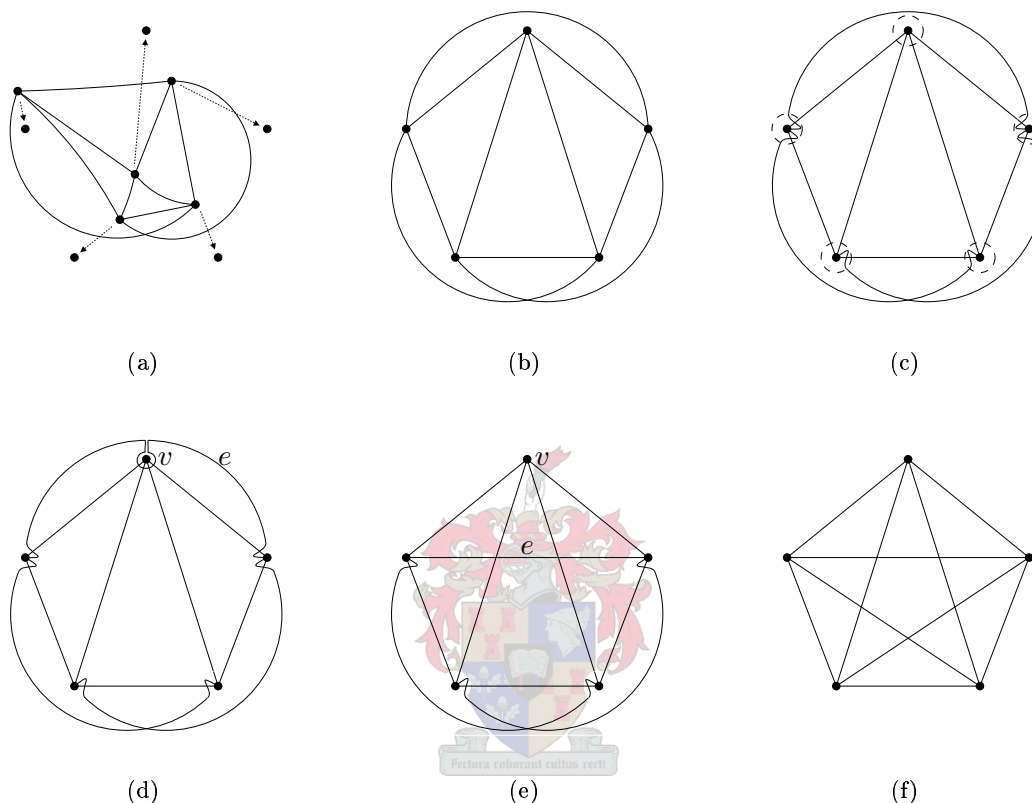


Figure 4.21: Transformations of the Pach–Tóth algorithm to render a graph drawing convex.

The following steps are followed to transform the odd crossing configuration of an arbitrary drawing of a graph into the odd crossing configuration of a convex drawing of that graph. The graph  $\mathcal{K}_5$  is used as an example:

1. Given a drawing  $\phi$  of  $\mathcal{K}_5$ , a topological transformation may be applied to  $\phi$  which will place the vertices of  $\mathcal{K}_5$  equi-spaced on the unit circle, without affecting the way that edges cross each other in  $\phi$ . This transformation is shown for a drawing of  $\mathcal{K}_5$  in Figure 4.21 (a), which is transformed to the drawing  $\phi_2$  in Figure 4.21(b).
2. The next step towards transforming  $\phi_2$  into a convex drawing, is to change the order of the edges around the vertices, so as to reflect the order of the edges in the convex drawing  $\phi_C$  shown in Figure 4.21(f). This may be achieved by a suitable number of edge twist transformations from the space  $\bar{\mathcal{Z}}_{\mathcal{K}_5}$ , from which the drawing  $\phi_3$ , shown in Figure 4.21(c), results.
3. Some sections of edges in the graph may run along the outside of the unit circle. As may be seen from Figure 4.21(d), the edge  $e$  runs outside the unit circle, and passes only the vertex  $v$ . A Tutte transformation, is applied to  $e$  and  $v$ , as is also shown in Figure 4.21(d).

It may be verified that the odd crossing configuration after the application of a Tutte transformation from the space  $\bar{\mathcal{Y}}_{\mathcal{K}_5}$  is the same as the odd crossing configuration when “pulling” the entire edge within the unit circle, as shown in Figure 4.21(e). This is applied to every edge — a number of Tutte transformations may be required for a single edge, depending upon how many vertices it passes as it runs outside the unit circle.

### General remarks

Since a crossing configuration is represented as a binary vector  $V$  where each component in  $V$  corresponds to the parity of the number of crossings between a pair of edges, there are no more than  $O(2^{\binom{|E(\mathcal{G})|}{2}})$  distinct configurations in  $V$  crossing configurations. Given an existing odd crossing number configuration, based on transformations  $Y$  from  $\bar{\mathcal{Y}}_{\mathcal{G}}$  and  $Z$  from  $\bar{\mathcal{Z}}_{\mathcal{G}}$ , a certain amount of computation time is required to generate a new transformation. The time required for both cases is considered.

1. The transformations in  $\bar{\mathcal{Y}}_{\mathcal{G}}$  correspond to general cross-coboundaries, *i.e.*, transformations based on the addition of a number of initial cross-coboundaries. The initial cross-coboundaries correspond to the way in which each edge that is not adjacent to a vertex is drawn around that vertex, so as to cross its adjacent edges in a certain manner. Therefore, inspection of an initial cross-coboundary for a vertex  $v$  and edge  $e$  takes  $O(\deg_{\mathcal{G}} v)$  time. A class of dense graphs (for example the complete graphs or the complete bipartite graphs) has the property that  $\sum_{v \in V(\mathcal{G})} \deg_{\mathcal{G}} v = \Omega(|V(\mathcal{G})|^2)$ , and the number of edges to which a vertex is not adjacent is  $O(|E(\mathcal{G})|)$ . Therefore, consideration of the cross-coboundaries for all vertices takes  $\Omega(|E(\mathcal{G})||V(\mathcal{G})|^2)$  time in the worst case. Updating of the cross-coboundaries of all vertices to a new cross-coboundaries can be achieved in  $O(|E(\mathcal{G})||V(\mathcal{G})|^2)$  time.
2. The transformations in  $\bar{\mathcal{Z}}_{\mathcal{G}}$  correspond to “edge twists” of the edges incident to a vertex. Computing a new permutation from an existing permutation may be performed in an amount of time that is linear to the number of elements. Therefore, this process takes  $O(\deg_{\mathcal{G}} v)$  time for a vertex  $v$ . In a class of dense graphs,  $\sum_{v \in V(\mathcal{G})} \deg_{\mathcal{G}} v = \Omega(|V(\mathcal{G})|^2)$  and therefore this step takes  $\Omega(|V(\mathcal{G})|^2)$  time at worst. The time required for this step is also bounded from above, so that it may run in  $O(|V(\mathcal{G})|^2)$  time.

The possibility exists that both of these transformations are applied at the same time, but the running time of the former transformation generally dominates the running time of the latter. Thus, each transformation may be performed in  $O(|E(\mathcal{G})||V(\mathcal{G})|^2)$  time, which results in a worst case running time of  $O(2^{\binom{|E(\mathcal{G})|}{2}}|E(\mathcal{G})||V(\mathcal{G})|^2)$  for the entire algorithm.

#### 4.3.1.4 Székely’s independent-odd crossing number algorithm

Székely’s algorithm [Szé04] computes the independent odd crossing number of a graph, combining some ideas from Pach and Tóth’s algorithm, and from two-page book layouts. Like Pach and Tóth’s algorithm, it maintains a vector of binary values, which corresponds to whether a pair of non-adjacent edges cross each other an even or odd number of times, and like in two-page book layouts, it obtains crossing information by determining whether a pair of edges are alternating or not.

Székely considered drawings of graphs in which vertices are placed equi-spaced on the unit circle (*i.e.*, he considered circular drawings, except that edges are allowed to cross the imaginary circle

in his drawings). As may be seen from Pach and Tóth’s algorithm (§ 4.3.1.3), any drawing  $\phi$  of a graph  $\mathcal{G}$  may be transformed to a drawing  $\phi'$  of  $\mathcal{G}$  in which the vertices of  $\mathcal{G}$  are drawn equi-spaced on the unit circle, without changing the way that edges of  $\mathcal{G}$  cross each other (in other words, the two drawings  $\phi$  and  $\phi'$  are topologically equivalent).

Using this model, Székely showed that the parity of the number of crossings between a pair of non-adjacent edges  $e$  and  $f$  depends only on two binary combinatorial properties. The first property is whether the vertices of  $e$  and  $f$  alternate each other or not. The second property relates to the way that edges are drawn. A pair of vertices may be “separated” by an edge, or not.

### Vertex separation

For the purpose of understanding the notion of vertex separation, rays are drawn from each vertex, outwards from, and perpendicular to the unit circle, stretching out into infinity. This is shown in Figure 4.22(a). For a vertex  $v_i$ , denote the ray emanating from it by  $r_{v_i}$ . These rays are not part of the graph drawing — they are merely an explanatory tool.

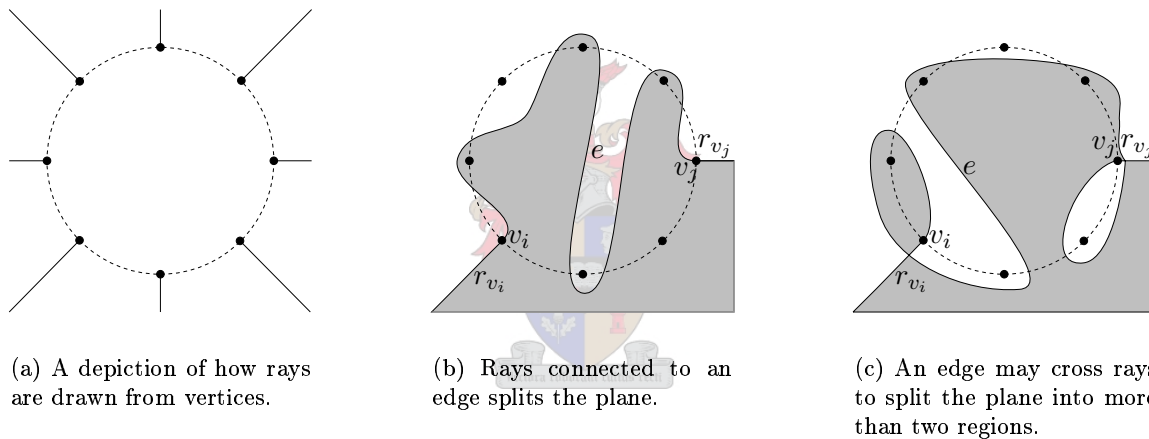


Figure 4.22: Vertex separation described by means of imaginary rays emanating from the vertices.

When an edge  $e = \{v_i, v_j\}$  is drawn, the line  $r_{v_i} \cup \phi(e) \cup r_{v_j}$ , denoted  $p_\phi(e)$ , partitions the plane into at least two regions. This is shown in Figure 4.22(b). The edge  $e$  is allowed to cross  $r_{v_i}$  and  $r_{v_j}$  multiple times, and in this case the plane is partitioned into three or more regions, as shown in Figure 4.22(c). The line  $p_\phi(e)$  always induces a two-colourable map of the plane. This is true, since where  $p_\phi(e)$  crosses itself, the “loops” that form, fall entirely within either a gray or white region. Since they are only adjacent to that region (from the perspective of a map), they may be coloured using the opposite colour to that with which the region is coloured. This may be seen in Figure 4.22(c).

The bicolouring of the plane by  $p_\phi(e)$  forms a bipartition of the vertices  $V(\mathcal{G}) \setminus \{v_i, v_j\}$ , since vertices cannot intersect any edges nor any rays (and they must each therefore fall wholly within a gray or white region). A drawing of  $e$  is said to *separate* a pair of vertices, if one vertex is in a gray region, and the other within a white region.

Let  $f$  be another edge of  $\mathcal{G}$ , with end vertices  $v_k$  and  $v_\ell$ . Suppose that  $e$  separates  $v_k$  and  $v_\ell$ , where  $v_k$  is in a gray region, and  $v_\ell$  in a white region. A Tutte transformation, whereby  $e$  is “pulled” across, say  $v_k$ , will cause  $v_k$  and  $v_\ell$  not to be separated anymore, since  $v_k$  will now be in a white region. The transformation also changes the number of crossings between  $e$  and  $f$  by



exactly one, and hence the parity, since it crosses both  $f$  and  $r_{v_k}$  once due to the transformation, but only the former has any impact (recall that the ray is not really present). Now,  $v_k$  and  $v_\ell$  may again be separated by another Tutte transformation. From this, it may be concluded that the separation that the edges  $e$  and  $f$  induce on each other's incident vertices, determines the parity of the number of crossings between the pair of edges.

### Calculating the parity of the number of crossings between two edges

If the vertex separation is kept constant, then the only other variable which has an effect on the parity of the number of crossings between a pair of edges is whether or not the incident vertices of the edges are alternating. For a pair of edges  $e$  and  $f$ , this makes for a total of 6 different configurations based on vertex alternation and vertex separation, that determines the parity of the number of crossings between them.

The configurations are easy to describe: for each of the two possible ways in which the vertices of  $e$  and  $f$  can be made to either alternate each other or not, it can happen that either

1.  $e$  and  $f$  both separate each other's vertices,
2.  $e$  separates the vertices of  $f$ , but  $f$  does not separate  $e$ 's vertices (or the reverse of this situation),
3. neither  $e$  nor  $f$  separate each other's vertices.

In order to implement an algorithm which employs the concepts of vertex alternation and of vertex separation, some additional notation must be introduced. Vertex alternation of four vertices  $v_i, v_j, v_k$  and  $v_\ell$  is indicated by the function  $A(ij, k\ell)$ , defined as

$$A(ij, k\ell) = \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ alternate } v_k \text{ and } v_\ell \\ 0 & \text{otherwise.} \end{cases} \quad (4.29)$$

Clearly it follows that  $A(ij, k\ell) = A(k\ell, ij)$ . Vertex separation obviously does not possess this symmetric property. For an edge  $e = \{v_i, v_j\}$ , its vertex separation of the vertices  $v_k$  and  $v_\ell$  is expressed as

$$S_{ij}(k\ell) = \begin{cases} 1 & \text{if } v_k \text{ and } v_\ell \text{ are separated by } \{v_i, v_j\} \\ 0 & \text{otherwise.} \end{cases} \quad (4.30)$$

For convenience,  $S_{ij}(k\ell)$  is taken to be zero when the edge  $\{v_i, v_j\}$  does not exist.

Using the above notation, the six possible crossing configurations for two edges  $\{v_i, v_j\}$  and  $\{v_k, v_\ell\}$  are shown in Figures 4.23(a)–(f) (recall that there is an infinity of equivalent configurations, for each depicted configuration — this is easy to see from the way that Tutte transformations generate different drawings, but equivalent configurations). From these considerations, a parameter  $\lambda^{(i)}(ij, k\ell)$ , which is similar to Tutte's  $\lambda(ij, k\ell)$  parameter (§ 4.2.1), is defined as

$$\begin{aligned} \lambda^{(i)}(ij, k\ell) &= A(ij, k\ell)S_{ij}(k\ell)S_{k\ell}(ij) \\ &\quad + A(ij, k\ell)[1 - S_{ij}(k\ell)][1 - S_{k\ell}(ij)] \\ &\quad + [1 - A(ij, k\ell)][1 - S_{ij}(k\ell)]S_{k\ell}(ij) \\ &\quad + [1 - A(ij, k\ell)]S_{ij}(k\ell)[1 - S_{k\ell}(ij)]. \end{aligned} \quad (4.31)$$

The parameter  $\lambda^{(i)}(ij, k\ell)$  describes the parity of the number of crossings between the edges  $\{v_i, v_j\}$  and  $\{v_k, v_\ell\}$ . Since vertex separation simply defines a bipartition of vertices of  $\mathcal{G}$  other

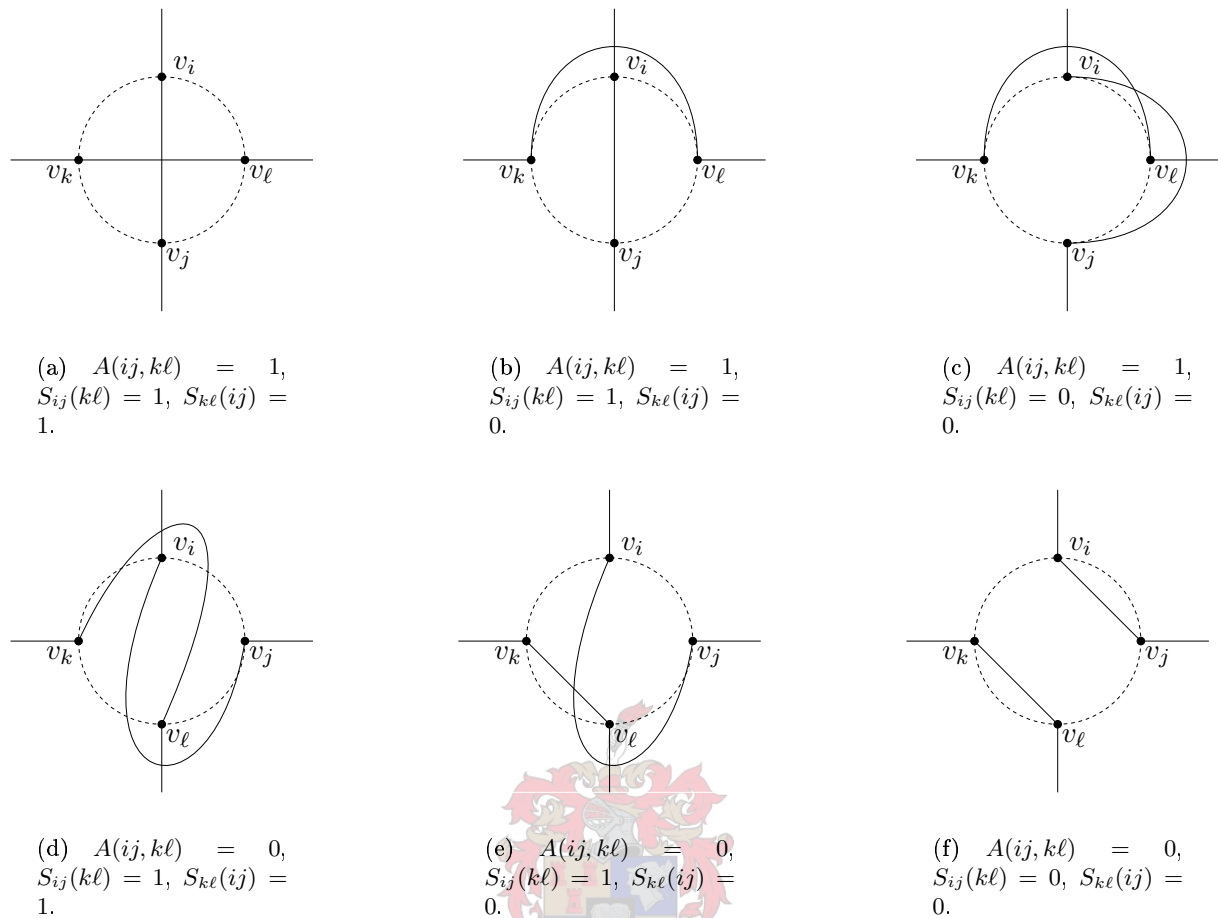


Figure 4.23: All configurations which affect the parity of the number of crossings between two edges.

than those of the edge which partitions them, one may simply discard the notion of drawings, and associate such a bipartition of vertices with each edge of  $\mathcal{G}$ . Denote the set of bipartitions corresponding to a drawing  $\phi$  by  $B(\phi)$ . It may be seen that each possible bipartition corresponds to a drawing (since edges may be woven around vertices in the desired manner). Thus, for a given drawing  $\phi$  of a graph  $\mathcal{G}$ , the independent-odd crossing number of  $\mathcal{G}$  corresponding to  $\phi$  is given by

$$\nu_{B(\phi)}^{(i)}(\mathcal{G}) = \sum_{\substack{\{v_i, v_j\}, \{v_k, v_\ell\} \in E(\mathcal{G}) \\ \{v_i, v_j\} \neq \{v_k, v_\ell\}}} \lambda^{(i)}(ij, k\ell).$$

The independent-odd crossing number of  $\mathcal{G}$  may now be defined simply as

$$\nu^{(i)}(\mathcal{G}) = \min_{\text{All bipartitions } B} \{\nu_B^{(i)}(\mathcal{G})\}.$$

A point worth examining is that different orderings of vertices around the unit circle need not be considered. This is true, because given the correct topological transformation, any drawing may be transformed with its vertices in some desired sequence on the unit circle, so that the configuration of edge crossings remain unaffected (*i.e.*, not just the parity remains unaffected, but the exact number of crossings remains unaffected). Thus, an implementation of Székely’s algorithm need only choose an initial arbitrary ordering of the vertices of a given graph around

the unit circle, and keep it fixed. The running time of the algorithm is determined by the number of bipartitions of vertices for each edge, and the time complexity of computing the number of independent-odd crossings that result from a configuration of bipartitions. Each term  $\lambda^{(i)}(ij, k\ell)$  in (4.31) may be computed in constant time. The function  $S_{ij}(k\ell)$  defined in (4.30), depends on the partitioning of  $v_k$  and  $v_\ell$  by the edge  $\{i, j\}$ , and since this information is explicitly stored (perhaps as a vector of elements), computation occurs in constant time; the same holds for the function  $S_{k\ell}(ij)$ . Because the vertex alternation is fixed throughout the algorithm, it is possible to construct a lookup matrix, indexed by edges of  $\mathcal{G}$ , which (say) contains 1 if the two index edges alternate and 0 otherwise — this could be used to determine  $A(ij, k\ell)$ , as defined in (4.29), in constant time. There are at most  $\binom{|E(\mathcal{G})|}{2}$  such terms. Therefore, the computation of  $\nu_{B(\phi)}^{(i)}(\mathcal{G})$  for a set of bipartitions takes  $O(\binom{|E(\mathcal{G})|}{2})$  time. For each edge there is a corresponding bipartition of  $|V(\mathcal{G})| - 2$  vertices. There are therefore  $2^{|V(\mathcal{G})|-2}$  configurations for a given bipartition. In total, there are therefore  $|E(\mathcal{G})|2^{|V(\mathcal{G})|-2}$  bipartition configurations for all edges. The total running time of the algorithm is thus  $O(\binom{|E(\mathcal{G})|}{2}|E(\mathcal{G})|2^{|V(\mathcal{G})|-2})$ .

### 4.3.2 Heuristic methods

One may see from the exact algorithms discussed in § 4.3.1, that their application to all but the smallest graphs (where small certainly means a graph with fewer edges than, say,  $\mathcal{K}_{10}$ ) is computationally impractical. This situation has naturally led to the development of heuristic algorithms.

#### 4.3.2.1 Recursive graph bisection

Leighton [Lei83] pioneered the idea of recursive graph bisection — that is, given a graph  $\mathcal{G}$ , one bisects  $\mathcal{G}$  into two components  $\mathcal{G}_1$  and  $\mathcal{G}_2$ , each of which is again bisected, and so on at each step, until only isolated vertices remain. Bhatt and Leighton [BL84] used this notion to develop an algorithm for graph layout that provides a drawing with the number of crossings within a factor of order  $\log(|V(\mathcal{G})|)^2$  from the crossing number of the graph. Their methods were all aimed at the design of electronic circuits, and consequently, due to the restrictions that such designs impose, they only considered graphs with a maximum degree of four. Their ideas were later generalised by Shahrokhi, Székely, Sýkora and Vrfo [SSSV96b] for graphs with arbitrary maximum degrees.

The approach discussed in this section is based on Shahrokhi, Székely, Sýkora, and Vrfo's work. A central notion to recursive graph bisection, is that of a *decomposition tree*, with which to represent the recursive bisection.

**Definition 4.3.1** *A recursive bisection of a graph  $\mathcal{G}$  leads to the notion of a decomposition tree, which is a binary tree  $T_{\mathcal{G}}$ , whose leaves correspond to isolated vertices of  $V(\mathcal{G})$ , and where the interior vertices of  $T_{\mathcal{G}}$  correspond to the sets of edges from  $E(\mathcal{G})$  whose removal bisected its corresponding subgraph (of  $\mathcal{G}$ ) into two smaller subgraphs. ■*

An example of a decomposition tree,  $T_{\mathcal{K}_{4,4}}$ , corresponding to a decomposition of  $\mathcal{K}_{4,4}$  may be seen in Figure 4.24. The bold edges represent the edges whose removal bisects the components at each level, and these bold edges are associated with the vertices of the decomposition tree within which the subgraphs are drawn.

For a graph  $\mathcal{G}$  let  $\mathcal{G}_1$  and  $\mathcal{G}_2$  be components obtained from a bisection of  $\mathcal{G}$ . Now, a one page drawing  $\phi$  of  $\mathcal{G}$  may be constructed by creating disjoint (in the sense that the drawings do not

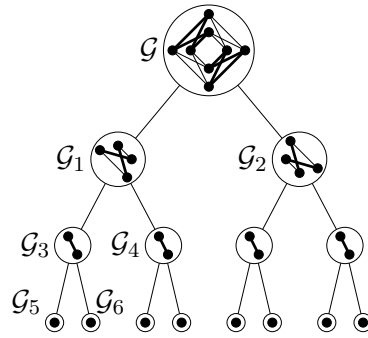


Figure 4.24: A bisection tree for  $\mathcal{K}_{4,4}$ .

intersect one another) one–page drawings  $\phi_1$  of  $\mathcal{G}_1$  and  $\phi_2$  of  $\mathcal{G}_2$ , and then by inserting the  $b(\mathcal{G})$  edges that were removed to bisect  $\mathcal{G}$ . Let  $\ell_{\phi_1}$  represent the maximum number of edges that will be crossed in  $\phi_1$  when drawing an edge from  $\mathcal{G}_1$  to  $\mathcal{G}_2$ . Similarly, let  $\ell_{\phi_2}$  denote the maximum such number of edges in  $\mathcal{G}_2$ . It must be true that

$$\nu_{\phi}(\mathcal{G}) \leq \nu_{\phi_1}(\mathcal{G}_1) + \nu_{\phi_2}(\mathcal{G}_2) + \binom{b(\mathcal{G})}{2} + (\ell_{\phi_1} + \ell_{\phi_2})b(\mathcal{G}),$$

since every one of the removed edges may cross every other, which accounts for  $\binom{b(\mathcal{G})}{2}$  crossings, and every removed edge may cross the maximum number of edges in  $\mathcal{G}_1$  and in  $\mathcal{G}_2$ , hence the term  $(\ell_{\phi_1} + \ell_{\phi_2})b(\mathcal{G})$ . Now, the fact that the bisection width of a graph may be used to bound its crossing number from below, as may be seen from (4.12), allows for the bisection width terms to be replaced by crossing number terms. Hence, the obtained crossing number  $\nu_{\phi}(\mathcal{G})$  of  $\mathcal{G}$  is bounded by a function of its crossing number  $\nu(\mathcal{G})$ . This idea is applied recursively from the bottom up (*i.e.*, beginning with isolated vertices) in the bisection tree.

The following theorem, due to Shahrokhi, Székely, Sýkora and Vrto, is a generalization of Leighton’s result for graphs with a maximum degree of four.

**Theorem 4.3.2** *With an approximation algorithm for bisecting an  $n$ -vertex graph  $\mathcal{G}$  such that the algorithm removes at most  $R(n)b(\mathcal{G})$  edges, where  $R(n)$  is a non-decreasing measure of error, a one–page drawing  $\phi$  of  $\mathcal{G}$  may be constructed so that*

$$\nu_{\phi}(\mathcal{G}) = O\left([\log n]^2 [R(n)]^2 (\nu(\mathcal{G}) + \sum_{v \in V(\mathcal{G})} [\deg_{\mathcal{G}}(v)]^2)\right). \quad (4.32)$$

**Proof:** Construct a decomposition tree  $T$  from  $\mathcal{G}$  by recursive application of the bisection algorithm to  $\mathcal{G}$ . Since a one–page drawing  $\phi'$  of  $\mathcal{G}$  is to be constructed, all vertices are drawn on the spine of a book, and for each component  $\mathcal{H}$  which is partitioned into components  $\mathcal{H}_1$  and  $\mathcal{H}_2$ , draw all the vertices of  $\mathcal{H}_1$  before drawing the vertices of  $\mathcal{H}_2$ . An edge  $\{v_i, v_j\} \in E(\mathcal{G})$  is drawn as a half circle, of which the diameter is equal to the distance from  $v_i$  to  $v_j$  on the spine. From the preceding discussion, it is known that

$$\nu_{\phi}(\mathcal{H}) \leq \nu_{\phi_1}(\mathcal{H}_1) + \nu_{\phi_2}(\mathcal{H}_2) + \binom{R(n)b(\mathcal{H})}{2} + (\ell_{\phi_1} + \ell_{\phi_2})R(n)b(\mathcal{H}), \quad (4.33)$$

where  $\ell_{\phi_1}$  and  $\ell_{\phi_2}$  denote the maximum number of half circles (which are edges) that enclose any vertex in  $\mathcal{H}_1$  and  $\mathcal{H}_2$  respectively. The total number of half circles that enclose any vertex in  $\mathcal{H}$  may also include the  $R(n)b(\mathcal{H})$  edges, and so

$$\ell_{\phi} \leq \max\{\ell_{\phi_1}, \ell_{\phi_2}\} + R(n)b(\mathcal{H}). \quad (4.34)$$

Since the decomposition tree has depth  $O(\log n)$ , the above recurrence relation (4.34) is easily solved for  $\mathcal{G}$  as

$$\ell_{\phi} = O(R(n)b(\mathcal{G}) \log n). \quad (4.35)$$

Substitution of (4.35) into (4.33), yields

$$\nu_{\phi}(\mathcal{H}) \leq \nu_{\phi_1}(\mathcal{H}_1) + \nu_{\phi_2}(\mathcal{H}_2) + O([R(n)b(\mathcal{H})]^2 \log n). \quad (4.36)$$

Further application of the bound (4.12), which when rewritten as

$$[b(\mathcal{G})]^2 \leq 1.58 \left( 16\nu(\mathcal{G}) + \sum_{v \in V(\mathcal{G})} [d_{\mathcal{G}}(v)]^2 \right), \quad (4.37)$$

yields

$$\nu_{\phi}(\mathcal{H}) \leq \nu_{\phi_1}(\mathcal{H}_1) + \nu_{\phi_2}(\mathcal{H}_2) + O\left([R(n)]^2(\nu(\mathcal{H}) + \sum_{v \in V(\mathcal{H})} [d_{\mathcal{H}}(v)]^2) \log n\right). \quad (4.38)$$

Now it remains a simple case of induction to show that:

$$\nu_{\phi}(\mathcal{G}) = O\left([R(n)]^2(\nu(\mathcal{G}) + \sum_{v \in V(\mathcal{G})} [d_{\mathcal{G}}(v)]^2) [\log n]^2\right) \quad (4.39)$$

(note the extra factor of  $\log |V(\mathcal{G})|$ ). ■

Chung and Yau [CY94] developed a bisection algorithm that finds solutions within a constant factor of the optimal value of  $\nu(\mathcal{G})$ . Therefore, the factor  $R(n)$  in the above theorem may be replaced by a constant. The bisection has to be performed  $|V(\mathcal{G})|$  times for a graph  $\mathcal{G}$ . If a bisection algorithm runs in  $O(B(\mathcal{G}))$  time, then the entire process can be accomplished at most in  $O(|V(\mathcal{G})|B(\mathcal{G}))$  time. This might be an over-estimation, since the time required to find bisections diminishes as the algorithm progresses down the bisection tree (because the graph components become smaller). The process of the construction of a drawing of  $\mathcal{K}_{4,4}$  from its bisection tree shown in Figure 4.24, is shown for the different recursive steps. Figure 4.25(a) corresponds to the third level of the bisection tree, whilst Figures 4.25(b) and (c) correspond respectively to the second and first levels of the bisection tree.

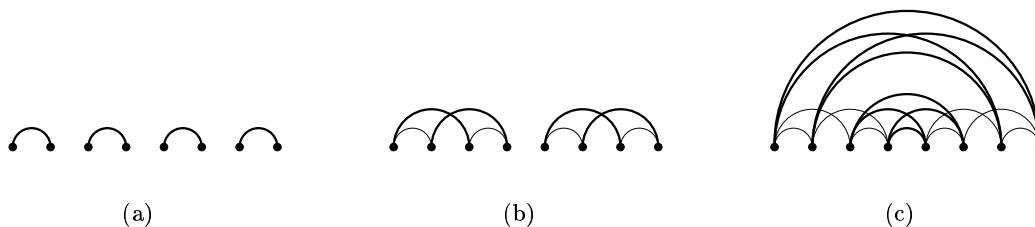


Figure 4.25: The process of reconstructing a drawing of  $\mathcal{K}_{4,4}$  from its bisection tree.

Finally, it should be noted that Bhatt and Leighton's layout algorithm constructs a drawing of a graph in a special type of binary tree with the same structure as the bisection tree. The layout tree contains meshes instead of vertices, and for a graph  $\mathcal{G}$  which must be drawn, the vertices of  $\mathcal{G}$  are embedded into the meshes which correspond to leaves of the tree, and all edges are routed through the lines of the meshes. Bhatt and Leighton called this graph the "tree of meshes," an example of which is shown in Figure 4.26, and as may be seen, the meshes are halved along

their longest sides for each step that is taken down towards the leaves, such that the leaves are  $1 \times 1$  meshes. Because the leaves and meshes on levels just above the leaves do not have a high number of edges emanating from them, it might not be possible to lay out a graph in a tree of meshes with the number of levels that appear in the bisection tree of that graph. This problem is avoided by using a tree of meshes with a larger number of levels, and by truncating it after  $n$  levels, where  $n$  is the number of levels present in the bisection tree.

Therefore, the depth of the tree of meshes is  $O(\log |V(\mathcal{G})|)$  and since the maximum number of crossings in each mesh is easily computed (it cannot be more than the total number of points in the mesh), to arrive at the results for this section. However, as has been stated, their results were only valid for graphs with a maximum degree of four. Shahrokhi, Sýkora, Székely and Vrfo’s method is easier to implement, and more general.

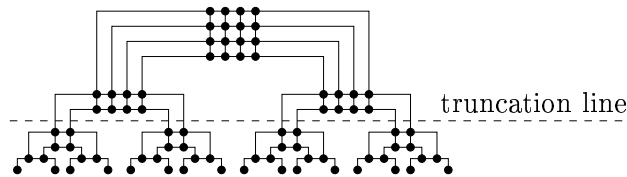


Figure 4.26: Bhatt and Leighton’s tree of meshes.

#### 4.3.2.2 Two–page layout algorithms

In principle, two–page layout methods are a generalization of one–page layouts. Therefore, any algorithm which produces a one–page layout is, by virtue of this observation, also a two–page layout algorithm (albeit a limited one). A new problem faced with two–page layouts, is to decide upon which edges are placed on which pages. This is an **NP**–hard problem [MNKF90].

##### Nicholson’s heuristic

Nicholson’s heuristic [Nic68] is one of the oldest heuristics for obtaining graph layouts, and was introduced in 1968. The algorithm creates an initial layout in a greedy fashion, followed by a post–optimization phase.

In the initial phase, for an input graph  $\mathcal{G}$ , the algorithm starts by first placing a vertex with the maximum degree  $\Delta(\mathcal{G})$  on the spine of the book. For each following placement, it finds the vertex  $v$  with the highest connectivity to vertices already on the spine, and places  $v$  in a position on the spine so that as few as possible crossings are caused (this means that at each position which is attempted, the optimal placement of edges adjacent to  $v$  must be found). In the worst case scenario, it takes  $|V(\mathcal{G})|O(\Delta(\mathcal{G}))$  time to find the vertex with the highest connectivity to the vertices already on the spine, since each vertex not on the spine must be enumerated, and all vertices adjacent to each such vertex must be enumerated to determine the level of connectivity to the vertices on the spine. For a dense graph, this time is  $O(|V(\mathcal{G})|^2)$ . The selected vertex must be inserted into each position along the spine, and for each position, it might be necessary to shift  $O(|V(\mathcal{G})|)$  vertices that are already on the spine, so as to open a position for the placement of the vertex. Then determining the number of edges that are caused by the insertion of a vertex  $v$ , requires the enumeration of its incident edges, and determining the number of edges drawn on the same page, that alternate the edges of  $v$ . In the worst case, each edge may be crossed  $O(|E(\mathcal{G})|)$  times, which translates into  $O(|E(\mathcal{G})||V(\mathcal{G})|)$  crossings that need to be counted for the inserted vertex. The vertex must be inserted in each of the  $|V(\mathcal{G})|$

positions along the spine, to ascertain the best placement. Thus, the time taken to map a vertex is  $O(|V(\mathcal{G})|^2) + O(|E(\mathcal{G})||V(\mathcal{G})|^2) = O(|E(\mathcal{G})||V(\mathcal{G})|^2)$ . Since  $|V(\mathcal{G})|$  vertices must be mapped, the total time required for the initial phase is  $O(|E(\mathcal{G})||V(\mathcal{G})|^3)$ . This is a rather loose upper bound, which may be improved if the operation count is aggregated across a number of operations, and where worst case counts are not assumed for every operation.

In the post-optimization phase, a vertex is found which has the maximum number of its incident edges crossed. This vertex is moved to a position on the spine which offers the largest reduction in the number of crossings of its adjacent edges. The process is repeated until no improvement is possible (note that this means that some vertices may be moved multiple times). It is more difficult to estimate the running time of this phase, than for the initial phase, because it is unclear how many times vertices may be moved so that improved layouts result. However, as with the placement of a vertex in the previous step, the total amount of time required to shift a vertex, and to count the number of crossings that it would be involved in takes  $O(|E(\mathcal{G})||V(\mathcal{G})|)$  time. Since each vertex has to be moved to every spine position to determine the best move,  $|V(\mathcal{G})|^2$  such moves will be performed. Therefore, the total running time required to find the best vertex to be moved is  $O(|E(\mathcal{G})||V(\mathcal{G})|^3)$ .

Cimikowski’s results [Cim02] indicate that Nicholson’s method provides good approximations in general. As with most other such heuristics, the algorithm can also be tailored, in the sense that the post-optimization phase may be replaced by alternative post-optimization techniques suitable for specific graph structures.

### Cimikowski’s approach

A fairly straightforward possibility for computing a two-page layout of a graph  $\mathcal{G}$ , is to apply a one-page layout algorithm, such as the recursive graph bisection method (§ 4.3.2.1) to  $\mathcal{G}$ , in order to obtain a vertex arrangement, and then to apply an algorithm which determines a layout of the edges on the two pages. The problem of determining an optimal edge layout, given a vertex arrangement was shown by Masuda, Nakajima, Kashiwabara and Fujisawa [MNKF90] to be **NP**-complete, meaning that this problem can only realistically be tackled by heuristic means.

Cimikowski [Cim02] studied this problem under the name of the “fixed linear crossing number problem.” For the computation of vertex orderings, his algorithms searched for Hamiltonian cycles in graphs, and placed the vertices in the order that they appear in such a cycle. The rationale is that the edges of the Hamiltonian path will each lie between pairs of vertices which are adjacent on the spine, and can therefore not be crossed. If a Hamiltonian cycle cannot be found, then the fewest number of cycles that can be found in the graph may be used to obtain edge orderings.

Several heuristic algorithms and an exact algorithm for obtaining two-page layouts were implemented by Cimikowski. His most successful heuristic algorithm is based on a neural network, and this is the only method that will be discussed here (according to his computational results, this method outperforms the other methods by far). He published a separate article with Shope [CS96] which elaborated on the neural network algorithm.

The regularity of Cimikowski and Shope’s neural network makes it possible to view it as a system of non-linear ordinary differential equations in an independent variable  $t$ . The two pages of the book on which a graph  $\mathcal{G}$  is to be drawn are called the “upper” and “lower” pages. This nomenclature describes a drawing of the book such that its spine lies on the  $x$ -axis, and edges are drawn either above the  $x$ -axis or below it. For each edge  $e$ , there are “up” and “down” functions, denoted  $U_e^{(1)}(t)$  and  $U_e^{(2)}(t)$  respectively. When  $U_e^{(1)}(t) > 0$ ,  $e$  should be drawn on the upper

page, and when  $U_e^{(l)}(t) > 0$ ,  $e$  should be drawn on the lower page. No ambiguity arises in the placement of  $e$ , when  $U_e^{(u)}(t) > 0$  and  $U_e^{(l)}(t) \leq 0$ , or when  $U_e^{(u)}(t) \leq 0$  and  $U_e^{(l)}(t) > 0$ , and it is the task of the algorithm to find a value of  $t$  so that these conditions are met for every edge in  $\mathcal{G}$ .

The functions  $U_e^{(u)}(t)$  and  $U_e^{(l)}(t)$  are, of course, unknown (since otherwise the crossing number problem would have been solved). The system of equations considered by Cimikowski and Shope are

$$\frac{dU_e^{(u)}}{dt} = A a_e(t) + B(-b_e^{(u)}(t) + b_e^{(l)}(t)) + C c_e(t), \quad (4.40)$$

$$\frac{dU_e^{(l)}}{dt} = A a_e(t) + B(-b_e^{(l)}(t) + b_e^{(u)}(t)) + C c_e(t). \quad (4.41)$$

The dominating terms in the differential equations are the coefficients of  $B$ . The function  $b_e^{(u)}(t)$  is defined as

$$b_e^{(u)}(t) = \sum_{\substack{f \in E(\mathcal{G}) \\ e \neq f}} x_e^{(u)}(f),$$

where

$$x_e^{(u)}(f) = \begin{cases} 1 & \text{if the vertices of } e \text{ and } f \text{ are alternating and } U_f^{(u)}(t) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The function  $b_e^{(l)}(t)$  is defined similarly. The value of  $b_e^{(u)}(t)$  is the total number of crossings in which  $e$  would be involved if each edge  $f \in E(\mathcal{G})$  for which  $U_f^{(u)}(t) > 0$  were to be mapped to the upper page. Likewise, the value of  $b_e^{(l)}(t)$  is the total number of crossings in which  $e$  would be involved if each edge  $f \in E(\mathcal{G})$  for which  $U_f^{(l)}(t) > 0$  were to be mapped to the lower page.

Thus, if the value  $-b_e^{(u)}(t) + b_e^{(l)}(t)$  in (4.40) is positive, then  $e$  would cross more edges on the lower page than on the upper page, which is a good reason to keep it on the upper page. This will be the case at a later time step (value of  $t$ ) due to the positive slope. However, if the value is negative, then it would be better to draw  $e$  on the lower page, and the fact that the term then induces a negative slope will ameliorate this at a later time step  $t$ . The same rationale holds for the value of  $-b_e^{(l)}(t) + b_e^{(u)}(t)$  in (4.41).

The remaining two terms are rather easy to justify. It is undesirable for an edge  $e \in \mathcal{G}$  that both  $U_e^{(u)}(t) > 0$  and  $U_e^{(l)}(t) > 0$ , or both  $U_e^{(u)}(t) \leq 0$  and  $U_e^{(l)}(t) \leq 0$ , since these cases correspond to the situations where respectively  $e$  is designated to be drawn on two pages, or otherwise designated not to be drawn at all. The function  $a_e(t)$  alleviates this problem, decreasing both in the former case, since it is likely that one will change sign sooner than the other. The function  $a_e(t)$  increases both in the latter case for the same reason and is defined by

$$a_e(t) = \begin{cases} -1 & \text{if } U_e^{(u)}(t) > 0, U_e^{(l)}(t) > 0 \\ 1 & \text{if } U_e^{(u)}(t) \leq 0, U_e^{(l)}(t) \leq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Finally, for an edge  $e \in E(\mathcal{G})$ , the term containing  $c_e(t)$  acts as a hill climbing function, which forces the system of equations out of the situation where  $e$  is not drawn on any page (and thus it steers the equations towards feasible solution sets). Its definition is rather simple, and is given by

$$c_e(t) = \begin{cases} 1 & \text{if } U_e^{(u)}(t) \leq 0 \text{ and } U_e^{(l)}(t) \leq 0, \\ 0 & \text{otherwise.} \end{cases}$$



This set of differential equations can almost certainly not be solved analytically, and it is most easily solved by Euler’s first order forward integration method, although other methods could conceivably be used. The constant  $\delta t$  specifies the size of the time steps in the discretized system

$$U_e^{(\uparrow)}(t+1) = U_e^{(\uparrow)}(t) + \frac{dU_e^{(\uparrow)}}{dt}\delta t \quad (4.42)$$

$$U_e^{(\downarrow)}(t+1) = U_e^{(\downarrow)}(t) + \frac{dU_e^{(\downarrow)}}{dt}\delta t. \quad (4.43)$$

For a graph  $\mathcal{G}$ , the set of equations is said to converge at a time  $t$ , when for each edge  $e \in E(\mathcal{G})$ , it holds that either  $U_e^{(\uparrow)}(t) > 0$  and  $U_e^{(\downarrow)}(t) \leq 0$ , or that  $U_e^{(\uparrow)}(t) \leq 0$  and  $U_e^{(\downarrow)}(t) > 0$ . This is, of course, by no means guaranteed to happen, but since the set of equations is solved iteratively, a limit can be put on the maximum number of iterations that are acceptable.

The initial values  $U_e^{(\uparrow)}(0)$  and  $U_e^{(\downarrow)}(0)$  for each edge  $e \in E(\mathcal{G})$  are randomly assigned values in the range  $(-\omega, 0)$ . The randomness gives the set of differential equations the opportunity to converge to different answers during repeated applications. If  $\omega$  is very large, convergence may take unreasonably long, as the values of  $U_e^{(\uparrow)}$  and  $U_e^{(\downarrow)}$  may vary slowly. The constants  $A$ ,  $B$  and  $C$  and the time step  $\delta t$  alter the convergence rate of the set of equations, and their values should be determined through experimentation. Cimikowski and Shope recommended the values  $A = 1.0$ ,  $B = 2.0$ ,  $C = 2.0$  and  $\delta t = 10^{-5}$ . They further limited the values  $U_e^{(\uparrow)}(t)$  and  $U_e^{(\downarrow)}(t)$  to a range, such as  $[-1, 1]$ , although they did not stipulate exactly how this should be achieved. The running time of the algorithm is discussed in detail later in the thesis, where a concrete implementation of the algorithm is provided.

#### 4.3.2.3 Shahrokhi, Székely, Sýkora and Vrto’s probabilistic embedding algorithm

The algorithm of Shahrokhi, Székely, Sýkora and Vrto [SSSV96c] is rather simple in principle. It uses the notion of single-edge graph-to-graph embeddings (see § 4.2.2.2), and constructs such an embedding  $\bar{\psi}$  from an order  $n$  graph  $\mathcal{G}$  into  $\mathcal{K}_p$  for which  $p \geq n$ , one vertex at a time. Given a partial single-edge graph-to-graph embedding  $f$  — that is, an embedding for which some vertices in  $\mathcal{G}$  are not mapped to vertices in  $\mathcal{K}_p$  — the algorithm chooses an unmapped vertex  $v \in V(\mathcal{G})$ , and a mapping  $\bar{\psi}(v)$  of  $v$ , that minimizes the sum of

1. the crossings in  $\phi(\mathcal{K}_p)$  where both edges involved in a crossing are the images of edges in  $\mathcal{G}$  under  $f$  (including any crossings introduced by the image  $\bar{\psi}(v)$  of  $v$ ),
2. the expected value of the number of crossings that will be caused by a random mapping of the remaining (unmapped) vertices into  $\mathcal{K}_p$ .

This process is initiated with an empty mapping  $f$  (*i.e.*, where every vertex of  $\mathcal{G}$  is unmapped), and it continues through a total of  $n$  iterations, where at each iteration, a unique vertex from  $\mathcal{G}$  is mapped, until after the  $n$ -th iteration, every vertex of  $\mathcal{G}$  is mapped.

In order to construct the single-edge graph-to-graph embedding  $\bar{\psi}$  from  $\mathcal{G}$  into  $\mathcal{K}_p$ , a total of  $n$  intermediate injections  $f : U_t \rightarrow V(\mathcal{K}_p)$  must be considered, where  $|U_t| = t$ ,  $0 \leq t \leq n$ , and  $U_{t-1} \subset U_t$ ,  $1 \leq t \leq n$ . The vertex set  $U_t$  denotes the  $t$  mapped vertices at the  $t$ -th iteration of algorithm.

The process of determining the expected number of crossings due to a random single-edge graph-to-graph embedding is the most involved part of the algorithm, but it follows from a number of simple probabilistic arguments.

For a crossing  $y$ , that is the intersection of two edges  $e_1 = \{v_i, v_j\}, e_2 = \{v_k, v_\ell\} \in E(\mathcal{K}_n)$  in a drawing  $\phi$  of  $\mathcal{K}_p$ , define the weight of  $y$  as  $w_t^f(y)$ . This is a measure of how much  $y$  contributes to the number of crossings in a drawing  $\phi_2$  of  $\mathcal{G}$  obtained from the drawing  $\phi$  of  $\mathcal{K}_n$ , and a given mapping  $f$  at iteration  $t$  of the algorithm.

If the incident vertices of  $e_1$  and  $e_2$  are images of vertices from  $V(\mathcal{G})$  under  $f$  at iteration  $t$ , then it is certain that  $y$  will contribute either no crossings or a single crossing, depending on whether  $e_1$  and  $e_2$  are both images of edges in  $E(\mathcal{G})$ . In these cases, the weight of  $y$  is defined as  $w_t^f(y) = 0$  or  $w_t^f(y) = 1$ , respectively for no crossings, and for a crossing.

On the other hand, if  $x$  of the vertices that are incident to  $e_1$ , or to  $e_2$ , or both, are not images of vertices in  $U_t$  under  $f$ , then a random mapping of  $x$  vertices from  $\overline{U_t}$  into  $f(\overline{U_t})$  has a probability of  $1/\binom{p-t}{x}$  to be mapped to all  $x$  of the vertices that are incident to  $e_1$  and/or  $e_2$ . Suppose that this is the case, then it does not follow that edges from  $\mathcal{G}$  will necessarily be mapped to both  $e_1$  and  $e_2$ , since some of the  $x$  (unmapped) vertices in  $\mathcal{G}$  may not be joined by edges to the vertices of  $e_1$  or  $e_2$  that have already been mapped. Let  $A$  denote the number of ways in which  $x$  vertices may be selected from  $\overline{U_t}$  so that both  $e_1$  and  $e_2$  would be images of edges in  $\mathcal{G}$  and let  $B$  denote the total number of ways in which  $x$  vertices may be selected from  $\overline{U_t}$ . From these considerations, the probability of  $e_1$  and  $e_2$  both being images under a random mapping  $f$ , and therefore that they will contribute to a crossing in the drawing  $\phi_2$  of  $\mathcal{G}$  obtained from  $\phi(\mathcal{K}_p)$ , is  $A/B$ . Accordingly, the weight of the crossing  $y$  is defined to be  $w_t^f(y) = A/B$  in this case. The expected number of crossings  $\beta_t^f$  in a drawing  $\phi_2$  of  $\mathcal{G}$  for the mapping  $f$  at the iteration  $t$  is given by

$$\beta_t^f = \sum_{\substack{y \text{ is a crossing} \\ \text{in } \phi(\mathcal{K}_n)}} w_t^f(y).$$

For a graph  $\mathcal{G} = (V, E)$ , let  $U \subseteq V$ . Define the following sets of which the first is a set of pairs of edges in  $\mathcal{G}$ , and the second is a set of vertices in  $\mathcal{G}$ :

$$\begin{aligned} \Xi(\mathcal{G}) &= \{ \{e, f\} : e, f \in E(\mathcal{G}) \text{ and } e \text{ is not adjacent to } f \} \text{ and} \\ A_U(v) &= \{ u : u \in U \text{ and } u \text{ is adjacent to } v \}. \end{aligned}$$

For a set  $S \subseteq V(\mathcal{G})$ , let  $A_U(S) = \cup_{u \in S} A_U(u)$ . Now, the different weight cases for a pair of edges may be computed. These cases depend on which of the vertices incident to the pair of edges  $e_1 = \{v_i, v_j\}$  and  $e_2 = \{v_k, v_\ell\}$  in  $\mathcal{K}_p$  are images of vertices in  $\mathcal{G}$  under a partial embedding  $f$ .

**Case 1**  $v_i, v_j, v_k, v_\ell \in f(U)$ :

$$w_U^f = \begin{cases} 1 & \text{if } e_1 \text{ and } e_2 \text{ are images of edges in } \mathcal{G} \text{ under } f \\ 0 & \text{otherwise.} \end{cases}$$

**Case 2**  $v_i, v_j, v_k, v_\ell \in \overline{f(U)}$ :

If none of the four vertices incident to  $e_1$  and  $e_2$  are images under  $f$ , then any of the unmapped edges in  $\Xi(\overline{U_t})$  could potentially be mapped to a pair of edges with incident vertices  $v_i, v_j, v_k$  and  $v_\ell$ . Only a third of such mappings would translate into a crossing, as illustrated in Figures 4.27(a)–(c). Therefore, there are a total of  $|\Xi(\overline{U_t})|/3$  different ways in which to map four vertices from  $U_t$  so that  $e_1$  and  $e_2$  will both be the images of edges under the mapping. The probability of choosing  $v_i, v_j, v_k$  and  $v_\ell$  is  $1/\binom{p-t}{4}$ , and therefore the probability that two edges from  $\mathcal{G}$  will be mapped to  $e_1$  and  $e_2$  is given by

$$w_U^f(y) = \frac{|\Xi(\overline{U_t})|}{3\binom{p-t}{4}}.$$

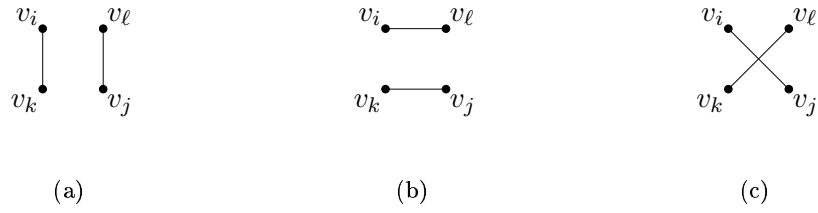


Figure 4.27: In case 2, only one in three mappings of a pair of edges to the vertices  $v_i, v_j, v_k$  and  $v_l$  will result in a crossing.



Figure 4.28: For case 3, the particular mappings to the incident vertices  $v_l$  and  $v_j$  determine whether a crossing occurs.

**Case 3**  $v_i, v_k \in f(U_t)$  and  $v_j, v_l \in \overline{f(U_t)}$ :

If each of  $e_1$  and  $e_2$  are incident to a vertex in  $\overline{f(U_t)}$ , then since  $f^{-1}(v_i) \in U_t$  is adjacent to the vertices  $A_{\overline{U_t}}(f^{-1}(v_i))$ , any of which may be selected randomly to map to  $v_j$ , and since  $f^{-1}(v_k) \in U_t$  is adjacent to the vertices  $A_{\overline{U_t}}(f^{-1}(v_k))$ , any of which may be mapped to  $v_l$ , there is a total of  $|A_{\overline{U_t}}(f^{-1}(v_i))||A_{\overline{U_t}}(f^{-1}(v_k))|$  ways in which to choose vertices from  $\overline{U_t}$  to map to  $v_j$  and  $v_l$ . However,  $|A_{\overline{U_t}}(f^{-1}(v_i)) \cap A_{\overline{U_t}}(f^{-1}(v_k))|$  of these enumerated mappings account for cases where the same vertex in  $\overline{U_t}$  is mapped to  $v_j$  and  $v_l$ , which is impossible — therefore this term must be subtracted. If the vertex which is adjacent to  $f^{-1}(v_i)$  is mapped to  $v_l$ , and the vertex which is adjacent to  $f^{-1}(v_k)$  is mapped to  $v_j$ , then no crossing results, as shown in Figure 4.28(a). The only other possibility is where the vertex adjacent to  $f^{-1}(v_i)$  is mapped to  $v_j$  and the vertex which is adjacent to  $f^{-1}(v_k)$  to  $v_l$ , as illustrated in Figure 4.28(b). In the latter case, a crossing certainly occurs. For this reason, only a half of all mappings would result in crossings. The probability of selecting  $v_l$  and  $v_j$  as images is  $1/\binom{p-t}{2}$  so that the weight of  $y$  is

$$w_U^f(y) = \frac{|A_{\overline{U_t}}(f^{-1}(v_i))||A_{\overline{U_t}}(f^{-1}(v_k))| - |A_{\overline{U_t}}(f^{-1}(v_i)) \cap A_{\overline{U_t}}(f^{-1}(v_k))|}{2\binom{n-t}{2}}.$$

**Case 4**  $v_i, v_j \in f(U_t)$  and  $v_k, v_l \in \overline{f(U_t)}$ :

There is a total of  $|E(\overline{U_t})|$  unmapped edges whose incident vertices may be mapped, so that an edge is mapped to  $e_2$ . For a crossing to occur at all, the edge  $\{v_i, v_j\}$  must exist as an image under  $f$ . The probability of choosing  $v_k$  and  $v_l$  is  $1/\binom{p-t}{2}$ , and therefore the probability that the crossing  $y$  contributes to the crossings in  $f$  is

$$w_U^f(y) = \begin{cases} |E(\overline{U_t})|/\binom{p-t}{2} & \text{if } \{v_i, v_j\} \text{ is the image of an edge in } E(\mathcal{G}) \text{ under } f \\ 0 & \text{otherwise.} \end{cases}$$

**Case 5**  $v_i, v_j, v_k \in f(U_t)$  and  $v_l \in \overline{f(U_t)}$ :

If only  $v_l$  is not the image of a vertex from  $U_t$  under  $f$ , then any of the adjacent vertices

$A_{\overline{U}_t}(f^{-1}(v_k))$  of the vertex  $f^{-1}(v_k)$  in  $\overline{U}_t$  may be selected randomly. The probability of selecting  $v_\ell$  from  $\overline{f(U)}$  is  $1/\binom{p-t}{1}$ , and it therefore follows that

$$w_U^f(y) = \begin{cases} |A_{\overline{U}_t}(f^{-1}(v_k))| / \binom{p-t}{1} & \text{if } \{v_i, v_j\} \text{ is the image of an edge in } E(\mathcal{G}) \text{ under } f \\ 0 & \text{otherwise.} \end{cases}$$

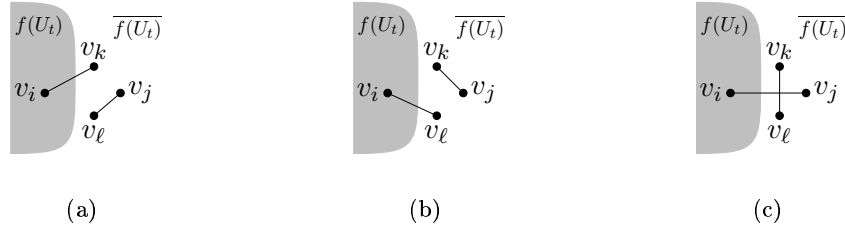


Figure 4.29: The vertex adjacent to  $f^{-1}(v_i)$  must map to  $v_j$  in for crossing to occur in case 6.

**Case 6**  $v_i \in f(U_i)$  and  $v_j, v_\ell, v_k \in \overline{f(U_t)}$ :

Since each vertex that is adjacent to the vertex  $f^{-1}(v_i)$  may be mapped to ensure that the first edge is mapped to  $\mathcal{K}_p$ . For each such vertex in  $A_{\overline{U}_t}(f^{-1}(v_i))$ , there is a total of  $E(\langle \overline{U}_t \setminus u \rangle)$  ways to select a pair of vertices to map to the other edge. Therefore, the total number of ways in which to map two edges to the vertices  $v_i, v_j, v_\ell$  and  $v_k$ , is given by the sum of all the edge terms taken over all vertices adjacent to  $f^{-1}(v_i)$  in  $A_{\overline{U}_t}(f^{-1}(v_i))$ . Unless the vertex adjacent to  $f^{-1}(v_i)$  is mapped to  $v_j$ , no crossing will occur, as illustrated in Figure 4.29. Therefore, only one of the three permutations of the mappings of  $v_j, v_\ell$  and  $v_k$  will result in a crossing. The probability of choosing  $v_j, v_\ell$  and  $v_k$  is  $1/\binom{p-t}{3}$ , so that the probability that  $y$  will contribute to a crossing is

$$w_U^f(y) = \frac{\sum_{u \in A_{\overline{U}_t}(f^{-1}(v_i))} |E(\langle \overline{U}_t \setminus u \rangle)|}{3 \binom{p-t}{3}}.$$

Shahrokhi, Székely, Sýkora and Vrto [SSSV96c] proved that the number of crossings in a drawing  $\phi_2$  of  $\mathcal{G}$ , obtained by the embedding method from a drawing of  $\phi$  of  $\mathcal{K}_p$  is subject to the inequality

$$x \leq \frac{8 \binom{m}{2} \nu_\phi(\mathcal{K}_p)}{p(p-1)(p-2)(p-3)}.$$

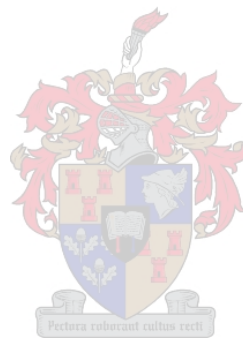
However, suppose that the drawing of  $\mathcal{K}_p$  realizes its crossing number, and that  $p \leq 10$ . In this case (see § 4.2.3.3),  $\nu_\phi(\mathcal{K}_p)$  may be replaced by  $1/4 \lfloor p/2 \rfloor \lfloor (p-1)/2 \rfloor \lfloor (p-2)/2 \rfloor \lfloor (p-3)/2 \rfloor$ , so that

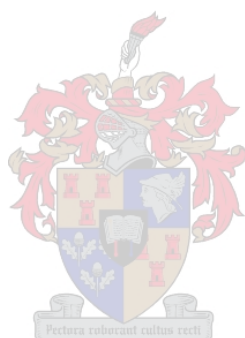
$$\frac{8 \binom{m}{2} \frac{1}{4} \lfloor p/2 \rfloor \lfloor (p-1)/2 \rfloor \lfloor (p-2)/2 \rfloor \lfloor (p-3)/2 \rfloor}{p(p-1)(p-2)(p-3)} \sim \frac{\binom{m}{2}}{8}.$$

The algorithm provides no guarantees with respect to the quality of its solutions in terms of the crossing number of the input graph  $\mathcal{G}$ , as the bisection algorithm (§ 4.3.2.1) does, except that the minimum weight decreases monotonically as the algorithm progresses. What this algorithm has in its favour is that if it can easily be determined which edges in the drawing of a complete graph should cross each other, then the rest of the algorithm is fairly easily implemented. At each step in the algorithm, all remaining unmapped vertices have to be considered, and each has to be mapped to every vertex in  $\mathcal{K}_p$  to determine the best mapping. This process is repeated  $|V(\mathcal{G})|$  times for an input graph  $\mathcal{G}$ . Therefore, the algorithm runs in  $O(p|V(\mathcal{G})|^2)$  time.

## 4.4 Chapter summary

In the first section, § 4.1, a brief overview of graph parameters — which, like the crossing number of a graph, may be seen as ways of measuring the non-planarity of graphs — was given. This was followed by § 4.2, in which the crossing algebra of Tutte was discussed. Bounding techniques and common proof techniques in crossing number theory were also described in this section, and analytical bounds for various classes of graphs were catalogued. The last section of the chapter, § 4.3, was concerned with the computational perspective of the crossing number problem. Brute force algorithms for determining the crossing number of a graph were first discussed in § 4.3.1, of which the Garey–Johnson algorithm, and the independent–odd crossing number algorithm of Székely are the most important. In the second half of the section on computational methods, § 4.3.2, various heuristic algorithmic approaches to the crossing number problem were discussed.





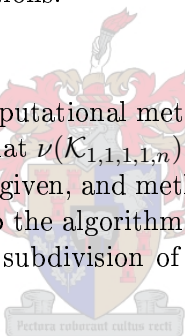
## Chapter 5

# Exact methods and novel results

It is truly distasteful when authors fabricate their own quotations, seemingly thinking them as wise as the words from sages, which they proceed to attach to their own work, attributing them to “anon” authors, as if the reader would be foolish enough to be swindled into believing such obvious deceptions.

— Anon (1979 - )

The main focus of this chapter is on computational methods for determining the crossing number of a graph exactly. Firstly, it is shown that  $\nu(\mathcal{K}_{1,1,1,1,n}) = \nu(\mathcal{K}_{4,n}) + n$ . Secondly, an implementation for the Garey–Johnson algorithm is given, and methods for exploiting symmetry information in graphs are discussed so as to speed up the algorithm. Finally, it is proved that, for any graph, the two–page crossing number of some subdivision of the graph is equal to the plane crossing number of the graph.



### 5.1 The crossing number of $\mathcal{K}_{1,1,1,1,n}$

In this section the method of edge set partitioning, as discussed in the previous chapter, is employed to prove the following theorem.

#### Theorem 5.1.1

$$\nu(\mathcal{K}_{1,1,1,1,n}) = \nu(\mathcal{K}_{4,n}) + n = \left\lfloor \frac{n^2 + 1}{2} \right\rfloor.$$

**Proof:** The graph  $\mathcal{K}_{1,1,1,1,n}$  has four partite sets of size one each. Denote the vertices in these partite sets by  $a_1$ ,  $b_1$ ,  $c_1$  and  $d_1$ . Denote the fifth partite set of size  $n$  by  $Z$ , and its elements by  $Z = \{z_1, z_2, \dots, z_n\}$ . The proof is divided into two parts: in the first part it is shown that  $\nu(\mathcal{K}_{1,1,1,1,n}) \leq \nu(\mathcal{K}_{4,n}) + n$  and in the second part that  $\nu(\mathcal{K}_{1,1,1,1,n}) \geq \nu(\mathcal{K}_{4,n}) + n$ .

**Part 1** —  $\nu(\mathcal{K}_{1,1,1,1,n}) \leq \nu(\mathcal{K}_{4,n}) + n$ :

First, create a drawing  $\phi$  of  $\mathcal{K}_{1,1,1,1,n}$  so that the vertices from  $Z = \{z_i : i \in \{1, \dots, n\}\}$  are positioned at coordinates

$$\phi(z_i) = \begin{cases} (0, i) & \text{if } i \text{ is odd} \\ (0, -i) & \text{if } i \text{ is even} \end{cases}$$

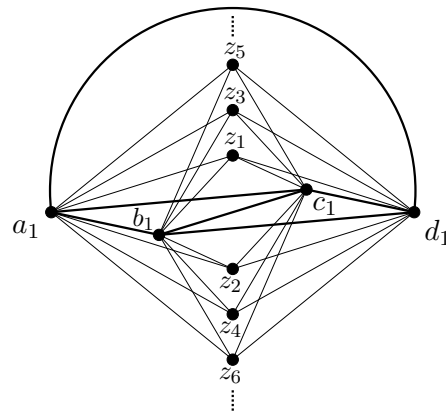


Figure 5.1: A drawing  $\phi$  of  $\mathcal{K}_{1,1,1,1,n}$  realising  $\nu_\phi(\mathcal{K}_{1,1,1,1,n}) = \nu(\mathcal{K}_{4,n}) + n$ .

in the plane. Now, place the remaining vertices  $a_1, b_1, c_1, d_1$ , so that

$$\begin{aligned} \phi(a_1) &= (-\lceil n/2 \rceil - 1, 0), \\ \phi(b_1) &= (-1, -\varepsilon), \\ \phi(c_1) &= (1, \varepsilon), \\ \phi(d_1) &= (\lceil n/2 \rceil + 1, 0), \end{aligned}$$

where  $\varepsilon > 0$  is a real number.

Draw all edges as straight lines, except for the edge between  $a_1$  and  $d_1$ , which should be drawn as a half circle, with centre  $(0, 0)$ , and radius  $\lceil n/2 \rceil + 1$ . This construction is depicted in Figure 5.1.

The removal of the edges between the induced subgraph  $\mathcal{K}_4$  on the vertices  $a_1, b_1, c_1$  and  $d_1$  (these edges are drawn as thicker lines in Figure 5.1), yields a drawing of  $\mathcal{K}_{4,n}$ , realising its crossing number. The only edges between  $a_1, b_1, c_1$  and  $d_1$  which are involved in any crossings in the construction, are  $\{a_1, c_1\}$  and  $\{b_1, d_1\}$ . By the construction,  $\{a_1, c_1\}$  is crossed  $\lceil n/2 \rceil$  times, and  $\{b_1, d_1\}$  is crossed  $\lfloor n/2 \rfloor$  times. Summation of these terms gives the desired result.

**Part 2** —  $\nu(\mathcal{K}_{1,1,1,1,n}) \geq \nu(\mathcal{K}_{4,n}) + n$ :

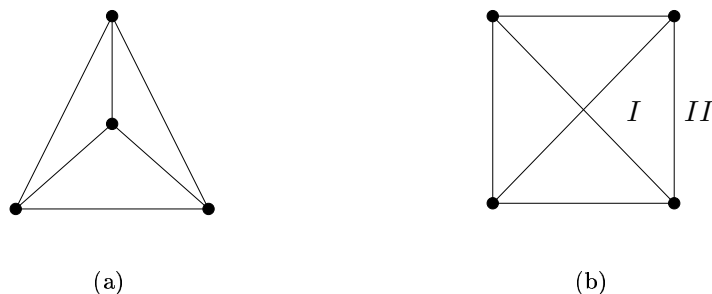


Figure 5.2: The two possible drawings of  $\mathcal{K}_4$  in the plane.

Assume, for the purposes of a proof by contradiction, that

$$\nu(\mathcal{K}_{1,1,1,1,n}) < \nu(\mathcal{K}_{4,n}) + n. \tag{5.1}$$



Since the vertices  $a_1, b_1, c_1, d_1$  induce the graph  $\mathcal{K}_4$ , one may write

$$\nu_\phi(\mathcal{K}_{1,1,1,1,n}) = \nu_\phi(\mathcal{K}_4) + \nu_\phi(\mathcal{K}_{4,n}) + \nu_\phi(\mathcal{K}_4, \mathcal{K}_{4,n}), \quad (5.2)$$

using the edge set partitioning method (described in § 4.2.2.5). Together, (5.1), (5.2), and further application of the edge set partitioning method on the term  $\nu_\phi(\mathcal{K}_4, \mathcal{K}_{4,n})$  imply that

$$\sum_{i=1}^n \nu_\phi(\mathcal{K}_4, E_{\mathcal{K}_4}(z_i)) < n - \nu_\phi(\mathcal{K}_4). \quad (5.3)$$

where  $E_U(v) = \{e : e \text{ joins } v \text{ with some } v \in V(U)\}$ . Now, there are exactly two distinct drawings of  $\mathcal{K}_4$  in the plane (up to isomorphism, where isomorphism is defined on graphs as usual, but where crossings are represented as artificial vertices of degree four) in single-cross normal form, which are shown in Figures 5.2(a) and (b). Consider the two cases separately.

**Case A:**  $\nu_\phi(\mathcal{K}_4) = 1$

For this case,  $\phi$  is equivalent to the drawing which is depicted in Figure 5.2(b). The inequality (5.3) implies that there is at least one  $z_i \in Z$ , such that  $\nu_\phi(\mathcal{K}_4, E_{\mathcal{K}_4}(z_i)) = 0$ . Let  $i = 1$  without loss of generality. From the two types of regions in  $\phi$  (the other regions are symmetrical to  $I$ ), shown in Figure 5.2(b), it may be seen that the only region in which  $z_1$  may be drawn so that none of its edges would cross  $\mathcal{K}_4$  in  $\phi$ , is region  $II$ .

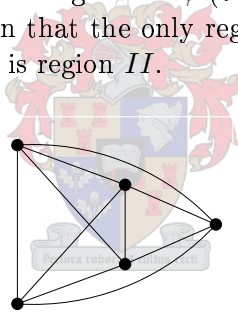


Figure 5.3:  $\mathcal{F} = \mathcal{K}_4 \cup \langle E_{\mathcal{K}_4}(z_1) \rangle$ .

Thus, place  $z_1$  there, and let  $\mathcal{F} = \mathcal{K}_4 \cup \langle E_{\mathcal{K}_4}(z_1) \rangle$ . This drawing is shown in Figure 5.3. Re-application of the edge partition method, with the edges of the graph  $\mathcal{F}$  as one partition, gives

$$\nu_\phi(\mathcal{K}_{1,1,1,1,n}) = \nu_\phi(\mathcal{F}) + \nu_\phi(\mathcal{K}_{4,n-1}) + \sum_{i=2}^n \nu_\phi(\mathcal{F}, E_{\mathcal{F}}(z_i)),$$

and a combination with (5.1) yields

$$\begin{aligned} \sum_{i=2}^n \nu_\phi(\mathcal{F}, E_{\mathcal{F}}(z_i)) &< n - 1 + \nu(\mathcal{K}_{4,n}) - \nu(\mathcal{K}_{4,n-1}) \\ &= n - 1 + 2 \left\lfloor \frac{n-1}{2} \right\rfloor \\ &\leq 2(n-1). \end{aligned}$$

This inequality implies that there exists at least one  $z_i \in Z \setminus \{z_1\}$ , such that  $\nu_\phi(\mathcal{F}, E_{\mathcal{F}}(z_i)) \leq 1$ . This is impossible, given the drawing in Figure 5.3.

**Case B:**  $\nu_\phi(\mathcal{K}_4) = 0$

The drawing in Figure 5.2(a) represents this case. The inequality (5.3) implies that there must be at least one  $z_i \in Z$ , such that  $\nu_\phi(\mathcal{K}_4, E_{\mathcal{K}_4}(z_i)) = 0$ . This is not possible for the given drawing.

Thus it must be assumed that  $\nu(\mathcal{K}_{1,1,1,1,n}) \geq \nu(\mathcal{K}_{4,n}) + n$ , which completes the proof. ■

## 5.2 An implementation of the Garey–Johnson algorithm

As has been mentioned in § 4.3.1.1, the Garey–Johnson algorithm obtains its name from the method used by Garey and Johnson [GJ83] to determine whether the crossing number of a graph is smaller than or equal to a given number. An implementation of the algorithm is provided in this section.

The algorithm itself is divided into three major parts, which are individually quite simple. Algorithm 5.1 (**GareyJohnson**), which is the starting point, enumerates all  $k$  pairs of crossing edges (where  $k$  is the number of crossings to test for). It launches Algorithm 5.2 (**TestPlanar**) for each set of  $k$  pairs of edges, and **TestPlanar** enumerates all permutations of crossings over each edge for the given set of  $k$  crossings. In its turn, for each set of permutations, it calls Algorithm 5.3 (**ConstructGraph**), which constructs a new graph from the input graph where the crossings that are specified by the permutations of edges pairs are represented as artificial vertices. The algorithm **TestPlanar** then determines whether this graph is planar.

---

**Algorithm 5.1** GareyJohnson: The Garey–Johnson algorithm

---

**Input:** A graph  $\mathcal{G}$ .

**Output:** The crossing number  $\nu(\mathcal{G})$  of  $\mathcal{G}$ , and a drawing realising  $\nu(\mathcal{G})$  where crossings are modelled as artificial vertices of degree 4.

```

1: for all  $C \in \{(p_1, p_2, \dots, p_k) : (p_1, p_2, \dots, p_k) \text{ is a set of } k \text{ pairs of edges of } \mathcal{G}\}$  do
2:    $\mathbf{D} \leftarrow \emptyset$ 
3:   for all  $e \in E(\mathcal{G})$  do
4:      $\mathbf{L}_e \leftarrow \emptyset$ 
5:   end for
6:   for all  $(e, f) \in C$  do
7:      $\mathbf{L}_e \leftarrow \mathbf{L}_e \cup \{f\}$ 
8:      $\mathbf{L}_f \leftarrow \mathbf{L}_f \cup \{e\}$ 
9:      $\mathbf{D} \leftarrow \mathbf{D} \cup \{e\}$ 
10:     $\mathbf{D} \leftarrow \mathbf{D} \cup \{f\}$ 
11:  end for
12:  if TestPlanar( $\mathbf{L}$ ,  $\mathbf{D}$ , FirstElement( $\mathbf{D}$ ),  $\mathcal{G}$ ) then
13:    return TRUE
14:  end if
15: end for
16: return FALSE

```

---

**Implementation:** An implementation of an improved version of this algorithm, described later in this chapter, is given in § B.4.1.

---

### 5.2.1 Description of the GareyJohnson algorithm

Two data structures are created by the algorithm: A list of lists  $\mathbf{L}$ , indexed by an edge  $e$ , records all edges that cross  $e$ , and a list  $\mathbf{D}$  contains the edges that are to be crossed for a given crossing configuration<sup>1</sup>.

The algorithm enumerates all crossing configurations with  $k$  crossings and the loop spanning lines 1–15, is responsible for this. The list  $\mathbf{D}$  is cleared at line 2, so that it may be filled with the crossed edges for the current iteration of the loop. The same is done for every list in  $\mathbf{L}$  in lines 3–5. From line 6 to line 11,  $\mathbf{D}$  and  $\mathbf{L}$  are filled with the crossing information needed later in the algorithm. It may be seen on lines 6 & 7, that a crossing between the edges  $e$  and  $f$  causes  $e$  to be inserted into  $f$ 's list, and *vice versa*. Lines 9 & 10 insert both  $e$  and  $f$  into the list  $\mathbf{D}$ , if they are not already there.

All permutations of edge crossings for the given crossing configuration are enumerated by the algorithm `TestPlanar`, which is called in line 12. The function `FirstElement(D)`, as its name suggests, returns the first element in the list  $\mathbf{D}$ . If the `TestPlanar` algorithm finds a set of crossing permutations for which  $\mathcal{G}$  has  $k$  crossings, it returns the boolean value “TRUE”, in which case the whole Garey–Johnson algorithm terminates and returns “TRUE” at line 13. If, after enumerating all possible configurations of  $k$  crossings, it is not possible to draw  $\mathcal{G}$  with  $k$  crossings, then the boolean value “FALSE” is returned at line 16.

### 5.2.2 Description of the TestPlanar algorithm

Given sets  $\mathbf{L}$  and  $\mathbf{D}$ , the `TestPlanar` algorithm generates every possible permutation of crossings for every edge, and determines whether such a configuration can manifest a drawing with  $k$  crossings, by constructing a new graph, where crossings are represented as artificial vertices of degree four, and by testing this graph for planarity (as is explained in § 4.3.1.1).

The distinct *sets of permutations* are generated recursively, which has the effect of producing a lexicographical ordering of the sets of edges in  $\mathbf{L}$ . Every permutation of the list  $\mathbf{L}_e$  for an edge  $e$  in the list of crossed edges  $\mathbf{D}$  is generated, and for each such step, all possible permutations of crossings for all edges after  $e$  in  $\mathbf{D}$  are generated (by the algorithm executing itself recursively) and when no more edges remain, the feasibility of the crossing configuration is determined for the set of permutations.

Permutations are themselves generated lexicographically. For an edge  $e$ , the loop spanning lines 2–16 enumerates the permutations of crossings over  $e$ , by advancing the permutation stored in  $\mathbf{L}_e$  on line 12. For each iteration of the loop, the algorithm first determines in line 3 whether there are any remaining edges in the list  $\mathbf{D}$ . If this is the case, it calls itself recursively (line 4). Otherwise,  $e$  is the last edge to be considered in  $\mathbf{D}$  and the algorithm executes `ConstructGraph` in line 8, which constructs the graph  $\mathcal{G}'$  from  $\mathcal{G}$ , such that the crossings given in  $\mathbf{L}$  are present in  $\mathcal{G}'$  as artificial vertices of degree four. A suitable planarity testing algorithm (for example the Hopcroft–Tarjan algorithm [HT74]) is used for the planarity testing performed in line 9. If  $\mathcal{G}'$  is planar, the algorithm on the current level of recursion returns the boolean value “TRUE.” The execution of the algorithm on the previous recursion level would receive a value of “TRUE” on line 4, prompting it to return “TRUE” on line 5. This process will continue until the recursion has “unrolled” into Algorithm 5.1, which itself will return a value of “TRUE,” indicating that a drawing with the given number of crossings was found. If no set of crossing permutations can

<sup>1</sup>Technically, the information in  $\mathbf{D}$  may be derived from  $\mathbf{L}$ , but doing so would obfuscate the functioning of the other algorithms.

---

**Algorithm 5.2** TestPlanar: The permutation enumeration algorithm

---

**Input:** A vector of edge sets  $\mathbf{L}$  which encodes the edge crossing set, a list of crossed edges  $\mathbf{D}$ , the graph  $\mathcal{G}$  of which  $\mathbf{L}$  is the crossing set, and the current edge  $e$  of which the order of edge crossings have to be permuted.

**Output:** TRUE if a set of permutations for the current edge crossings set was found, FALSE otherwise.

```

1:  $\mathbf{L}_e \leftarrow \text{FirstPermutation}(\mathbf{L}_e)$ 
2: loop
3:   if HasNextElement( $\mathbf{D}, e$ ) then
4:     if TestPlanar( $\mathbf{L}, \mathbf{D}, \text{NextElement}(\mathbf{D}, e), \mathcal{G}$ ) = TRUE then
5:       return TRUE
6:     end if
7:   else
8:      $\mathcal{G}' \leftarrow \text{ConstructGraph}(\mathbf{L}, \mathbf{D}, \mathcal{G})$ 
9:     if IsPlanar( $\mathcal{G}'$ ) then
10:      return TRUE
11:    end if
12:  end if
13:  if  $\mathbf{L}_e \neq \text{LastPermutation}(\mathbf{L}_e)$  then
14:     $\mathbf{L}_e \leftarrow \text{NextPermutation}(\mathbf{L}_e)$ 
15:  else
16:    return FALSE
17:  end if
18: end loop

```

---

**Implementation:** An implementation of an improved version of this algorithm, described later in this chapter, may be found in § B.4.3.

---

be found for which a planar graph  $\mathcal{G}'$  may be constructed, all permutations  $\mathbf{L}_e$  of crossings over  $e$  will eventually be enumerated. When  $\mathbf{L}_e$  contains the last permutation in the lexicographic order of crossings over  $e$ , the condition on line 13 will be false and the algorithm will terminate by returning a value “FALSE” to the previous recursion level, indicating that no planar graph  $\mathcal{G}'$  could be constructed. No “unrolling” process occurs when “FALSE” is returned, as occurs when “TRUE” is returned. The next permutation of crossings for the edge  $f$ , corresponding to the previous recursion level will simply be generated and the recursive process of examining all crossing permutations for each edge following  $f$  in  $\mathbf{D}$  will recommence. Of course, if no drawing of  $\mathcal{G}$  with the given number of crossings can be found, a value of “FALSE” will eventually be returned to Algorithm 5.1.

### 5.2.3 Description of the ConstructGraph algorithm.

In order to test whether a given crossing configuration is feasible, a graph must be constructed from  $\mathcal{G}$  such that crossings are substituted for artificial vertices of degree four. If this graph is planar, then a planar layout of this graph is a drawing of  $\mathcal{G}$  with  $k$  crossings (where the vertices representing crossings are again viewed simply as crossings).

The algorithm is very simple —  $k$  vertices for representing crossings are created, and associated with every edge pair. This is done in line 1 of the algorithm. Every edge  $e$  in  $\mathcal{G}$  that has any crossings over it must be replaced by a path for which the end vertices correspond to the incident vertices of  $e$ , and for which the interior vertices correspond to crossings in the order mandated

**Algorithm 5.3** ConstructGraph**Input:** A graph  $\mathcal{G}$ , a list of crossed edges  $\mathbf{D}$ , and the crossing configurations for the edges  $\mathbf{L}$ .**Output:** The graph  $\mathcal{G}$ , modified so that the crossings from  $\mathbf{D}$  are present as artificial vertices of degree four, in the order given for the edges in  $\mathbf{L}$ .

```

1:  $\phi \leftarrow$  a bijection of the edge pairs in  $\mathbf{D}$  to  $k$  new vertices in  $\mathcal{G}$ 
2: for all  $e = \{u, v\} \in \mathbf{D}$  do
3:    $x \leftarrow u$ 
4:   for all  $\{e, f\} \in \mathbf{L}_e$  do
5:      $w \leftarrow \phi(\{e, f\})$ 
6:      $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \cup \{\{x, w\}\}$ 
7:      $x \leftarrow w$ 
8:   end for
9:    $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \cup \{\{x, v\}\}$ 
10:   $E(\mathcal{G}) \leftarrow E(\mathcal{G}) \setminus \{e\}$ 
11: end for
12: return  $\mathcal{G}$ 

```

**Implementation:** An implementation of this algorithm may be found in § B.4.3.

by the current crossing configuration in  $\mathbf{L}_e$ . This process is performed for each edge  $e$  involved in crossings by the loop between lines 2 and 11. The variable  $x$  (line 3) keeps track of the latest vertex addition to the path that is to replace  $e$  in  $\mathcal{G}$ . On line 3,  $x$  is assigned one of the vertices incident to  $e$ ,  $u$  (say), which becomes the first vertex of the path. All crossings in which  $e$  is involved are looped over between lines 4 and 8 of the algorithm, and they are added to the path in the order that they appear. Line 5 is used to obtain a reference to the artificial vertex (which models a crossing) associated with the current edge pair (this vertex may already have some adjacent edges due to the fact that the edge pair  $\{e, f\}$  may already have been examined when the crossing list for  $f$  was examined). In line 6 of the algorithm, the path is extended, and  $x$  is made to point to the new path end. The last edge is added to the path in line 9, and the edge  $e$  is then removed in line 10. Finally, the new graph is returned in line 12 of the algorithm.

**5.2.4 Reducing the number of cases**

It should be clear that the Garey–Johnson algorithm would have a prohibitive running time, even for very small graphs. The first graph property that requires consideration in order to exploit any symmetry (so as to speed up the algorithm), is the degree of vertex–transitivity of a graph, or rather, the transitivity of its partite sets. It is assumed that adjacent edges may not cross.

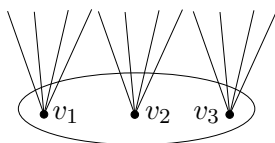
**5.2.4.1 Symmetry considerations for multipartite graphs**

Figure 5.4: The star graphs induced by the edges from  $v_1$ ,  $v_2$  and  $v_3$  are all isomorphic, and this symmetry allows for pruning the number of calculations in the Garey–Johnson algorithm.

Let  $\mathcal{P}$  be one of the partite sets of a complete multipartite graph  $\mathcal{G}$ . For a vertex  $v \in \mathcal{P}$ , the edges incident to  $v$  induce a star. If  $\mathcal{G}$  is a complete multipartite graph, the induced stars containing vertices in  $\mathcal{P}$ , are pairwise isomorphic. Now, consider Figure 5.4 — suppose that edges of  $v_1$  are crossed  $x$  times and that the edges of  $v_2$  are crossed  $y$  times. Since the two star graphs induced by the edges incident to each of the two respective vertices are isomorphic, the vertices may have their labels swapped, without any change in the structure of the crossing configuration. Therefore, it is safe to consider only the cases where  $x \geq y$ . This generalizes to any number of vertices in the same partite set, and it holds independently for every partite set.

When two or more *partite sets* have the same number of vertices, symmetry may be exploited in a manner analogous to the above argument. Let  $n$  be the size of each partite set, then denote by a tuple  $(x_1, x_2, \dots, x_n)$  the numbers of crossings in which the  $n$  respective induced stars in  $\mathcal{P}$  are involved. Now, let two partite sets  $\mathcal{P}_1, \mathcal{P}_2$  have crossings represented respectively by the sets  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$ . As before, a relabelling will result in a crossing configuration that is unchanged, and so it suffices to consider only the cases where  $(x_1, x_2, \dots, x_n) \geq (y_1, y_2, \dots, y_n)$ , where tuples are compared lexicographically. Again, this generalizes to any number of partite sets of the same size.

#### 5.2.4.2 Partial verification

In the basic implementation of the Garey–Johnson algorithm, all permutations of the orders of crossings on edges are considered. However, when the permutations of only the first few edges result in a non-planar configuration involving those edges, then there should be no need to enumerate the permutations of the remaining edges.

Suppose that the permutations of crossings across the first  $x$  edges in the list  $\mathbf{D}$  are enumerated, whilst the orders of crossings for the rest are kept constant. When line 9 of the `TestPlanar` algorithm is reached, then only the first  $x$  edges are constructed (the rest of the edges in the edge crossing set for which crossing orderings are kept constant are *not* included in any way) into the graph  $\mathcal{G}'$ . If it transpires that  $\mathcal{G}'$  is non-planar, it is then obvious that the permutations of the remaining edges need not be enumerated at all. Of course, if  $\mathcal{G}'$  is planar, then the permutations of the rest of the edges need to be considered, and because these edges were omitted with the construction of  $\mathcal{G}'$ , it could mean that for all of the permutation sets of the remaining edges, only non-planar configurations will result.

This scheme could be used to partition the search space. However, caution should be practised, since the extra work required to test for constructing such partial graphs, and for the planarity tests may become acute when it happens often that a partial graph is planar, whereas no permutation of its remaining edges would result in a planar graph.

#### 5.2.4.3 Independent crossing subgraphs

Define a new graph  $\mathcal{I}_{\mathbf{C}}$ , called the intersection graph of the crossing set  $\mathbf{C}$  (*i.e.*, a set of pairs of crossing edges). For each edge  $e_i$  in  $\mathcal{G}$ , there is a corresponding vertex  $v_i$  in  $\mathcal{I}_{\mathbf{C}}$ , and a pair of vertices  $v_i, v_j \in V(\mathcal{I}_{\mathbf{C}})$  are adjacent only if the edge pair  $\{e_i, e_j\}$  is present in  $\mathbf{C}$ .

An example of an intersection graph is shown in Figure 5.5(b) for the graph  $\mathcal{G}^*$  in Figure 5.5(a). Only the pairs of bold edges corresponding to crossings occur in the set  $\mathbf{C}$  of edge pairs. The labels for edges that are not involved in crossings, have been omitted to render the figure less cluttered.

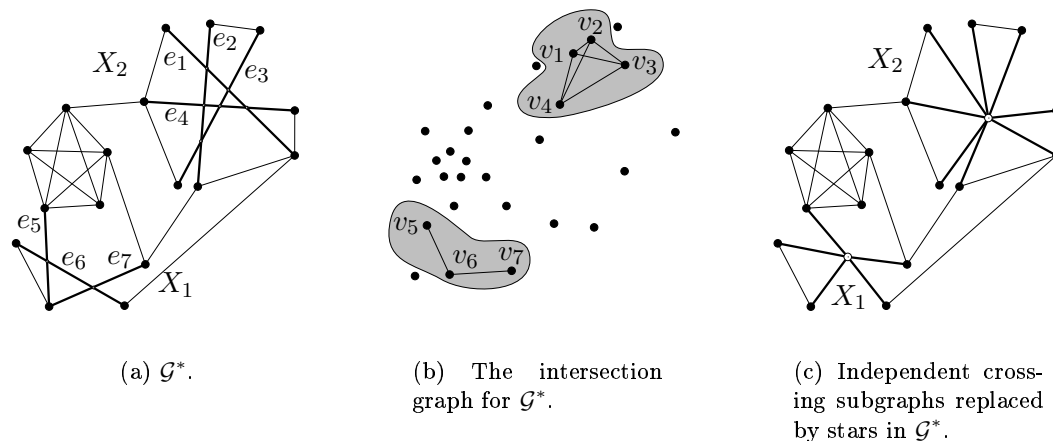


Figure 5.5: An illustration of how the intersection graph specifies the independent crossing subgraphs, and how these subgraphs may be replaced by stars.

The components of the graph  $\mathcal{I}_{\mathcal{C}}$ , that are not isolated vertices are called *intersection components*. For the graph  $\mathcal{G}^*$  in Figure 5.5(a), it may be seen that the corresponding intersection graph of  $\mathcal{G}^*$  in Figure 5.5(b) has two intersection components, indicated by the grey regions.

Each intersection component directly specifies a subgraph in the main graph, called an *independent crossing subgraph*. The vertices from an intersection component map to edges in the main graph which edge-induce an independent crossing subgraph (in the main graph). The two subgraphs  $X_1$  and  $X_2$  of  $\mathcal{G}^*$  in Figure 5.5(a) that are comprised of bold edges, were formed from the intersection graph of  $\mathcal{G}^*$ , in Figure 5.5(b), by this method.

In the Garey–Johnson algorithm, a crossing set must be found that permits the appropriate permutations of crossings in each of the resulting independent crossing subgraphs to be determined, so that a planar graph, containing artificial vertices representing crossings, may be constructed. It is possible that, regardless of whether each independent crossing subgraph may be planar (when crossings are modelled as artificial vertices of degree four), the input graph in its entirety may be non-planar.

To see how this may be possible, let all the artificial vertices (representing crossings) in an independent crossing subgraph  $\mathcal{H}$  be replaced by a single artificial vertex  $v$ , which is joined to every remaining (non-artificial) vertex in  $\mathcal{H}$ . One may think of the artificial vertices as being “compressed” into  $v$ , so that  $v$  may be thought of as a “black box” that renders the details of the permutations of crossings within the independent crossing subgraph hidden. The edges adjacent to the vertex  $v$  induce a star. When an independent crossing subgraph is replaced by a star, it is said to be *contracted*, otherwise it is said to be *expanded*. In Figure 5.5(c), both independent crossing subgraphs,  $X_1$  and  $X_2$ , of the graph  $\mathcal{G}^*$  in Figure 5.5(a) have been contracted, where the white vertices are the single artificial vertices replacing the artificial vertices of  $X_1$  and  $X_2$ , respectively.

If each independent crossing subgraph in the input graph  $\mathcal{G}$  is contracted, the graph  $\mathcal{G}'$ , resulting from the contractions, must be planar if the crossing set  $C$  that yielded  $\mathcal{G}'$  is to permit a planar graph  $\mathcal{G}''$ , with artificial vertices representing crossings, to be constructed from  $\mathcal{G}$  (*i.e.*, all independent crossing subgraphs are expanded). Conversely, if  $\mathcal{G}'$  is non-planar, the crossing set  $C$  cannot possibly permit the construction of such a planar graph  $\mathcal{G}''$ . Thus, the non-planarity of  $\mathcal{G}'$  obviates the necessity of examining the permutations of crossings within the independent crossing subgraphs of  $\mathcal{G}$ . As an example, the graph in Figure 5.5(c), derived from  $\mathcal{G}^*$  by the con-

traction of all independent crossing subgraphs corresponding to the crossing set  $C$ , is non-planar, due to the presence of the subgraph that is isomorphic to  $\mathcal{K}_5$ .

This technique may be used in conjunction with partial verification (described in the previous section). For a given crossing set  $C$ , all independent crossing subgraphs of a graph  $\mathcal{G}$  are initially contracted. If  $\mathcal{G}$  is non-planar in this state, then no independent crossing subgraphs need be expanded (since  $\mathcal{G}$  will be non-planar regardless of the permutations of crossing within any independent crossing subgraph). However, if  $\mathcal{G}$  is planar, an independent crossing subgraph  $\mathcal{H}_1$  of  $\mathcal{G}$  must be expanded, and all permutations of crossings within  $\mathcal{H}_1$  enumerated. If any of these configurations permit the construction of a planar graph  $\mathcal{G}'$  from  $\mathcal{G}$ , another crossing subgraph  $\mathcal{H}_2$  must be expanded, and all permutations within  $\mathcal{H}_2$  must be examined and so forth. If, for some crossing subgraph  $\mathcal{H}_i$ , no set of permutations of crossings of the edges within  $\mathcal{H}_i$  permit the construction of a planar graph  $\mathcal{G}''$  from  $\mathcal{G}$ ,  $\mathcal{H}_i$  is contracted, and the remainder of the crossing permutations within the independent crossing subgraph  $\mathcal{H}_{i-1}$ , which was expanded prior to  $\mathcal{H}_i$ , are enumerated. Again, if any of these configurations permit the construction of a planar graph from  $\mathcal{G}$ ,  $\mathcal{H}_i$  is again expanded, and so on.

Thus, the recursive expansion and contraction of independent crossing subgraphs achieves the goal of partial verification, although this scheme has the additional advantage that edges for which crossing permutations are not enumerated, are not omitted, but the subgraphs in which these edges appear, are replaced by stars. This is likely to indicate at an earlier stage that a given crossing configuration of a graph cannot lead to a feasible drawing for  $\mathcal{G}$  with the given number of crossings, due to the fact that the graph  $\mathcal{G}$  contains more edges at any stage, than with the application of pure partial verification.

The Garey–Johnson algorithm is only required to be modified slightly, in order to implement the concept of partial verification combined with that of independent crossing subgraphs. The independent crossing subgraphs partition the set of crossed edges,  $\mathbf{D}$ , that is used in Algorithms 5.1–5.3. In the modification, instead of the set  $\mathbf{D}$  being examined in its entirety, the edges of the most recently expanded independent crossing subgraph are examined in isolation of the edges in other independent crossing subgraphs. As all independent crossing subgraphs are expanded recursively, all of the crossing edges which constitute  $\mathbf{D}$  are eventually examined.

### Modifying Algorithm 5.1 (GareyJohnson)

The Garey–Johnson algorithm has to be modified to compute the independent crossing subgraphs for a given crossing configuration before commencing the enumeration of the possible crossings in the crossing configuration. This must occur in Algorithm 5.1 (GareyJohnson), which is appropriately modified to produce Algorithm 5.4 (GareyJohnson'). In this algorithm, the set  $\mathbf{D}$  is removed, and therefore also the code for its initialization from the loop now only spanning lines 5–8 (in Algorithm 5.4). The independent crossing subgraphs are constructed for the input graph  $\mathcal{G}$ , from the crossing configuration  $C$ , by the routine called in line 9. No explicit algorithm for the computation of the independent crossing subgraphs is mentioned here, but the independent crossing subgraphs may be obtained by selecting the components from the intersection graph of  $\mathcal{G}$ , that are not isolated vertices. Initially, all independent crossing subgraphs are contracted (line 10), and  $\mathcal{G}$  (or more correctly, the subgraph that derives from  $\mathcal{G}$  by the contraction of the independent crossing subgraphs) is tested for planarity by the conditional structure spanning lines 11–17. If  $\mathcal{G}$  is non-planar, the crossing configuration  $C$  will always lead to a non-planar configuration, and therefore, no set of permutations of crossings in  $C$  will produce a feasible drawing of  $\mathcal{G}$  with the given number of crossings (where crossings are modelled as artificial vertices of degree four). If, however,  $\mathcal{G}$  is planar, a crossing subgraph is expanded, and Algorithm 5.5



---

**Algorithm 5.4** *GareyJohnson'*: The revised Garey–Johnson algorithm
 

---

**Input:** A graph  $\mathcal{G}$ .**Output:** The crossing number  $\nu(\mathcal{G})$  of  $\mathcal{G}$ , and a drawing realising  $\nu(\mathcal{G})$  where crossings are modelled as artificial vertices of degree 4.

```

1: for all  $C \in \{(p_1, p_2, \dots, p_k) : (p_1, p_2, \dots, p_k) \text{ is a set of } k \text{ pairs of edges of } \mathcal{G}\}$  do
2:   for all  $e \in E(\mathcal{G})$  do
3:      $\mathbf{L}_e \leftarrow \emptyset$ 
4:   end for
5:   for all  $(e, f) \in C$  do
6:      $\mathbf{L}_e \leftarrow \mathbf{L}_e \cup \{f\}$ 
7:      $\mathbf{L}_f \leftarrow \mathbf{L}_f \cup \{e\}$ 
8:   end for
9:   ConstructCrossingSubgraphs( $\mathcal{G}, p$ )
10:  ContractAllCrossingSubgraphs( $\mathcal{G}$ )
11:  if IsPlanar( $\mathcal{G}$ ) then
12:     $\mathbf{D} \leftarrow \text{FirstCrossingSubgraph}(\mathcal{G})$ 
13:    ExpandCrossingSubgraph( $\mathcal{G}, \mathbf{D}$ )
14:    if TestPlanar'( $\mathbf{L}, \mathbf{D}, \text{FirstElement}(\mathbf{D}), \mathcal{G}$ ) then
15:      return TRUE
16:    end if
17:  end if
18: end for
19: return FALSE

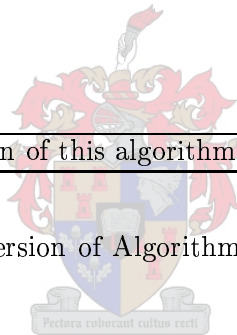
```

---

**Implementation:** An implementation of this algorithm may be given in § B.4.1.
 

---

(*TestPlanar'*), which is a modified version of Algorithm 5.2 (*TestPlanar*), is executed with  $\mathcal{G}$  as input.



### Modifying Algorithm 5.2 (*TestPlanar*)

All changes to Algorithm 5.2, occur in the “else” part of the conditional structure that starts at line 3, from line 10 onwards. A graph  $\mathcal{G}'$  with artificial vertices representing crossings is constructed on line 8, as in Algorithm 5.2. The difference (with respect to Algorithm 5.2) is that the planarity of this graph cannot, of course, imply the feasibility of a drawing of  $\mathcal{G}$  with the given number of crossings, unless all independent crossing subgraphs have been expanded. Therefore, a test is performed on line 10 to determine whether  $\mathcal{G}$  contains any contracted independent crossing subgraphs. If it does not, then  $\mathcal{G}'$  is indeed a feasible drawing with the given number of crossings, and the algorithm returns the value “TRUE” on line 19 to signify this condition. On the other hand, if contracted independent crossing subgraphs remain, one of them must be selected (line 11) and expanded (line 12). Algorithm 5.5 (*TestPlanar'*) is invoked on line 13 by the conditional structure that spans up to line 17. If the call returns the value “TRUE,” then it means that the recursive process ended with all crossing subgraphs expanded and that this led to a planar configuration of the graph constructed from  $\mathcal{G}$  (with artificial vertices for crossings). Therefore, this invocation of the *TestPlanar'* algorithm returns “TRUE” in its turn, initiating an recursion “unrolling” process, as described for Algorithm 5.2. However, if the call returns “FALSE,” then all permutations of crossings in some independent crossing subgraph expanded later in the recursion led to non-planar configurations. The crossing subgraph that was expanded is therefore contracted on line 16, so that the next set of permutations for crossings in the current

---

**Algorithm 5.5** *TestPlanar'*: The revised permutation enumeration algorithm

---

**Input:** A vector of edge sets  $\mathbf{L}$  which encodes the edge crossing set, a list of crossed edges  $\mathbf{D}$ , the graph  $\mathcal{G}$  of which  $\mathbf{L}$  is the crossing set, and the current edge  $e$  of which the order of edge crossings have to be permuted.

**Output:** TRUE if a set of permutations for the current edge crossings set was found, FALSE otherwise.

```

1:  $\mathbf{L}_e \leftarrow \text{FirstPermutation}(\mathbf{L}_e)$ 
2: loop
3:   if HasNextElement( $\mathbf{D}, e$ ) then
4:     if TestPlanar'( $\mathbf{L}, \mathbf{D}, \text{NextElement}(\mathbf{D}, e), \mathcal{G}$ ) = TRUE then
5:       return TRUE
6:     end if
7:   else
8:      $\mathcal{G}' \leftarrow \text{ConstructGraph}(\mathbf{L}, \mathbf{D}, \mathcal{G})$ 
9:     if IsPlanar( $\mathcal{G}'$ ) then
10:      if HasContractedCrossingSubgraph( $\mathcal{G}'$ ) then
11:         $\mathbf{D}' \leftarrow \text{NextCrossingSubgraph}(\mathcal{G}'$ )
12:        ExpandCrossingSubgraph( $\mathcal{G}', \mathbf{D}'$ )
13:        if TestPlanar'( $\mathbf{L}, \mathbf{D}', \text{FirstElement}(\mathbf{D}'), \mathcal{G}'$ ) then
14:          return TRUE
15:        else
16:          ContractCrossingSubgraph( $\mathcal{G}', \mathbf{D}'$ )
17:        end if
18:      else
19:        return TRUE
20:      end if
21:    end if
22:  end if
23:  if  $\mathbf{L}_e \neq \text{LastPermutation}(\mathbf{L}_e)$  then
24:     $\mathbf{L}_e \leftarrow \text{NextPermutation}(\mathbf{L}_e)$ 
25:  else
26:    return FALSE
27:  end if
28: end loop

```

---

**Implementation:** An implementation of this algorithm may be found in § B.4.3.

---

independent crossing subgraph may be considered.

### 5.2.5 Example execution of the Garey–Johnson algorithm

As an example of the execution of the algorithm, a single crossing configuration for the graph  $\mathcal{K}_6$  is examined. For this example, the Garey–Johnson algorithm has to determine whether  $\mathcal{K}_6$  can be drawn with five crossings. The concept of independent crossing subgraphs, combined with partial verification, as described in the previous section, is employed in the example.

The example commences with an execution of Algorithm 5.4, where the set of edge crossing pairs  $C = \{\{\{v_0, v_1\}, \{v_0, v_2\}\}, \{\{v_0, v_1\}, \{v_0, v_3\}\}, \{\{v_0, v_1\}, \{v_2, v_3\}\}, \{\{v_0, v_2\}, \{v_3, v_4\}\}, \{\{v_1, v_4\}, \{v_2, v_5\}\}\}$  is selected in line 1. After the execution of lines 5–8, the contents of  $\mathbf{L}$  is

$$\begin{aligned} \mathbf{L}_{\{v_0, v_1\}} &= \{\{v_0, v_2\}, \{v_0, v_3\}, \{v_2, v_3\}\}, & \mathbf{L}_{\{v_0, v_2\}} &= \{\{v_0, v_1\}, \{v_3, v_4\}\}, & \mathbf{L}_{\{v_0, v_3\}} &= \{\{v_0, v_1\}\}, \\ \mathbf{L}_{\{v_1, v_4\}} &= \{\{v_2, v_5\}\}, & \mathbf{L}_{\{v_2, v_3\}} &= \{\{v_0, v_1\}\}, & \mathbf{L}_{\{v_2, v_5\}} &= \{\{v_0, v_2\}\}, \\ \mathbf{L}_{\{v_3, v_4\}} &= \{\{v_0, v_2\}\}. \end{aligned}$$

There are two independent crossing subgraphs in this choice of crossings. The first subgraph,  $\mathcal{H}_1$  contains the set of edges  $E(\mathcal{H}_1) = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}\}$ , whilst the second,  $\mathcal{H}_2$  contains the set of edges  $E(\mathcal{H}_2) = \{\{v_1, v_4\}, \{v_2, v_5\}\}$ . The subgraph  $\mathcal{H}_1$  is drawn in Figure 5.6(a), with only the crossings mandated by the crossing configuration  $C$ , whilst  $\mathcal{H}_2$  is drawn in Figure 5.6(b), containing the single crossing required by  $C$ .



(a) Crossing subgraph with edges  $e_0, e_1, e_2, e_9$  and  $e_{12}$ . (b) Crossing subgraph with edges  $e_1$  and  $e_7$ .

Figure 5.6: Two independent crossing subgraphs corresponding to the choice of edge crossings  $\{\{v_0, v_1\}, \{v_0, v_2\}\}, \{\{v_0, v_1\}, \{v_0, v_3\}\}, \{\{v_0, v_1\}, \{v_2, v_3\}\}, \{\{v_0, v_2\}, \{v_3, v_4\}\}, \{\{v_1, v_4\}, \{v_2, v_5\}\}$  from  $\mathcal{K}_6$ .

When the algorithm commences, all independent crossing subgraphs are contracted. The input graph,  $\mathcal{G}$ , in this state, is shown in Figure 5.7(a), where the artificial vertices of the contracted independent crossing subgraphs are labelled according to the labels of expanded independent crossing subgraphs they represent. Clearly this graph is planar, and an independent crossing subgraph must therefore be expanded. The subgraph  $\mathcal{H}_1$  is selected for this purpose, and Algorithm 5.5 (`TestPlanar'`) is called where  $\mathbf{D} = \{\{v_0, v_2\}, \{v_0, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_0, v_1\}\}$ . The order in which elements from  $\mathbf{D}$  are selected by Algorithm 5.5 is arbitrary, but it must remain fixed for the duration of the algorithm. Suppose that the order is given as  $\{\{v_0, v_2\}, \{v_0, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_0, v_1\}\}$ .

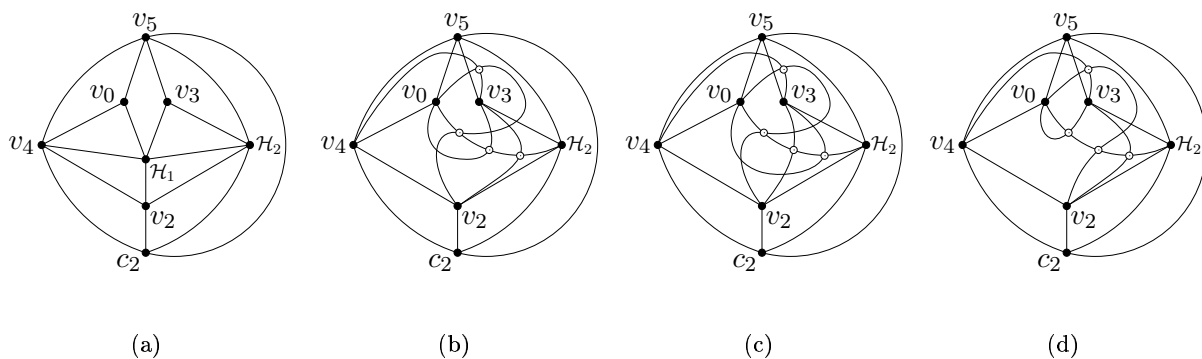


Figure 5.7: Steps in the execution of the Garey–Johnson algorithm for the example.

The edge  $\{v_0, v_2\}$  is the first edge of  $\mathcal{H}_1$  to be considered by Algorithm 5.5. This edge is crossed by the two edges  $\{v_0, v_1\}, \{v_3, v_4\}$  and suppose that this order is the order generated by the call to the function `FirstPermutation` on line 1 of Algorithm 5.5. The test `HasNextElement` performed on line 3 evaluates to “TRUE,” since  $\mathbf{D}$  contains four other edges besides  $\{v_0, v_2\}$ . Algorithm 5.5 is invoked recursively (line 4), and the edge  $\{v_0, v_3\}$  is the next examined edge.

The algorithm again determines that edges remain in  $\mathbf{D}$ , and so it calls itself recursively again. This process repeats itself until the last edge,  $\{v_0, v_1\}$ , in  $\mathbf{D}$  is considered. This edge is crossed by the three edges  $\{v_0, v_2\}$ ,  $\{v_0, v_3\}$ , and  $\{v_2, v_3\}$  and suppose that this order is generated by the call to `FirstPermutation` on line 1 of Algorithm 5.5. The test `HasNextElement` evaluates to “FALSE” in this case, since there are no remaining edges in the crossing set  $\mathbf{D}$  to examine. The “else” part of the conditional structure spanning lines 3–22, is now executed. A graph  $\mathcal{G}'$  containing artificial vertices of degree four, modelling the crossings corresponding to the current permutations of edge crossings, is constructed on line 8. This graph is shown in Figure 5.7(b), where the artificial vertices are coloured white. This graph is non-planar, which causes the test on line 9 to evaluate to “FALSE.” The next line of code to be executed is line 23, which evaluates to “TRUE,” since not all permutations of crossings over the edge  $\{v_0, v_1\}$  have yet been considered. Thus, line 24 is executed, generating the next permutation order of edge crossings over the edge  $\{v_0, v_1\}$ , which is  $\{\{v_0, v_2\}, \{v_2, v_3\}, \{v_0, v_3\}\}$ . The loop spanning lines 2–28 is repeated and again a graph  $\mathcal{G}'$  with artificial vertices modelling crossings is constructed on line 8. This graph is shown in Figure 5.7(c). The graph  $\mathcal{G}'$  is non-planar and execution moves to line 23. Permutations for crossings over  $\{v_0, v_1\}$  remain, causing the next permutation order  $\{\{v_0, v_3\}, \{v_0, v_2\}, \{v_2, v_3\}\}$  to be generated on line 24. The graph  $\mathcal{G}'$  corresponding to this permutation is shown in Figure 5.7(d). This process continues for all permutation orders of crossings over the edge  $\{v_0, v_1\}$  and it can be shown that no such permutation leads to a planar graph  $\mathcal{G}'$ .

If, however, there had been a configuration for which the constructed graph  $\mathcal{G}'$  had been planar, it would have been determined on line 10 that  $\mathcal{G}'$  contained a contracted crossing subgraph, namely  $\mathcal{H}_2$ . This independent crossing subgraph would have been expanded, and Algorithm 5.5 would have been invoked recursively with the edges of  $\mathcal{H}_2$  as argument. The same operations as described above would have been performed for the edges of  $\mathcal{H}_2$ . If no set of permutations of crossings in  $\mathcal{H}_2$  yielded a planar configuration,  $\mathcal{H}_2$  would have been contracted again on line 16, so that the enumeration of crossing configurations in  $\mathcal{H}_1$  could be continued.

When all crossing permutation possibilities for the edge  $\{v_0, v_1\}$  have been exhausted, the Algorithm 5.5 returns “FALSE” to the previous recursion corresponding to the edge  $\{v_3, v_4\}$ . This causes the test on line 4 to evaluate to “FALSE,” and the next line of code to be executed is thus line 23, which evaluates to “TRUE,” since  $\{v_3, v_4\}$  contains one more permutation of crossings. The next permutation of crossings over  $\{v_3, v_4\}$  is generated, and when the loop repeats, the algorithm will again call itself recursively on line 4, and enumerate all crossing permutations for the edge  $\{v_0, v_1\}$ . Since the edge  $\{v_3, v_4\}$  is crossed by only two edges, it only has two crossing permutations. After both of these permutations have been examined, the algorithm will fall back one recursion level to the edge  $\{v_0, v_2\}$ . This edge is only crossed by a single edge, and its only permutation has therefore already been examined. This causes the algorithm to fall back yet another level in the recursion. In fact, both the edges  $\{v_0, v_3\}$  and  $\{v_2, v_3\}$  are only crossed once each, and the recursion therefore “unwinds” back into Algorithm 5.4, from whence the next crossing configuration is generated. This whole process is then repeated, if such a next crossing configuration exists.

### 5.3 All drawings may be transformed to two-page layouts

When an arrangement for the vertices on the spine are chosen via recursive graph bisection (§ 4.3.2.1) in a two-page combinatorial book layout, one is guaranteed (due to the work of

Shahrokhi, Székely, Sýkora, and Vrto [SSSV96b]) of the bound

$$x \leq O\left([\log |V(\mathcal{G})|]^2(\nu(\mathcal{G}) + \sum_{v \in V(\mathcal{G})} [\deg_{\mathcal{G}} v]^2)\right),$$

where  $x$  is the number of crossings that would result if  $\mathcal{G}$  is drawn constructing its bisected components as shown in Figure 4.25. However, a large constant may be hidden by the  $O$  notation. It is shown in this section that although this may be the case, there exists a subdivision  $\mathcal{H}$  of  $\mathcal{G}$  for which there exists a vertex arrangement, and for which an edge layout may be found that renders the drawing obtained from this configuration is in single-cross normal form, with  $\nu(\mathcal{G})$  crossings.

This result on its own is quite useless, due to the **NP**-complete nature of the problem (§ 4.3.2.2). What it does, however, suggest, is that one may start with an initial solution generated by recursive graph bisection, after which the graph could be subdivided, and the solution potentially improved by the application of other optimization techniques. The construction of the proof of the following theorem shows how a subdivision of a graph on a book may be achieved without increasing the number of crossings.

**Theorem 5.3.1** *For a graph  $\mathcal{G}$  and a subdivision  $\mathcal{H}$  of  $\mathcal{G}$ ,  $\nu_{B_2}(\mathcal{H}) \leq \nu_{B_2}(\mathcal{G})$ .*

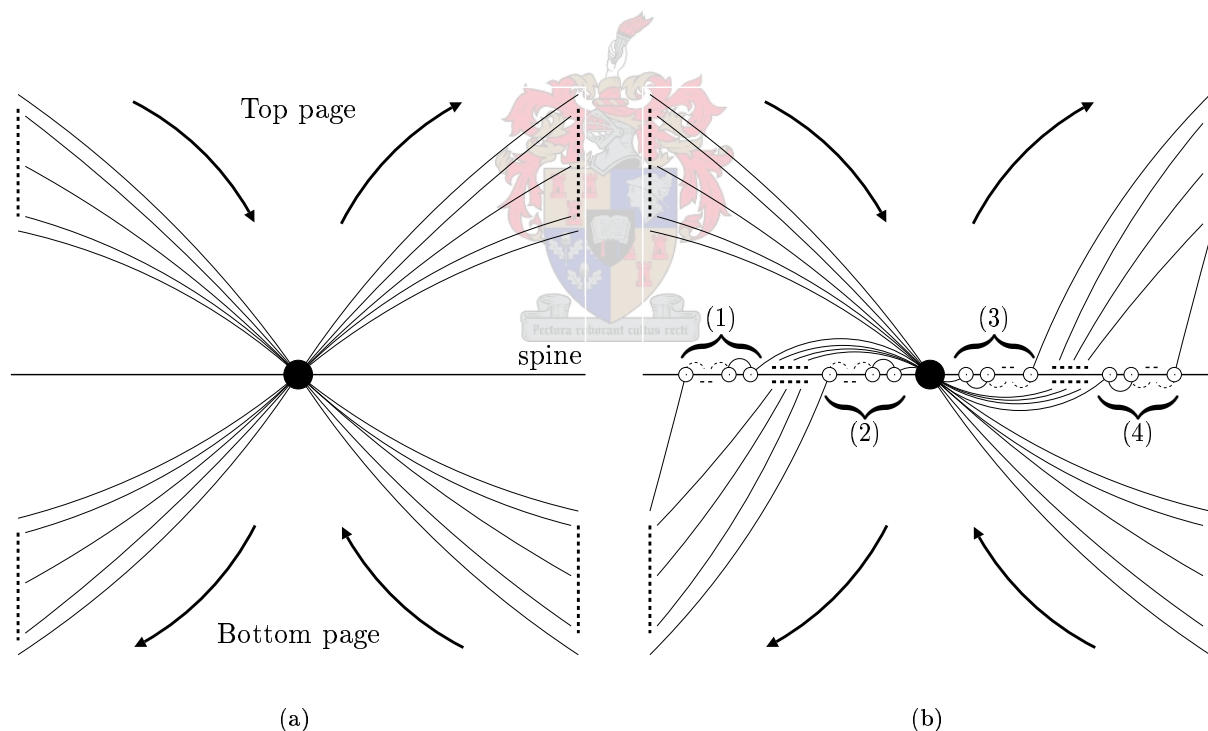


Figure 5.8: The insertion of subdivision vertices around a vertex.

**Proof:** The first step is to demonstrate a mechanical method of constructing a two-page layout  $\phi'$  of  $\mathcal{H}$  from the layout  $\phi$  of  $\mathcal{G}$ , such that  $\nu_{\phi'}(\mathcal{H}) \leq \nu_{\phi}(\mathcal{G})$ . For the purposes of this part of the proof, consider the edges of  $\mathcal{G}$  to be directed. Let the spine of the book run horizontally, so that one page is at the top, and the other at the bottom when the book is completely folded open.

All arcs on the upper page are considered directed from left to right — that is, their source vertices have smaller indices than their target vertices. All arcs on the lower page are considered directed from right to left — their source vertices have larger indices than their target vertices.

This is depicted in Figure 5.8(a) (individual arrow tips are not drawn, but the lines are directed as shown by the arrows).

An arc  $(v_i, v_d)$ ,  $v_i < v_d$  is said to be higher than an arc  $(v_i, v_c)$ , if  $v_i < v_c < v_d$  — this choice of terminology reflects the fact that the drawing of  $(v_i, v_d)$  passes over the drawing of  $(v_i, v_c)$ , since both must be on the upper page. In a similar fashion, an arc  $(v_i, v_a)$ ,  $v_i > v_a$ , is said to be lower than an arc  $(v_i, v_b)$  if  $v_a < v_b < v_i$ , since both must be on the lower page.

Let a particular vertex, say  $v_i$ , be isolated — a procedure for placing subdivision vertices around  $v_i$  with the property that no additional crossings are caused, will be applicable to all the other vertices. All subdivision vertices of edges in  $\phi'$  corresponding to arcs in  $\phi$ , with  $v_i$  as the source, are to be placed next to  $v_i$  in  $\phi'$  according to the following method:

Suppose each arc in  $\phi$  is to be subdivided  $s$  times, and let a subdivision vertex of an edge  $e_h$  (in  $\phi'$ ) be denoted  $v_{h_i}$ , where  $i$  is the index of  $v_{h_i}$  in the list of subdivision vertices. For all arcs on the upper page (*i.e.*, those joining  $v_i$  to vertices occurring further to the right of  $v_i$  on the spine in  $\phi$ ), let the subdivision vertices  $\{v_{h_{11}}, v_{h_{12}}, \dots, v_{h_{1s}}\}$  of the corresponding subdivided edge  $e_{h_1} = (v_i, v_d)$  be placed to the right of  $v_i$  (in  $\phi'$ ). Next, let the subdivision vertices  $\{v_{h_{21}}, v_{h_{22}}, \dots, v_{h_{2s}}\}$  of the next highest arc, corresponding to  $e_{h_2}$ , be placed to the right of the subdivision vertices of  $e_{h_1}$ , and so on. The subdivision edge  $(v_{h_{11}}, v_i)$  is placed on the lower page, as well as the subdivision edge  $(v_{h_{21}}, v_i)$ , and so on, so that all edges joining  $v_i$  and the first subdivision vertices of the various incident edges occur on the lower page. None of the subdivision edges can cross any other subdivision edges, since the page crossing condition (§ 3.1.3.1) is not met. Next, let the subdivision edges  $(v_{h_{1k}}, v_{h_{1k+1}})$ ,  $k < s$ , be added to the upper page. Finally  $(v_{h_{1s}}, v_j)$  is drawn on the upper page. The same is done for the subdivided edge  $e_{h_2}$  as well as for the rest of the subdivided edges, leaving  $v_i$  on the upper page.

The situation is completely symmetrical for the arcs on the lower page — let “right” be replaced by “left,” “upper” by “lower,” “first” by “last” and “highest” by “lowest.”

The method described above is illustrated graphically in Figure 5.8(b). At (1) the subdivision vertices of the arcs (around the given vertex) which is the highest amongst the low arcs are placed, (2) is the lowest arc, (3) is the highest arc, and (4) represents the placement of the lowest of the high arcs. Between (1) and (2) there may be any number of sets of subdivision vertices belonging to other arcs, and same is true between (3) and (4).

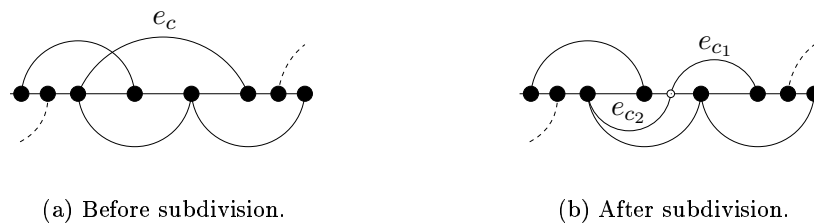


Figure 5.9: Subdividing a graph might lower its book crossing number.

Yannakakis showed in [Yan86] that four pages are necessary and sufficient for combinatorial book embeddings of planar graphs so that no crossings occur. Therefore, there exists a planar graph  $\mathcal{G}$  for which  $\nu_{B_2}(\mathcal{G}) > 0$ , but for which  $\nu_{B_2}(\mathcal{H}) = 0$ , where  $\mathcal{H}$  is a subgraph of  $\mathcal{G}$ .

An example of a configuration where the number of crossings is certainly decreased by the given subdivision is shown in Figure 5.9. In Figure 5.9(a), the edge  $e_c$  would be involved in a crossing, whether on the upper or the lower page. The subdivision of  $e_c$  into  $e_{c_1}$  and  $e_{c_2}$  in Figure 5.9(b) clearly removes this possibility. ■

The following theorem shows that all planar graphs may be drawn as rectilinear drawings. This rigid drawing structure makes it a simple to generate drawings in which all vertices are placed at distinct points on a straight line, such as, for example, a book drawing.

**Lemma 5.3.1** *For every planar graph  $\mathcal{G}$ , a straight-lined drawing  $\phi$  of  $\mathcal{G}$  in  $\mathbb{R}^2$  exists, such that the  $x$ -coordinates of the points representing vertices are all distinct.*

**Proof:** It follows from Fary’s theorem [Far48] that any planar graph  $\mathcal{G}$  may be drawn in the plane using straight lines such that no line crossings are present. Let the remaining pairs of vertices in the drawing that are not joined by edges be joined by straight lines. There is a finite number of straight lines and consequently there exists a gradient  $g$  that is not equal to the gradients of any of the lines. Let  $g$  be chosen so that the gradient that is perpendicular to it is not equal to any of the gradients of the lines and let the  $x$ -axis be chosen so that its gradient is perpendicular to  $g$ . Then the  $x$ -coordinates of the vertices of  $\mathcal{G}$  are all distinct. ■

Now it is possible to give the main result, which justifies the emphasis that this thesis places on the use of two-page algorithms for finding upper bounds to the crossing numbers of graphs.

**Theorem 5.3.2** *For a graph  $\mathcal{G}$ , there exists a subdivision  $\mathcal{H}$  of  $\mathcal{G}$ , such that a two-page combinatorial book embedding  $\phi$  of  $\mathcal{H}$  exists, satisfying  $\nu_\phi(\mathcal{H}) = \nu(\mathcal{G})$ . Furthermore, if an edge in  $\mathcal{G}$  cannot be crossed more than  $t$  times, then each edge has to be subdivided at most  $(t+1)(|V(\mathcal{G})|-2)$  times.*

**Proof:** Let  $\phi'$  be a drawing of  $\mathcal{G}$ , such that  $\nu_{\phi'}(\mathcal{G}) = \nu(\mathcal{G})$ . By inserting vertices of degree four where there are crossings in  $\phi'$ , a planar graph is obtained.

Using Lemma 5.3.1, a straight line drawing  $\phi'$  of  $\mathcal{G}$  in  $\mathbb{R}^2$  may be obtained so that the  $x$ -coordinate of each vertex is unique. For some vertices, there will be edges passing below, and/or above them, where the concepts “below” and “above” refer to the  $y$  coordinates of the vertices. These vertices are said to be “above” and “below” the edges respectively.

Now, set the  $y$  coordinates of all the vertices to 0, effectively projecting all vertices onto the  $x$ -axis. Any edges that were below a vertex  $v$  must pass under  $v$ , and must thus be drawn below the  $x$ -axis where they pass  $v$ . Likewise, edges that were above  $v$  must be drawn above the  $x$ -axis where they pass  $v$ .

All edges must now either be drawn above the  $x$ -axis, below it, or they must be “woven” over and under vertices (see Figures 5.10(c) and (d)). Where an edge passes over the  $x$ -axis, between vertices, a subdivision vertex is inserted, subdividing the edge.

This is done for every edge. Note that no additional crossings are introduced, nor are any removed. Thus, the final subdivided graph  $\mathcal{H}$  has the same number of crossings as  $\mathcal{G}$ . The only difference is that  $\mathcal{H}$  may be considered from a combinatorial book layout perspective.

Each edge can only be “woven” between a maximum of  $|V(\mathcal{G})| - 2$  vertices (since it does not need to do so for its own incident vertices). Any edge in  $\mathcal{G}$  that is involved in  $x$  crossings would be subdivided into  $x + 1$  edges due to the insertion of vertices for crossings. Thus, an edge in  $\mathcal{G}$  would potentially “weave” a maximum of  $(t + 1)(|V(\mathcal{G})| - 2)$  times, and would therefore need no more subdivisions in order to be drawn on a two-page book. ■

The projection action technique of Theorem 5.3.2 may be seen in Figure 5.10. In part (a) of the figure, there is only a single vertex,  $u_1$  above the edge  $e$ , and thus in the projection,  $e$  need only

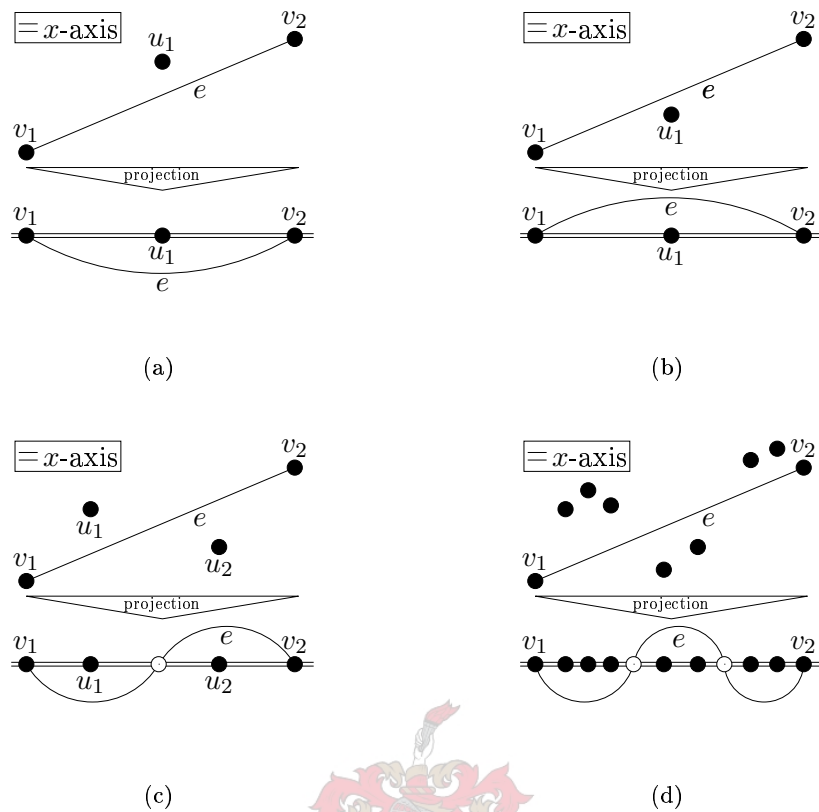


Figure 5.10: Projection of a straight line drawing of a graph  $\mathcal{G}$  onto the  $x$ -axis. Vertices of  $\mathcal{G}$  are denoted by black points, while subdivision vertices are denoted by white points.

pass under  $u_1$ . In part (b) of the figure the situation is reversed. When it happens that  $e$  has some vertices above and below it, as in part (c) of the figure, it must be subdivided to allow the subdivided sections to pass under and over  $u_1$  and  $u_2$  respectively. When groupings of vertices appear consecutively with respect to their  $x$ -coordinates, on one side of  $e$ , then an edge need only “stretch” over the whole grouping, as is illustrated in part (d) of the figure.

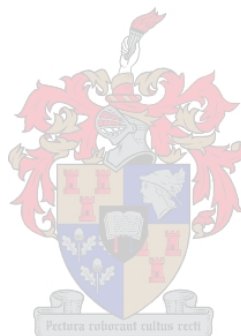
Theorems 5.3.1 and 5.3.2 together imply that the two–page crossing number of a graph approaches its plane crossing number, as the edges are increasingly subdivided. An important question to pursue, is how many edge subdivisions are required to guarantee the existence of a two–page layout of the graph in question, that will realise its crossing number. This, however, is outside the scope of this thesis.

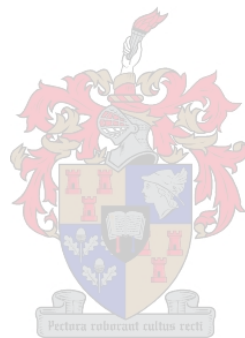
An implementation of an algorithm that employs the subdivisions, as described above, may be found in § B.5.2. It should be noted that graph subdivision raises the possibility that two “parent” edges (*i.e.*, edges that are subdivided) may cross one another more than once, since the subdivision edges are treated as distinct edges in a book layout. If it is important that a drawing be in single–cross normal form, the method for the removal of multiple crossings between a pair of edges, described in § 3.1.2 may be used for this purpose. Alternatively, an algorithm that generates book layouts may be programmed to avoid introducing multiple crossings of this kind.



## 5.4 Chapter summary

The application of the edge partitioning method was demonstrated in § 5.1 with the proof that the crossing number of  $\mathcal{K}_{1,1,1,1,n}$  satisfies  $\nu(\mathcal{K}_{1,1,1,1,n}) = \nu(\mathcal{K}_{4,n}) + n$ . A concrete implementation of the Garey–Johnson algorithm was developed in § 5.2, and it was shown how certain types of symmetry information may be exploited to reduce the total number of operations required to be performed by the algorithm. It was shown in § 5.3 that every graph  $\mathcal{G}$  has a subdivision for which a book embedding exists that realises the crossing number of  $\mathcal{G}$ . Furthermore, a mechanical method for subdividing a graph in a book was demonstrated, to prove that superfluous subdivisions need never increase the total number of crossings in a book layout.





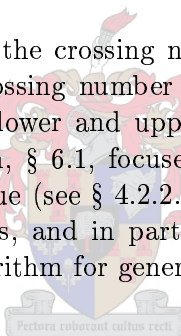
## Chapter 6

# Heuristic methods and novel results

One of the symptoms of an approaching nervous breakdown  
is the belief that one's work is terribly important

— *Bertrand Russell (1872–1970)*

The high computational complexity of the crossing number problem prompts one to consider heuristic methods for bounding the crossing number of a given graph. This chapter is dedicated to heuristic methods for finding lower and upper bounds on the crossing number of an arbitrary graph. The first main section, § 6.1, focuses on a lower bound algorithm, based on the graph-to-graph embedding technique (see § 4.2.2.2), whilst the second main section, § 6.2, is dedicated to upper bound algorithms, and in particular, the computer implementations of two-page algorithms, as well as an algorithm for generating drawings from two-page layouts.



### 6.1 A lower bound algorithm for the crossing number

In § 4.2.2.2, it was shown how Shahrokhi, Sýkora, Székely and Vrto [SSSV96a, SSSV94] applied the graph-to-graph embedding method to the problem of determining a lower bound for a graph  $\mathcal{H}$ , given a graph  $\mathcal{G}$  with a known lower bound on its crossing number, and a graph-to-graph embedding  $\psi = (\psi^{(v)}, \psi^{(e)})$  of  $\mathcal{G}$  into  $\mathcal{H}$ .

Shahrokhi, *et al.* state no algorithmic method by which to generate the injection  $\psi^{(v)}$  or  $\psi^{(e)}$ , meaning that the quality of the solution obtained by means of their technique depends on the quality of a guess, which limits the general applicability of the method. Furthermore, they made some overly strict assumptions with regards to how the crossing number bound is computed from the edge and vertex congestion values.

This section firstly deals with the question of how the lower bound of Shahrokhi, *et al.* may be improved analytically. After this, a heuristic algorithm for finding an edge embedding  $\psi^{(e)}$ , given a vertex embedding  $\psi^{(v)}$  is discussed. It is then shown how the ideas of the two sections may be integrated for a complete and practicable lower bound algorithm, and finally, the problem of determining an appropriate graph to embed into an input graph is considered.

### 6.1.1 Theoretical improvements

Whilst algebraically compact, the lower bound of Shahrokhi, Székely, Sýkora and Vrto’s lower bound inequality (4.10), reproduced here as

$$\nu(\mathcal{H}) \geq \frac{\nu(\mathcal{G}) - |V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2}}{[c^{(e)}(\psi)]^2}, \tag{6.1}$$

does not take into account the facts that:

1. the vertex congestion values of some vertices may be somewhat smaller than  $c^{(v)}(\psi)$ ,
2. there are cases when, even though two paths share a single vertex (and thus both contribute towards the vertex’s congestion value), they need never cross one another,
3. the number of crossings between a pair of edges need not be as large as  $[c^{(e)}(\psi)]^2$  in all cases, and thus the lower bound could be improved in some cases.

This section is divided into two subsections, the first of which is dedicated to the problem of improving the lower bound analytically (by examining how vertex congestion values influence the bound) and the second of which is concerned with the problem of replacing the factor  $[c^{(e)}(\psi)]^2$  in the denominator of (6.1) with a smaller value.

#### 6.1.1.1 How does the vertex congestion relate to crossings at a vertex?

Firstly, a trivial observation regarding (6.1), is that unless the vertex congestion values of all vertices in  $\mathcal{H}$  are equal to one another, the term  $|V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2}$  may be replaced by  $\sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}$  for an improved lower bound. In [SSSV96a], [SSSV94] Shahrokhi, Székely, Sýkora and Vrto were mostly interested in the asymptotic behaviour of their lower bound, which is probably why they substituted the factor  $\sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}$  by the factor  $|V(\mathcal{H})| \binom{c^{(v)}(\psi)}{2}$ .

They did, however, not consider whether situations arise from a graph-to-graph embedding where one can be assured that when two paths share a vertex, they need never cross one another. This means that for a vertex  $v$  with the congestion value  $c^{(v)}(v, \psi)$ , the number of crossings that may possibly occur at  $v$  might be much smaller than  $\binom{c^{(v)}(v, \psi)}{2}$ .

Now consider the way that vertex congestion influences the lower bound (6.1). From § 4.2.2.2 and Figure 4.4, it may be seen that when determining upper bounds, a high vertex congestion value for a vertex does not necessarily imply that all edges passing through that vertex will cross one another, thereby leading to a high crossing count at the vertex.

The ideas used for the upper bound method cannot be applied directly to the lower bound method, since, in the upper bound method, some crossings are avoidable by virtue of the way that edges are ordered around their incident vertices in the given drawing  $\phi$  of  $\mathcal{H}$ , into which  $\mathcal{G}$  is being embedded — such a favourable ordering of edges makes it possible to avoid an edge crossing in Figure 4.4(b), whereas the edge crossing in Figure 4.4(c) forces a crossing. Of course, no assumption about the order of edges around vertices may be made for the lower bound method, since one is working with an unspecified drawing of  $\mathcal{H}$ . Thus, when there is a possibility of a crossing for the lower bound method, one must assume that it will be manifested.

Consider the case where two mapped paths  $p_1 = \psi^{(e)}(e_1)$  and  $p_2 = \psi^{(e)}(e_2)$  share an edge  $\{v_1, v_2\}$ , as shown in Figures 6.1(a) and (b). Although no assumptions may be made concerning

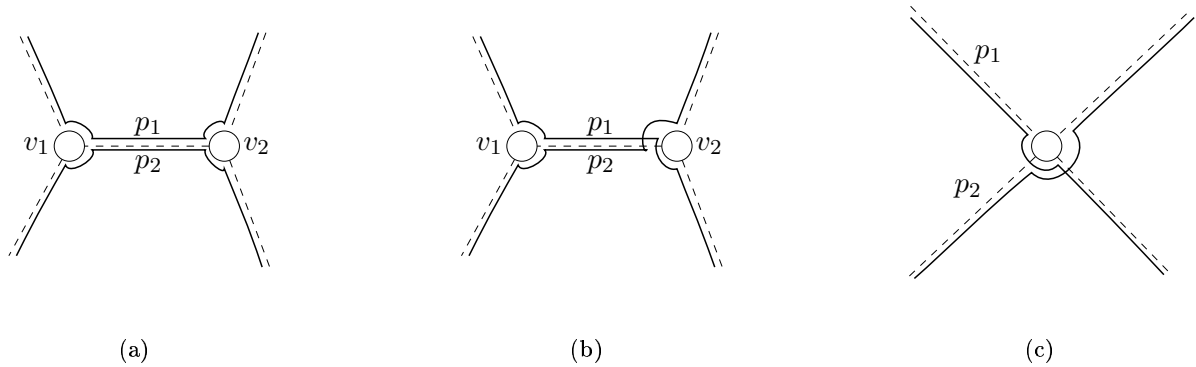


Figure 6.1: For each sub-path shared by a pair of paths, at most a single crossing need occur.

the orderings of edges around  $v_1$  and  $v_2$ ,  $v_1, v_2 \in V(\mathcal{H})$ , only at most a single crossing needs to occur. This is true, because  $p_1$  and  $p_2$  may always be drawn so that no crossing occurs at  $v_1$ , and at most a single crossing may occur at  $v_2$ , as shown in Figure 6.1(b). A special case of this scenario is where  $p_1$  and  $p_2$  share a single vertex, as shown in Figure 6.1(c) — this may be seen as the case where the common edge in the paths  $p_1$  and  $p_2$  in Figures 6.1(a) and (b) is contracted to a single vertex; since the order of the edges is unknown, a crossing must be assumed possible.

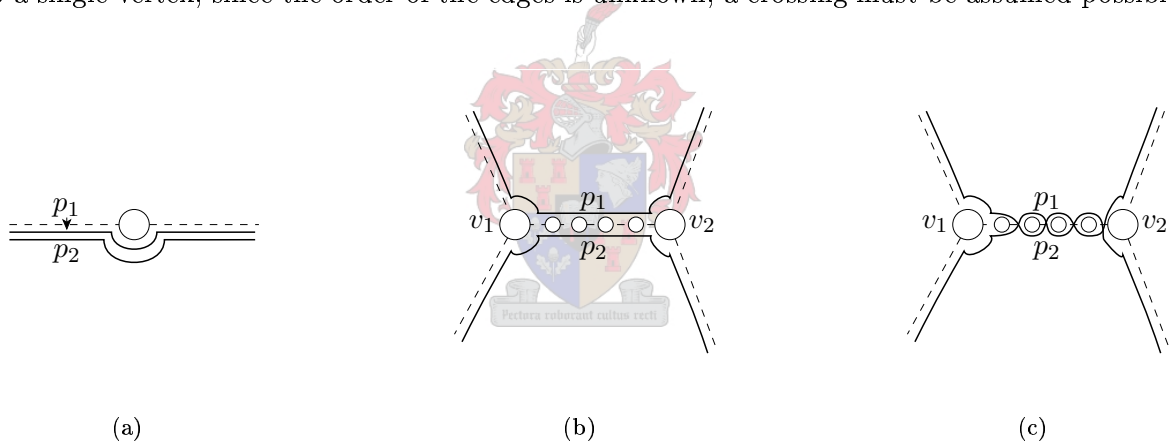


Figure 6.2: No crossings need occur at internal vertices of shared sub-paths.

In another configuration, observe that when the two mapped paths  $p_1$  and  $p_2$  share a sub-path, then no crossings need occur at internal vertices of the path, as demonstrated in Figure 6.2(a).

By combining the configurations from the two previous paragraphs, one arrives at the more general situation where  $p_1$  and  $p_2$  may share entire sub-paths, demonstrated in Figure 6.2(b). Even in this general case, at most a single crossing need occur. Suppose that  $p_1$  and  $p_2$  cross each other multiple times, as may be seen in Figure 6.2(c) — using the analogy of two steel wires that are twisted around one another,  $p_1$  and  $p_2$  may be “unwound” as the wires would, to give the situation depicted in Figure 6.2(b); the only property that cannot be guaranteed is that the last crossing may be “unwound”, because this depends on the ordering of the edges around  $v_2$ , which is unknown.

It remains to consider what happens at the leaf vertices of a mapped path. In Figure 6.3(a), it is shown that when two paths  $p_1$  and  $p_2$  start at the same vertex  $v_1$ , and share a common sub-path which ends at a vertex  $v_2$ , then no crossings need occur on *any* vertex of the path  $v_1 - v_2$ . This situation is very similar to the previously described situations where a sub-path

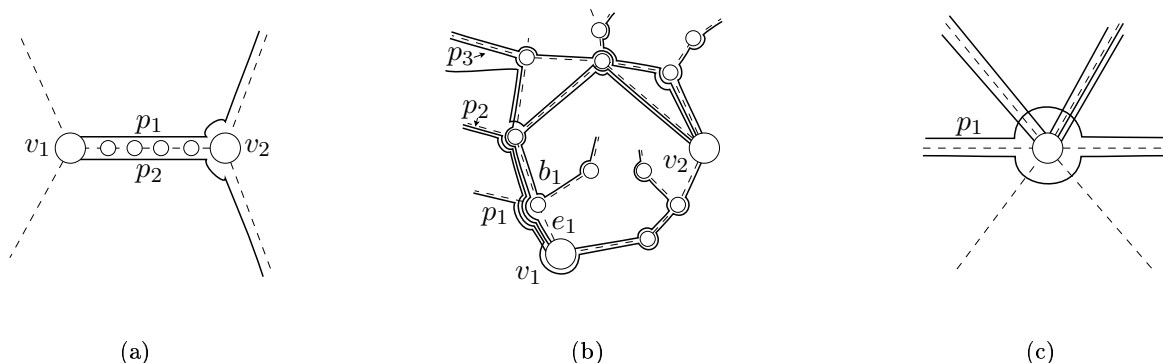


Figure 6.3: Paths starting at the same vertex need never cross one another.

may be shared between two paths. Using the metaphor again of untwisting two wires, one may start with any ordering of edges around  $v_2$ , and then proceed to start drawing  $p_1$ , parallel to  $p_2$ , from  $v_2$  towards  $v$ ; the order of  $p_1$  and  $p_2$  around  $v_1$  is automatically determined, but of course, it does not lead to a crossing. A special case of this situation is when  $p_1$  and  $p_2$  only share a leaf vertex. In this case, they share a path of length 0, and a crossing still need not be counted.

The above case is very simple, and a more convincing argument is required for the general case. Inspection of the drawing in Figure 6.3(b) shows that none of the paths starting at  $v_1$  cross one another, and the same holds true for all paths starting at  $v_2$  (although, as it may be seen, there is no reason for the paths starting at  $v_1$  not to cross those starting at  $v_2$ , except for the path  $v_1 - v_2$ , which of course may not be crossed by any of these edges). It may be seen that the paths  $p_1$ ,  $p_2$  and  $p_3$  leave  $v_1$  in that order, when moving clockwise around  $v_1$ . It is no accident that  $p_1$  veers off left before  $p_2$ , and that  $p_2$  veers off left before  $p_3$ .

A vertex  $\psi^{(v)}(u) = v \in V(\mathcal{H})$  which is an image of a vertex  $u \in V(\mathcal{G})$  under the graph-to-graph embedding  $\psi$ , must be the starting point for the paths which are the images of edges incident to  $u$ . If, for each edge  $e$  that is incident to  $v$ , only a single path runs through  $e$ , then of course no crossings can occur at  $v$ , as no edge leaving  $v$  shares an initial sub-path with any other edge leaving  $v$ . Therefore, suppose that at least one edge, say  $e_1 \in V(\mathcal{H})$ , which is incident to  $v$ , has two or more paths running through it in the embedding.

Now, referring to Figure 6.3(b), start a depth-first search down  $e_1$ , with  $v_1$  at the root. The clockwise orders of the edges of  $\mathcal{H}$  around its vertices (which are determined by the drawing  $\phi$ ) determine the order in which edges are traversed in the depth-first search — the edges are visited in clockwise order, and the first edge visited is the first edge in the clockwise order after the edge that was used to enter the current vertex (in Figure 6.3(b), the vertex  $b_1$  is entered by  $e_1$  and thus the first edges of  $p_1$  is the first edge to be visited in the depth-first search). Suppose the depth-first search reaches the vertex  $b_1$  in Figure 6.3(b), which is the last vertex that  $p_1$  has in common with the other paths shown — now  $p_1$  splits off into an edge which is left, relative to the edge through which the other paths proceed. Thus,  $p_1$  must be the left-most path to be drawn when starting all path drawings at  $e_1$ . If  $p_1$  had, however, split off to the right, then it would have had to be drawn as the right-most path. Now one continues the depth-first search for the remaining paths, and the process continues in the same fashion. This method completely specifies in which order the paths have to draw from left to right, as they leave their leaf vertices. This depth-first search need never be executed — it is simply used to prove the assumption that sub-paths which originate at the same vertex need never cross one another.

A last issue to be addressed, is whether one may make any assumptions regarding the sides

around which paths are drawn with respect to vertices in  $\mathcal{H}$ . From Figure 6.3(c), it is apparent that, depending on whether  $p_1$  passes over the top or the bottom of the vertex, it will either cross a number of paths that enter the vertex, or none. If, when a drawing of  $\mathcal{H}$  is made, paths could be drawn around vertices such that as few as possible crossings result, then these paths will each in turn cross at most half of all the paths that enter that vertex. This could lead to a substantial reduction in the maximum number of crossings in some circumstances, and the question remains whether one may apply this in addition to the other improvement ideas mentioned above. The answer seems to be that, in general, this is not possible, since when an algorithm is designed to map paths around any side of a vertex the conditions mentioned earlier may be violated. However, this remains to be proven. This concept of “side choosing” may be applied to some special cases, when it is evident that the sides of vertices around which paths are to be drawn may be chosen freely, such as when only a single path passes through a vertex, or more generally, when a number of paths share a sub-path, “side choosing” may be applied for each vertex on the sub-path through which no other paths pass.

### 6.1.1.2 Reconsidering the maximum number of crossings due to edge congestion

The denominator  $[c^{(e)}(\psi)]^2$  in (6.1) derives from assumptions (which are reiterated in the next paragraph) made with regards to the first term in the upper bound (4.8), which is reproduced here for the sake of convenience as

$$\nu(\mathcal{G}) \leq \sum_{\substack{e_i, e_j \in E(\mathcal{H}) \\ e_i \text{ crosses } e_j \text{ in } \phi}} c^{(e)}(e_i, \psi) \times c^{(e)}(e_j, \psi) + \sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}. \quad (6.2)$$

This inequality is in terms of a graph-to-graph embedding  $\psi$ , and a particular drawing of  $\mathcal{H}$ . When the right-hand side is made independent of a particular drawing of  $\mathcal{H}$ , it must also be valid for optimal drawings of  $\mathcal{H}$ . This property means that one cannot assume that there may be any more than  $\nu(\mathcal{H})$  crossings in total in a drawing of  $\mathcal{H}$ . The number of crossings counted on the right-hand side depends, in part, on the congestion values of edges that cross one another in  $\mathcal{H}$ . Shahrokhi, *et al.* [SSSV96a, SSSV94] (§ 4.2.2.2) reasoned that, since one is considering unspecified drawings of  $\mathcal{H}$ , one must assume that each crossing is between a pair of edges which both have the maximum congestion value  $c^{(e)}(\psi)$ . Since at most  $\nu(\mathcal{H})$  crossings may be assumed to be present in a drawing of  $\mathcal{H}$ , Shahrokhi, *et al.* formulated the upper bound

$$\nu(\mathcal{G}) \leq [c^{(e)}(\psi)]^2 \nu(\mathcal{H}) + \sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}. \quad (6.3)$$

Unless the edges in the  $\nu(\mathcal{H})$  pairs of edges which cross one another all have the same edge congestion values, this bound overestimates the number of crossings. Let  $c_i$  denote an edge congestion value for an edge under  $\psi$ , and let  $c_1 \geq c_2 \geq \dots \geq c_{|E(\mathcal{H})|}$  (thus  $c_1 = c^{(e)}(\psi)$ ). Now if the two edges  $e_1$  and  $e_2$  with the highest and second highest edge congestion values respectively (the values may be equal) cross one another,  $c_1 \times c_2$  crossings result. Since these edges may cross one another at most once, the worst case crossing total for another pair of edges is  $c_1 \times c_3$ . The next worst case crossing total after that is  $c_1 \times c_4$ ; when  $e_1$  has been crossed by each edge, the next worst case is  $c_2 \times c_3$ , and so on. Thus the first  $\nu(\mathcal{G})$  pairs of congestion values in a lexicographical ordering of these pairs of values, give the worst case crossing values for the  $\nu(\mathcal{G})$  crossings. Of course, since  $\mathcal{H}$  must be in single-cross normal form, adjacent edges may not cross one another, but for simplicity, it is assumed that any pair of edges may cross — this is still

valid, if not ideal (better bounds may of course be obtained by taking into account that some pairs of edges will never cross one another).

Denote a pair of edges by a tuple  $t = (c_j, c_k)$ , where  $t_1 = c_j$  and  $t_2 = c_k$ . For two tuples  $s, t$  let  $s < t$  if  $s$  comes before  $t$  in a lexicographical ordering. There are in total  $\binom{|E(\mathcal{H})|}{2}$  such tuples, corresponding to the  $\binom{|E(\mathcal{H})|}{2}$  pairs of edges in  $\mathcal{H}$ . Attach indices to the tuples, so that  $t_1 \leq t_2 \leq \dots \leq t_{\binom{|E(\mathcal{H})|}{2}}$ . Then, in the worst case, the number of crossings that will be counted, is  $\sum_{i=1}^{\nu(\mathcal{G})} t_{i1} \times t_{i2}$ . This leads to the upper bound

$$\nu(\mathcal{G}) \leq \alpha_{\psi}^* \nu(\mathcal{H}) + \sum_{v \in V(\mathcal{H})} \binom{c^{(v)}(v, \psi)}{2}, \quad (6.4)$$

where  $\alpha_{\psi}^* = \left( \sum_{i=1}^{\nu(\mathcal{G})} t_{i1} \times t_{i2} \right) / \nu(\mathcal{H})$ . The value  $\alpha_{\psi}^*$  is called the *maximized average edge congestion of  $\mathcal{H}$  with respect to  $\psi$* . Of all of the drawings of  $\mathcal{H}$  that realize its crossing number, let  $\phi'$  be such a drawing where the sum of products of the edge congestion value for edges involved in crossings attains a minimum. Denote each tuple of crossing edges by  $s$ , and the set of such tuples by  $S$ . Define  $\alpha_{\psi}$  in a similar fashion to  $\alpha_{\psi}^*$ , namely

$$\alpha_{\psi} = \left( \sum_{s \in S} s_1 \times s_2 \right) / \nu(\mathcal{H}).$$

From this it may be seen that  $\alpha_{\psi}^* \geq \alpha_{\psi}$ . The value  $\alpha_{\psi}$  is called the *minimized average edge congestion of  $\mathcal{H}$  with respect to  $\psi$* . This value is important, since it gives the true reflection of the average edge congestion of edges involved in crossings. When  $\alpha_{\psi}$  is substituted for  $\alpha_{\psi}^*$ , the first term in (6.4) attains its minimum.

The bad news is that  $\alpha_{\psi}$  cannot be computed, since one needs to enumerate all optimal drawings of  $\mathcal{H}$  in order to determine  $\alpha_{\psi}$ . To make matters worse, not even  $\alpha_{\psi}^*$  can be computed, since one needs to know  $\nu(\mathcal{H})$  in order to do so. It is, however, possible to compute an approximation to  $\alpha_{\psi}^*$ .

Note that, for any  $j \leq \nu(\mathcal{H})$ , it is true that

$$\beta_{\psi, j} = \left( \sum_{i=1}^j t_{i1} \times t_{i2} \right) / j \geq \left( \sum_{i=1}^{\nu(\mathcal{G})} t_{i1} \times t_{i2} \right) / \nu(\mathcal{H}) = \alpha_{\psi}^*,$$

since the average of the first  $j$  largest tuples must certainly be larger than or equal to the average of the first  $\nu(\mathcal{H})$  tuples. Paradoxically, it might seem, one needs a lower bound to  $\nu(\mathcal{H})$  in order to compute a value  $\beta_{\psi, j}$ . Note, however, that any lower bound will do, and any of the analytical techniques from § 4.2.2 may be employed. When  $j = 1$ , one has  $\beta_{\psi, 1} = t_{11} \times t_{12} = c_1 \times c_2 \leq [c^{(e)}(\psi)]^2$ , which is a marginal improvement over the bound (6.3) by Shahrokhi, et al. [SSSV96a, SSSV94]. An interesting aspect of this superficially circular argument, is that the lower bound may be improved by consecutive feedback. That is, if  $j_1$  is an initial (weak) lower bound, which is used in the term  $\beta_{\psi, j_1}$ , then the lower bound  $j_2$  which is obtained from an inequality such as (6.4), may itself play the role of  $j_1$  in a subsequent computation to obtain yet a better lower bound. This process indeed plays an important role in the application of the lower bound algorithm presented in the next section.

It is noted, that when the edge congestion values of all edges in  $\mathcal{H}$  are equal,  $\beta_{\psi, j} = \alpha_{\psi}^* = \alpha_{\psi}$ . This is exactly what one expects, and it shows again why it is beneficial to find graph-to-graph



embeddings for which the edge congestion values are as equal as possible. However, since the edge congestion values will not always be exactly equal, one could still benefit marginally from using the approximation  $\beta_{\psi,j}$  for some  $j$  as opposed to the factor  $[c^{(e)}(\psi)]^2$  for the denominator of (6.1).

### 6.1.2 A heuristic for edge and vertex congestion computation

The problem of finding a graph-to-graph embedding  $\psi$  from a graph  $\mathcal{G}$  to a graph  $\mathcal{H}$  with a maximum edge congestion value of at most  $k_e$  and a maximum vertex congestion value of at most  $k_v$ , is a decision problem in **NP** (since it may be verified in polynomial time whether the maximum edge congestion value is smaller than or equal to  $k_e$  and whether the maximum vertex congestion value is smaller than or equal to  $k_v$ ). Finding the minimum values possible for  $k_e$  and  $k_v$  is computationally more expensive; any algorithm for determining these values is likely to have at least an exponential running time. There is no great advantage to be obtained from an exact solution, because this would, in general, still not allow one to use this lower bound method to find the crossing number of a graph, since the lower bound is never guaranteed to be sharp. This provides a good case in favour of the development of heuristic methods for generating graph-to-graph embeddings.

In this section, such an algorithm is developed, that, when given a vertex mapping  $\psi^{(v)}$  (some ideas for generating  $\psi^{(v)}$  are given at the end of this section, although finding good embeddings is by no means a trivial task), generates an edge mapping  $\psi^{(e)}$ . Like many heuristic algorithms, this algorithm is a greedy algorithm — it starts with the empty edge mapping  $\psi^{(e)}$ , and sequentially maps the edges from  $\mathcal{G}$  to paths in  $\mathcal{H}$ , whilst at each step it attempts to find a path in  $\mathcal{H}$  which minimizes the maximum vertex and edge congestion values. If the congestion values are reinterpreted as weights, the problem of finding suitable paths is transformed to the problem of finding shortest paths.

From (6.1), it seems fair to assume that the edge congestion has a greater impact on the final lower bound, and the algorithm should thus first attempt to avoid finding a path which will increase the maximum edge congestion before one that would increase the vertex congestion values of some vertices. Thus, the edge weights for the shortest paths algorithm have to be chosen in a way that satisfies this requirement.

The weight of an edge  $e = \{v_i, v_j\} \in E(\mathcal{H})$  ought to be influenced by its congestion, but also by the congestion values of its incident vertices  $v_i$  and  $v_j$ . This makes sense, since the vertex congestion should also be minimized. One could simply compute the weight of  $e$  as the weighted sum of  $c^{(e)}(e, \psi)$ ,  $c^{(v)}(v_i, \psi)$  and  $c^{(v)}(v_j, \psi)$ . However, consider Figure 6.4, in which  $v_i$  has a high vertex congestion relative to other vertices (not shown),  $v_j$  has a relatively low vertex congestion, and  $c^{(e)}(\{v_i, v_j\}, \psi)$  is not equal to  $c^{(e)}(\psi)$ . Suppose that whilst the shortest path algorithm is embedding an edge from  $\mathcal{G}$  into a path in  $\mathcal{H}$ , it arrives at  $v_i$  before it arrives at  $v_j$  — in this case (presuming that the vertex congestion values of all of the neighbours of  $v_i$  which are not shown, are high enough) it would make sense to choose  $\{v_i, v_j\}$  as its next edge, since the congestion of  $v_j$  is low. On the other hand, suppose that the shortest path algorithm arrives at  $v_j$  first, then unless the vertex congestion values of the neighbours of  $v_j$  which are not shown, are higher than that of  $v_i$ ,  $\{v_i, v_j\}$  should not be chosen.

The point is that desirability of an edge selection during the mapping procedure depends on the endpoint of the edge at which a shortest path algorithm starts, because the congestion values of the edge's incident vertices may differ. This problem may be resolved by transforming  $\mathcal{H}$  to a directed graph, where an arc  $(v_i, v_j)$  is weighted by the weighted sum of the congestion of the

undirected edge  $\{v_i, v_j\}$  and the congestion of  $v_j$ . Its opposite,  $(v_j, v_i)$ , is weighted in the same way, except for the substitution of  $v_j$  for  $v_i$ .

Assuming that two possible paths  $p_1$  and  $p_2$  in  $\mathcal{H}$  neither contain an arc with the maximum edge congestion, their weights should be chosen so as to keep the possibility of crossings due to vertex congestion values as low as possible. This may be achieved by choosing the vertex congestion component of the arc weights on  $p_1$  and  $p_2$  simply as the vertex congestion values of their target vertices, since the vertex congestion of a vertex is simply interpreted as the largest number of crossings that may be caused when an arc is mapped through the vertex in question. Since the edge congestion values of the arcs on  $p_1$  and  $p_2$  are not equal to  $c^{(e)}(\psi)$ , the vertex congestion values of vertices in  $p_1$  and  $p_2$  are likely to have a more significant impact on the quality of the lower bound than the edge congestion values could (besides, it is only when these edge congestion values are included in the computation of  $\beta_j$ ,  $1 \leq j \leq \nu(\mathcal{H})$ , that they will have any impact at all, but even then, any gain here is likely to be offset by larger vertex congestion values). Nevertheless, one wants to discriminate between paths that are equally favourable in terms of vertex congestion values, but which have different edge congestion values — it makes sense in this case to ensure that the sum of the edge congestion components for an entire path remains smaller than 1, so that it cannot affect the choice of paths unless the paths have the same sum total of vertex congestion values; this may be achieved by taking the edge congestion component of the weight of an arc  $e$  to be  $c^{(e)}(e, \psi)/(c^{(e)}(\psi) \times |V(\mathcal{H})|)$ , since there are at most  $|V(\mathcal{H})|$  vertices on any path in  $\mathcal{H}$ , and since  $c^{(e)}(e, \psi) < c^{(e)}(\psi)$ .

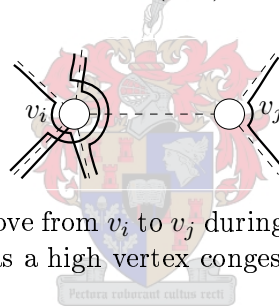


Figure 6.4: It is more desirable to move from  $v_i$  to  $v_j$  during the construction of a graph-to-graph embedding, than *vice-versa*, if  $v_i$  has a high vertex congestion compared to that of  $v_j$ .

The algorithm should aggressively attempt to keep the maximum edge congestion value as low as possible. Thus, the arc weights are selected so that very long paths containing vertices with high vertex congestion values are preferred to even a single arc of which the congestion value is equal to the maximum, when the congestion of no arc on such a path achieves the maximum congestion value. The largest congestion value that a vertex  $v \in V(\mathcal{H})$  may attain is  $|E(\mathcal{G})|$  (which is, of course, when each edge in  $\mathcal{G}$  is mapped through  $v$ ). The largest number of vertices that any path in  $\mathcal{H}$  may contain is, of course,  $|V(\mathcal{H})|$  vertices. This means, that, when taking only vertex congestion values into account, the maximum weight for any path in  $\mathcal{H}$  is  $|E(\mathcal{G})| \times |V(\mathcal{H})|$ . Thus, in order for the algorithm to avoid an arc with the maximum edge congestion, the arc should have  $|E(\mathcal{G})| \times |V(\mathcal{H})|$  added to its weight (which would push the arc weight beyond  $|E(\mathcal{G})| \times |V(\mathcal{H})|$ , as the congestion value of its target vertex must also be added to it), thereby ensuring that it will be avoided in favour of arcs which do not attain the maximum edge congestion.

Finally, no arc should be assigned a weight of zero, since this would render any pair of paths for which all arcs have zero weight equally favourable, regardless of the difference in the number of arcs in the two paths. It is desirable that mapped paths should be as short as possible, because this reduces the number of arcs of which the edge congestion has to be increased, improving the chances of finding better mappings. For this reason, when, for an arc  $e = (u, v)$ , both the edge congestion and the vertex congestion of  $v$  are zero,  $e$  is weighted by a small positive constant  $\varepsilon$ , which should preferably be much smaller than 1, so as not to render a path on which  $e$  appears undesirable.

### 6.1.2.1 The implementation of the algorithm

In this subsection the algorithm is specified in its entirety. The algorithm sequentially embeds the edges of  $\mathcal{G}$  into the bidirected graph  $\mathcal{H}$ . For each embedding of an edge  $e = \{s, t\} \in E(\mathcal{G})$ , to a  $\psi^{(v)}(s) - \psi^{(v)}(t)$  path  $\mathcal{P}$ ,  $\mathcal{P}$  is traversed, one vertex at a time from  $\psi^{(v)}(t)$  to  $\psi^{(v)}(s)$ . For each vertex  $v$  in  $\mathcal{H}$ , the algorithm maintains a set  $\Xi_v$  of paths (from prior mapped edges) which map through  $v$ . This allows the algorithm to determine the paths in  $\mathcal{H}$  with which  $\mathcal{P}$  shares sub-paths, and using this information, it can compute the number of crossings that  $\mathcal{P}$  will cause.

A set  $\tau$  contains the paths considered by the algorithm at the previous vertex that it visited in the path  $\mathcal{P}$ .

When the algorithm considers a vertex  $v$  in  $\mathcal{P}$ , the paths contained in  $\Xi_v$ , but not in  $\tau$ , did not map through the vertex that is prior to  $v$  in  $\mathcal{P}$ , and therefore,  $\mathcal{P}$  is not part of an intersecting sub-path with any such path in  $\Xi_v$ . At  $v$ , however,  $\mathcal{P}$  commences shared sub-paths with the paths in  $\Xi_v$ . Because a crossing must be counted for each shared sub-path (except for those sub-paths which terminate at one of the end vertices of  $\mathcal{P}$ ), crossings are counted at the first vertex of such shared sub-paths (*i.e.*,  $v$  in this case). After all paths in  $\Xi_v$  have been considered, the list  $\tau$  is cleared, and filled with the paths in  $\Xi_v$ , so that the algorithm may again determine the sub-paths in which  $\mathcal{P}$  occurs at the next vertex after  $v$  in  $\mathcal{P}$ . Special cases for this scheme occur at the vertices  $\psi^{(v)}(s)$  and  $\psi^{(v)}(t)$  which are the end vertices of  $\mathcal{P}$ :

- At  $\psi^{(v)}(t)$ , where the traversal of  $\mathcal{P}$  commences, crossings are only counted for paths in  $\Xi_{\psi^{(v)}(t)}$  which do not have  $\psi^{(v)}(t)$  as an end vertex. As an example of such a possible crossing, consider the path  $p_2$  which has  $v_2$  as an end vertex, and crosses the path  $p_1$  which commences at  $v_1$  in Figure 6.3(b). The paths in  $\Xi_{\psi^{(v)}(t)}$  which end at  $\psi^{(v)}(t)$ , have already been shown never to have to cause crossings in the sub-paths they share with  $\mathcal{P}$ .
- At  $\psi^{(v)}(s)$ , which is the last vertex to be visited in  $\mathcal{P}$ , it may happen that some other paths in  $\mathcal{H}$  with which  $\mathcal{P}$  shares sub-paths, also have  $\psi^{(v)}(s)$  as an end vertex. These are sub-paths for which no crossings need occur, but for shared sub-paths of length at least 1 (*i.e.*, not the special case sub-paths which are, in fact, single shared end vertices of separate, otherwise non-intersecting paths), crossings would have been counted for the vertices where these sub-paths joined  $\mathcal{P}$ . This must be corrected by subtracting a crossing for each of these paths.

Although crossings are counted at the vertices where two paths join into a common sub-path, there is nothing special about these vertices. As has been noted before, the crossing between a pair of paths may occur at any vertex that is part of a shared sub-path. For each vertex  $v$  in a graph  $\mathcal{H}$ , there is no running total that is kept for such crossings, which is then incremented when a pair of paths join at  $v$ . If this were the case, it would create a situation where  $v$  itself would become undesirable, due to an increased “congestion”, although the other vertices in the sub-path of which it is an end-vertex, would seem relatively more desirable. This is not desirable, because an entire sub-path should logically be treated as a crossing, which should preferably be avoided if possible. The vertex congestion values are better indicators of the suitability of vertices for shortest path routing, because all vertices in a common sub-path of two distinct paths will receive increased congestion values as opposed to only a single vertex with the former mechanism.

The variables  $\tau$  and  $\Xi$  were said to maintain sets of paths, and this simplifies the algorithm conceptually. It is, however, cumbersome to implement an algorithm which manipulates paths. A path  $\mathcal{P}$  is just the image of an edge  $e$  in  $\mathcal{G}$ , and the edges of  $\mathcal{G}$  may therefore be used

**Algorithm 6.1** ComputeWeights

**Input:** Graphs  $\mathcal{G}$  and  $\mathcal{H}$ , where  $\mathcal{G}$  is being graph-to-graph embedded into  $\mathcal{H}$ . The edge congestion values  $\mathbf{c}^{(e)}$  and the vertex congestion values  $\mathbf{c}^{(v)}$  of  $\mathcal{H}$ .

**Output:** A set of arc weights  $\mathbf{w}$  for  $\mathcal{H}$ .

```

1: for all  $e = (u, v) \in E(\mathcal{H})$  do
2:    $\mathbf{w}_e \leftarrow \mathbf{c}_v^{(v)}$ 
3:   if  $\mathbf{c}_e^{(e)} = \max_{e \in E(\mathcal{H})} \mathbf{c}_e^{(e)}$  then
4:      $\mathbf{w}_e \leftarrow \mathbf{w}_e + |E(\mathcal{G})|^2 \times |V(\mathcal{H})|$ 
5:   else
6:      $\mathbf{w}_e \leftarrow \mathbf{w}_e + \mathbf{c}_e^{(e)} / (\max_{e \in E(\mathcal{H})} \mathbf{c}_e^{(e)} \times |V(\mathcal{H})|)$ 
7:   end if
8:   if  $\mathbf{c}_e^{(e)} = 0$  and  $\mathbf{c}_v^{(v)} = 0$  then
9:      $\mathbf{w}_e \leftarrow \varepsilon$ 
10:  end if
11: end for
12: return  $\mathbf{w}$ 

```

**Implementation:** An implementation of this algorithm may be found in § B.3.

to represent the paths of  $\mathcal{H}$ . This is exactly the method used by the implementation of the algorithm. Therefore,  $\tau$  and  $\Xi_v$ , are taken as sets of edges of  $\mathcal{G}$ .

A variable  $C$  maintains the count of necessary crossings, and it is initialized to zero at line 1 of Algorithm 6.2. The main loop between lines 2 and 33 enumerates every edge in the input graph  $\mathcal{G}$ . Within this loop each edge is embedded into  $\mathcal{H}$ . The arc weights, which are required for the execution of Dijkstra's shortest path algorithm, are computed at line 3 by the algorithm **ComputeWeights**; this algorithm is described later. At line 4, Dijkstra's shortest paths algorithm is applied to  $\mathcal{H}$ , with the source vertex  $\psi^{(v)}(s)$  (although the vertex  $\psi^{(v)}(t)$  could equally well have been used as the source vertex), and the weight vector  $\mathbf{w}$ . Dijkstra's algorithm returns a parent vector  $\pi$ , where the vector component  $\pi_v$  is the parent of a vertex  $v$  in the shortest path tree (where the source vertex  $\psi^{(v)}(s)$  is at the root of the tree).

The parent vector is used to traverse the path  $\mathcal{P}$ , which is the image of  $e$  in  $\mathcal{H}$ . Before traversal of  $\mathcal{P}$  is initiated, the vertex  $\psi^{(v)}(t)$  must be handled as a special case (as described earlier) with regards to the manner in which other paths which map through  $\psi^{(v)}(t)$  might cause crossings with  $\mathcal{P}$ . All of the edges of  $\mathcal{G}$  which map to paths that intersect the vertex  $\psi^{(v)}(t)$ , are enumerated in the loop between lines 6–10. These edges are added to  $\tau$  for use of the next vertex after  $\psi^{(v)}(t)$  in  $\mathcal{P}$ . As has been stated before, crossings are only counted for those paths that do not end at  $\psi^{(v)}(t)$ ; this is verified at line 7, and a crossing for  $\psi^{(v)}(t)$  is added to  $C$  for each edge of which the image in  $\mathcal{H}$  does not have  $\psi^{(v)}(t)$  as an end vertex.

The actual traversal of  $\mathcal{P}$  is performed within the loop spanning lines 13–32. For each iteration, the vertex  $v$  is moved along by one vertex in  $\mathcal{P}$ , and a test is performed at the start of the loop (line 13) to determine whether  $v$  has reached the source vertex  $\psi^{(v)}(s)$ . Before the position of  $v$  is updated at line 16, the edge congestion values of the two opposite faces arcs  $(v, \pi_v)$  and  $(\pi_v, v)$  which join  $v$  and its parent, are updated (lines 14 and 15). Because these two arcs represent the same undirected edge  $\{v, \pi_v\}$  in the underlying graph of  $\mathcal{H}$ , they are assigned the same congestion values; hence the assignment of line 15.

The loop spanning lines 18–22, is responsible for finding the edges in  $\mathcal{G}$  whose images in  $\mathcal{H}$  initiate common sub-paths with  $\mathcal{P}$  at  $v$ . Each edge  $f \in \Xi_v$  is enumerated, and it is determined at line 19 whether  $f \in \tau$ . If this is not the case, then there are two situations to consider:

**Algorithm 6.2** ComputeEmbedding**Input:** A graph  $\mathcal{G}$ , a bidirected graph  $\mathcal{H}$ , and a vertex mapping  $\psi^{(v)} : V(\mathcal{G}) \rightarrow V(\mathcal{H})$ .**Output:** An edge mapping  $\psi^{(e)}$ .

```

1:  $C \leftarrow 0$ 
2: for all  $e = \{s, t\} \in E(\mathcal{G})$  do
3:    $\mathbf{w} \leftarrow \text{ComputeWeights}(\mathcal{G}, \mathcal{H}, \mathbf{c}^{(e)}, \mathbf{c}^{(v)})$ 
4:    $\pi \leftarrow \text{Dijkstra}(\mathcal{H}, \psi^{(v)}(s), \mathbf{w})$ 
5:    $v \leftarrow \psi^{(v)}(t)$ 
6:   for all  $f = (w, x) \in \Xi_v$  do
7:     if  $\psi^{(v)}(w) \neq \psi^{(v)}(t)$  and  $\psi^{(v)}(x) \neq \psi^{(v)}(t)$  then
8:        $C \leftarrow C + 1$ 
9:     end if
10:  end for
11:   $\tau \leftarrow \bar{\Xi}_v$ 
12:   $\mathbf{c}_v^{(v)} \leftarrow \mathbf{c}_v^{(v)} + 1$ 
13:  while  $v \neq \psi^{(v)}(s)$  do
14:     $\mathbf{c}_{(\pi_v, v)}^{(e)} \leftarrow \mathbf{c}_{(\pi_v, v)}^{(e)} + 1$ 
15:     $\mathbf{c}_{(v, \pi_v)}^{(e)} \leftarrow \mathbf{c}_{(\pi_v, v)}^{(e)}$ 
16:     $v \leftarrow \pi_v$ 
17:     $\mathbf{c}_v^{(v)} \leftarrow \mathbf{c}_v^{(v)} + 1$ 
18:    for all  $f = (w, x) \in \Xi_v$  do
19:      if  $f \notin \tau$  and  $(v \neq \psi^{(v)}(s) \text{ or } [\psi^{(v)}(x) \neq \psi^{(v)}(s) \text{ and } \psi^{(v)}(w) \neq \psi^{(v)}(s)])$  then
20:         $C \leftarrow C + 1$ 
21:      end if
22:    end for
23:    if  $v = \psi^{(v)}(s)$  then
24:      for all  $f = (w, x) \in \tau$  do
25:        if  $\psi^{(v)}(w) = \psi^{(v)}(s)$  or  $\psi^{(v)}(x) = \psi^{(v)}(s)$  then
26:           $C \leftarrow C - 1$ 
27:        end if
28:      end for
29:    end if
30:     $\tau \leftarrow \bar{\Xi}_v$ 
31:     $\bar{\Xi}_v \leftarrow \bar{\Xi}_v \cup \{e\}$ 
32:  end while
33: end for

```

**Implementation:** An implementation of this algorithm is given in § B.3.

1. the vertex  $v$  is not at the end of  $\mathcal{P}$ , and therefore the boolean expression  $v \neq \psi^{(v)}(s)$  will be true, which renders the entire boolean expression in line 19 true. A crossing will be added for  $v$  in accordance with the prior explanation.
2. the vertex  $v$  is at the end of  $\mathcal{P}$ , and crossings only need to be counted for those edges in  $\mathcal{G}$  which map to paths in  $\mathcal{H}$  that do not have  $\psi^{(v)}(s)$  as an end vertex. This is verified by the boolean expression  $[\psi^{(v)}(x) \neq \psi^{(v)}(s) \text{ and } \psi^{(v)}(w) \neq \psi^{(v)}(s)]$  where  $\psi^{(v)}(x)$  and  $\psi^{(v)}(w)$  are the end vertices of the path corresponding to  $f$ .

The last portion of complex code may be found between lines 23–29. This code is only executed when  $v$  is at the final vertex of the path  $\mathcal{P}$  (i.e., the vertex  $\psi^{(v)}(s)$ ). The crossings that were

added for sub-paths that end at  $\psi^{(v)}(s)$  are corrected here. The edges of  $\mathcal{G}$  for which the paths end in  $\psi^{(v)}(s)$ , must be in  $\tau$ , since by definition,  $\tau$  contains a list of edges whose paths are on sub-paths with  $\mathcal{P}$ . One of the two incident vertices of these edges must map to  $\psi^{(v)}(s)$ ; hence the test at line 25. For each such edge, a crossing is subtracted on line 26. Finally, the variable  $\tau$  is updated to reflect the edges of  $\mathcal{G}$  whose paths share sub-paths with  $\mathcal{P}$  (line 30), and the set of edges  $\Xi_v$ , that pass through  $v$  is updated (line 31).

The arc weights are computed by the algorithm `ComputeWeights` (Algorithm 6.1). The assignments correspond to the method given at the end of the previous section. The algorithm enumerates each arc  $e = (u, v) \in E(\mathcal{H})$ , and it computes the weight of  $e$  as a weighted sum of the congestion of the target vertex  $v$  of  $e$ , and of the edge congestion of  $e$  itself. Initially,  $w_e$  is assigned the vertex congestion value of  $v$  (line 2). The added edge congestion component depends on whether the edge congestion of  $e$  achieves the maximum edge congestion value or not. The two possibilities in lines 4 and 6 correspond to previously described assignment strategy. The only time when  $w_e$  is assigned a predetermined value, is when the edge congestion of  $e$  and the vertex congestion value of  $v$  are both zero, so as to favour the selection of shorter paths.

When all of the edges of  $\mathcal{G}$  have been mapped to  $\mathcal{H}$ , a lower bound for the crossing number of  $\mathcal{H}$  is

$$\nu(\mathcal{H}) \geq \frac{\nu(\mathcal{G}) - C}{\beta_{\psi,j}}, \quad (6.5)$$

where  $C$  is the variable from Algorithm 6.2 which contains the total number of possible crossings, and  $j \geq 1$ .

When one chooses the graph  $\mathcal{G}$  to be a simple graph (*i.e.*, a graph in which each pair of vertices is joined by at most a single edge), it may happen that the term  $C$  becomes almost as large as  $\nu(\mathcal{G})$ . This problem may be ameliorated by a result due to Kainen [Kai72]. Let  $k\mathcal{G}$  denote the graph that is obtained from a simple graph  $\mathcal{G}$  by replacing each of the edges of  $\mathcal{G}$  by  $k$  parallel edges. The graph  $k\mathcal{G}$  is an example of a multi-graph (*i.e.*, a graph in which a pair of vertices may be joined by multiple edges; it should, however, be noted that in a multi-graph in general, it is not necessary that each pair of adjacent vertices be joined by the same number of edges as the other pairs of vertices as in this case). Kainen [Kai72] proved that  $\nu(k\mathcal{G}) = k^2\nu(\mathcal{G})$ . This result has the effect of transforming the inequality (6.5) into the form

$$\nu(\mathcal{H}) \geq \frac{k^2\nu(\mathcal{G}) - f(C)}{h(\beta_{\psi,j})}, \quad (6.6)$$

where  $f$  and  $h$  are functions that give the number of crossings,  $C$ , and the edge congestion component  $\beta_{\psi,j}$  under the transformation. In the worst case, all of the edges between a pair of vertices  $u$  and  $v$  in  $k\mathcal{G}$  are mapped to the same path in  $\mathcal{H}$  to which the edge  $\{u, v\}$  in the underlying graph  $\mathcal{G}$  was mapped. Then each crossing will be replaced by  $k^2$  crossings ( $k$  edges crossing over  $k$  other edges). Also, in this case, the edge congestion values are exactly increased  $k$ -fold. In this case,  $f(C) = k^2C$  and  $h(\beta_{\psi,j}) = k^2\beta_{\psi,j}$ , rendering (6.6) equivalent to (6.5). However, in cases where the edges of the multi-graph  $k\mathcal{G}$  are mapped differently, it is possible that  $f(C)$  and  $h(\beta_{\psi,j})$  could be small enough to ensure an improvement.

### 6.1.2.2 Determining good vertex embeddings

A tabu search algorithm, which is based on the tabu search algorithm later in this chapter was implemented with the goal of obtaining vertex embeddings from a graph  $\mathcal{G}$  to a graph  $\mathcal{H}$ , that would permit good edge embeddings into  $\mathcal{G}$  to be found. However, the computational complexity

required for finding edge embeddings from  $\mathcal{G}$  to  $\mathcal{H}$  is in general so high, that it is more feasible to simply attempt a number of edge embeddings from  $\mathcal{G}$  to  $\mathcal{H}$ , each with a different random vertex embedding. An implementation of this algorithm has been included in § B.3.3 for completeness.

### 6.1.2.3 Complexity of the algorithm

First, if the crossing reduction techniques described in this chapter are ignored, the problem of graph-to-graph embedding a graph  $\mathcal{G}$  to a graph  $\mathcal{H}$ , is the straightforward mapping of the edges of  $\mathcal{G}$  by means of Dijkstra’s method. For each edge in  $\mathcal{G}$ , this takes  $O(|V(\mathcal{H})|^2 \log |V(\mathcal{H})|)$  time. The updating of the arc weights in  $\mathcal{H}$  after the mapping of each edge in  $\mathcal{G}$  takes constant time, because the weight of an arc in  $\mathcal{H}$  is simply the weighted sum of its edge congestion and the vertex congestion of its target vertex. Summing this over all edges of  $\mathcal{G}$  produces a running time of  $O(|E(\mathcal{G})||V(\mathcal{H})|^2 \log |V(\mathcal{H})|) + O(|E(\mathcal{G})|)$ , where the terms correspond respectively to the invocations of Dijkstra’s algorithm, and the updating of arc weights. The latter term is insignificant with respect to the former, and therefore the running time is expressed as  $O(|E(\mathcal{G})||V(\mathcal{H})|^2 \log |V(\mathcal{H})|)$ .

## 6.2 Upper bound algorithms for the crossing number

Except for Shahrokhi, Székely, Sýkora and Vrto’s probabilistic embedding algorithm (§ 4.3.2.3), the upper bound approximation algorithms of Chapter 4 fall into the two-page combinatorial layout paradigm. The main attraction of this paradigm derives from the fact that the data structures required to represent two-page layouts are simple, and that the computations to obtain the number of crossings in a given configuration may be achieved efficiently (see § 3.2.2). For these reasons, the upper bound algorithms in this thesis have all been implemented in the two-page combinatorial layout model.

For a two-page heuristic combinatorial layout algorithm to find a good upper bound on the crossing number of a graph  $\mathcal{G}$ , it has to find an arrangement of the vertices of  $\mathcal{G}$  on the spine, that will permit a layout of the edges of  $\mathcal{G}$  with a low number of crossings. Since the problem of determining optimal arrangements is a permutation problem, its running time is  $O(n!)$  for  $n$  elements. Furthermore (as mentioned in § 4.3.2.2), the problem of finding an optimal layout of edges, given an arrangement of vertices is an **NP**-hard problem (Masuda, Nakajima, Kashiwabara and Fujisawa [MNKF90]).

With these points in consideration, heuristics are applied within two conceptual levels for a heuristic two-page layout algorithm (except for the approximation algorithm described later in this chapter which is based on Székely’s algorithm; introduced in § 4.3.1.4). On the main level, there is a *vertex arrangement heuristic* that searches for good vertex orderings on the spine. This heuristic normally “drives” an *edge layout heuristic* which is responsible for obtaining reasonable edge layouts. For example, a local optimization algorithm for minimizing the number of crossings in a graph drawing may continually scan every vertex on the spine and attempt to move it from its position to another position in an attempt to improve the solution. For each such move, a good layout of the edges must be computed, so as to ascertain whether the move is an improvement with respect to the number of crossings. Thus, in effect, the edge layout heuristic is invoked by the vertex arrangement heuristic.

First, the edge layout heuristics are described in § 6.2.1; an important concept for the understanding of the edge layouts algorithms in this thesis is that the edge layout problem may be reinterpreted as a vertex partitioning problem of an auxiliary graph called the “intersection

graph,” and this equivalence is described in this section. Section 6.2.2 is dedicated to the vertex arrangement problem; the tabu search method is employed for this purpose, and the section details the basic theory of the tabu search method, as well as an implementation of the method.

For a graph  $\mathcal{G}$ , the symbol  $A$  indicates a vertex arrangement on the spine (*i.e.*, a permutation of the vertices of  $\mathcal{G}$ ). It is simply a vector, and is indexed by position, so that  $A_i$  is the  $i$ -th vertex in the vector.  $A$  is itself identified with the spine, and as such, will be referred to either as a vertex arrangement, or as the spine.

### 6.2.1 Edge layout heuristics

Cimikowski (§ 4.3.2.2, [Cim02]) considered the problem of two-page layouts under the name of the “Fixed linear crossing number problem.” Eight heuristic algorithms were developed by him for this purpose. This section introduces two new heuristics — one is a simple, although effective iterating greedy method, whilst the other is a hybrid genetic/local improvement algorithm, which is effective, but computationally expensive.

An important (in the context of this thesis) reformulation of the problem of determining an edge layout on the spine of a book is made in this section: it is shown that determining such a layout for a given graph  $\mathcal{G}$  and an arrangement  $A$  of the vertices of  $\mathcal{G}$  is equivalent to the problem of finding a particular vertex partition of the so-called “intersection graph”  $\mathcal{X}_{\mathcal{G},A}$  corresponding to  $A$ . All of the edge layout algorithms in this thesis are actually implemented as partitioning algorithms.

#### The intersection graph

For a given vertex arrangement on the spine of a book, an edge layout algorithm has to be able to determine efficiently whether a pair of edges are alternating (since alternating edges cross). For a given edge  $e = \{u, v\}$ , where say,  $u$  occurs before  $v$  on the spine, one way to discover the set of edges that cross  $e$ , is to enumerate each vertex  $w$  that occurs between  $u$  and  $v$  on the spine and to consider each edge  $f = \{w, x\}$  incident to  $w$ , for which the other incident vertex  $x$  of  $f$  occurs either to the right of  $v$ , or to the left of  $u$  (in other words,  $f$  alternates  $e$ ). This is essentially the method employed by Cimikowski’s algorithms in [Cim02], and also by the neural network algorithm of Cimikowski and Shope [CS96]. The main shortcomings of this method are that

1. for the edge  $e = \{u, v\}$ , there may be a large number of edges for which both incident vertices lie entirely between  $u$  and  $v$ , and which therefore do not alternate  $e$ ,
2. the computations required to determine whether two edges  $e$  and  $f$  cross are performed twice — once when  $e$  is under consideration, a the second time when  $f$  is under consideration,
3. when only a single vertex  $t$  is moved on the spine, the rest of the vertex arrangement remains fixed. This means that all edges that were not incident to  $t$ , will still have the same alternation structure with respect to one another. This may be exploited to reduce the computation required, since it should only be necessary to determine anew how the edges incident to  $t$  alternate the other edges in the graph.

A simple solution to these issues exists in the form of the *intersection graph*. For a graph  $\mathcal{G}$  and a given arrangement  $A$  of the vertices of  $\mathcal{G}$  along the spine, define the intersection graph  $\mathcal{X}_{\mathcal{G},A}$  so



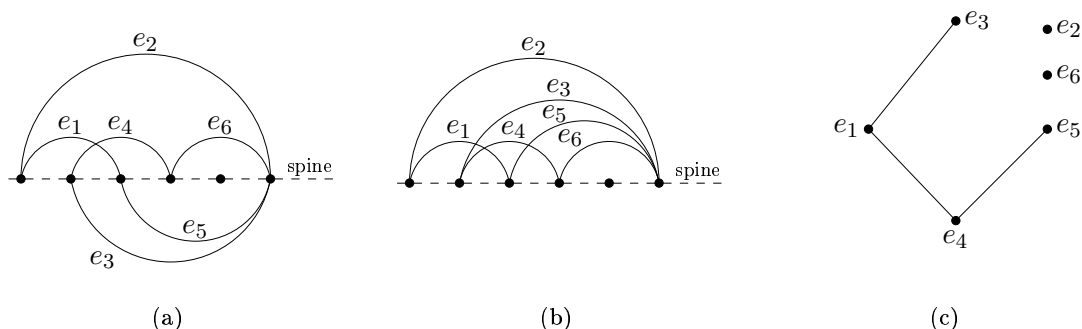


Figure 6.5: The intersection graph indicates which edges may possibly cross.

that there is a vertex in  $\mathcal{X}_{\mathcal{G},A}$  for every edge in  $\mathcal{G}$ , and so that a pair of vertices in  $\mathcal{X}_{\mathcal{G},A}$  is joined by means of an edge if the corresponding edges in  $\mathcal{G}$  alternate one another. This is stated more formally in the following definition.

**Definition 6.2.1** *The intersection graph  $\mathcal{X}_{\mathcal{G},A}$  of a graph  $\mathcal{G}$  with respect to the vertex arrangement  $A$  has the vertex and edge sets*

$$\begin{aligned} V(\mathcal{X}_{\mathcal{G},A}) &= \{v_i : e_i \in E(\mathcal{G})\}, \\ E(\mathcal{X}_{\mathcal{G},A}) &= \{f = \{v_i, v_j\} : \text{if } e_i \text{ alternates } e_j \text{ in } A, e_i, e_j \in E(\mathcal{G})\}. \end{aligned} \quad \blacksquare$$

The graph shown in Figure 6.5(c) is an intersection graph for both two–page book layouts of the same graph shown in Figures 6.5(a) and (b). This clearly illustrates that the intersection graph is independent of a given edge layout (the drawings in Figures 6.5(a) and (b) differ only with regards to edge layouts; they are otherwise identical), and only dependent on vertex arrangements.

Now, by using the intersection graph, it is a simple matter to determine the set of edges that cross a given edge  $e_i \in E(\mathcal{G})$ . This may be achieved by finding the vertex  $v_i \in V(\mathcal{X}_{\mathcal{G},A})$  corresponding to the edge  $e_i \in E(\mathcal{G})$  and by enumerating the neighbouring vertices of  $v_i$  — an edge  $e_j \in E(\mathcal{G})$  that crosses  $e_i$  will have a corresponding incident vertex  $v_j \in V(\mathcal{X}_{\mathcal{G},A})$ , and is located on the same page as  $e_i$ . Thus, the pages on which edges are drawn may be used to partition the vertices of  $\mathcal{X}_{\mathcal{G},A}$ , so that vertices which are part of the same partition correspond to edges in  $\mathcal{G}$  that are drawn on the same page. Thus, shortcomings 1 and 2 described above are addressed by using the intersection graph representation.

Now, suppose that a vertex  $v \in V(\mathcal{G})$  is moved away from its position in the vertex arrangement on the spine. To obtain an intersection graph representing the new vertex arrangement, only the edges in  $\mathcal{X}_{\mathcal{G},A}$  that are incident to the vertices in  $\mathcal{X}_{\mathcal{G},A}$  which represent edges incident to  $v$  need be deleted, and new edges must be inserted into  $\mathcal{X}_{\mathcal{G},A}$  in order to reflect the way in which edges incident to  $v$  alternate the edges in  $\mathcal{G}$ , based on its new position. This procedure addresses shortcoming 3.

The observation that a vertex partition of the intersection graph corresponds to an edge layout, obviates the necessity of keeping track of the input graph  $\mathcal{G}$  and the intersection graph  $\mathcal{X}_{\mathcal{G},A}$  at the same time — the latter maintains all the information necessary to obtain an edge layout. The total number of crossings may be computed simply by enumerating each edge  $e \in E(\mathcal{X}_{\mathcal{G},A})$ , and by adding to the total number of crossings if the vertices incident to  $e$  occur in the same partition.

A major boon of using the intersection graph is that when operating with a subdivision of a graph, the subdivision edges that are not crossed, show up as isolated vertices in the intersection graph. Therefore, an algorithm using the intersection graph to determine whether edges cross, pays no time complexity penalty for the subdivision edges that are not crossed. If the subdivision vertices of a graph are placed in the manner described in the proof of Theorem 5.3.2, then *all* the subdivision edges will be represented by isolated vertices in the intersection graph.

Although the graph  $\mathcal{G}$  is disregarded, it is conceptually still useful to think in terms of the placement of edges of  $\mathcal{G}$ , instead of the partitioning of vertices of  $\mathcal{X}_{\mathcal{G},A}$ . One could still think about edges of  $\mathcal{G}$  to mean the vertices of  $\mathcal{X}_{\mathcal{G},A}$ , but this introduces an opportunity for ambiguity when the edges of  $\mathcal{X}_{\mathcal{G},A}$  themselves need to be considered (indeed as they are in the algorithms). For this reason, the vertices and edges of  $\mathcal{X}_{\mathcal{G},A}$  are somewhat awkwardly respectively called the *intersection-vertices* and the *intersection-edges*. As a slight abuse of terminology (for the sake of convenience), intersection-vertices are identified with edges of  $\mathcal{G}$ , and the terms are used interchangeably. Thus, it is occasionally mentioned that a pair of intersection-vertices cross one another — in this case it is obviously implied that their corresponding edges in  $\mathcal{G}$  cross one another.

As an aside, the intersection graph defined here is similar in spirit to the intersection graph used in some proofs (including the one that may be found in Appendix A) of Kuratowski’s theorem. It seems that this kind of structure is, in general, a useful conceptual tool in crossing number theory.

### Reinterpreting the edge layout problem as a vertex partitioning problem

Considering that crossings are only counted for edges in the intersection graph whose incident vertices occur in the same vertex partition, it may be seen that the sum of the edges in the two disjoint subgraphs induced by the vertices from the two partitions are counted. Thus, the problem of finding an optimal edge layout is equivalent to the problem of minimizing the sum of edges in the respective subgraphs induced by the vertex partitions of the intersection graph<sup>1</sup>.

#### 6.2.1.1 An iterating greedy algorithm

The iterating greedy algorithm for the determination of edge layouts is conceptually very simple. It enumerates the edges of an input graph  $\mathcal{G}$ , and for each edge it determines whether the total number of crossings will be reduced if the layout page of the edge is changed. If this is the case, it repeats the process after the enumeration of the remaining edges has been completed. This procedure is repeated until no more edges may be shifted to pages that are opposite to their layout pages so as to lower the total number of crossings in the configuration. This process must terminate, since the number of crossings is bounded from below by  $\nu(\mathcal{G})$ .

The main while-loop spanning lines 2–12 of Algorithm 6.3 is executed until convergence (*i.e.*, no more edges may be shifted to opposite pages to improve the crossing number bound) is achieved. The variable  $c$  is consulted at each iteration to determine whether this has occurred. At line 1,  $c$  is set to the boolean value “FALSE”, so that at least one iteration of the loop is completed — after all, it must at least verify that no edges need be placed on alternative pages.

For an iteration, the variable  $\nu'$  keeps track of the total number of crossings in which edges are involved (because the total number of crossings for each edge is counted, crossings are counted

---

<sup>1</sup>This generalizes naturally to cases where there are more than two pages — there are simply as many partitions as there are pages. However, such cases are not considered in this thesis.

**Algorithm 6.3 GreedySide**

**Input:** An intersection graph  $\mathcal{X}_{G,A}$  and a vector  $\mathbf{p}$  which maintains the pages assigned to intersection-vertices.

**Output:** The number of crossings for the edge layout found, along with the updated vector  $\mathbf{p}$  which reflects the layout.

```

1:  $c \leftarrow \text{FALSE}$ 
2: while  $c \neq \text{TRUE}$  do
3:    $\nu' \leftarrow 0$ 
4:    $c \leftarrow \text{TRUE}$ 
5:   for all  $v \in V(\mathcal{X}_{G,A})$  do
6:      $p' \leftarrow \mathbf{p}_v$ 
7:      $\nu' \leftarrow \nu' + \text{ChooseBestPage}(\mathbf{p}, \mathcal{X}_G, v)$ 
8:     if  $p' \neq \mathbf{p}_v$  then
9:        $c \leftarrow \text{FALSE}$ 
10:    end if
11:  end for
12: end while
13: return  $\nu'/2$ 

```

**Implementation note:** An implementation of this algorithm in C is given in § B.1.1.

twice). It is set to 0 at line 3, before enumeration of the edges starts (which occurs between lines 5–11).

Once inside the main loop, it is assumed that convergence will occur, until proven otherwise. Thus,  $c$  is set to “TRUE” at line 4 in accordance with this assumption. The vector  $\mathbf{p}$  keeps track of the pages to which intersection-vertices are assigned. The procedure `ChooseBestPage` returns the number of crossings in which the intersection-vertex  $v$  is involved, although it modifies the vector  $\mathbf{p}$  if the number of crossings in which the intersection-vertex  $v$  is involved may be reduced. The value that it returns is added to the total number of crossings counted in the current iteration of the main loop at line 7. Due to the fact that it must be determined after the call to `ChooseBestPage` whether  $v$  was moved to a different page (to ascertain whether convergence will occur) the original page on which  $v$  occurs is stored into the variable  $p$  at line 6. The test at line 8 verifies whether the page on which the intersection-vertex occurs has changed, and if so, it sets the variable  $c$  to “FALSE” to indicate that convergence has not been achieved.

This algorithm produces good results, as will be demonstrated in the next chapter. Perhaps part of the reason is due to the way that the tabu algorithm (which is responsible for vertex arrangements, as will be described later) “drives” the layout algorithms. At each iteration, the tabu search algorithm scans every vertex  $v$  on the spine, and it attempts to move  $v$  to every other position on the spine in turn, evaluating the feasibility of each such move by means of determining an edge layout for the resulting vertex arrangement. The vector  $\mathbf{p}$ , containing the pages assigned to intersection-vertices, is not modified between evaluations of such vertex moves. One of the important properties of the intersection graph is exactly that when a single vertex is moved on the spine, very little of the intersection graph itself needs to be altered, as described earlier. This means that every time the tabu algorithm attempts to insert a vertex at a different position on the spine, it only updates the intersection graph partially, and because it leaves the page vector unmodified, most of the layout information remains intact. It seems that a previous high quality layout improves the chances that a new layout of a high quality will be found (since the layouts for the edges that are incident to vertices maintaining their positions on the spine are presumably better than a random layout for these edges).

This algorithm may be compared to some of Cimikowski’s [Cim02] simpler algorithms. However, none of these simple algorithms iterate in this fashion to reach a point of convergence<sup>2</sup>.

### Algorithmic complexity of the GreedySide algorithm

The computational running time of the procedure `ChooseBestPage` on line 7 depends on the degree of the crossing-vertex  $v$  — in other words, on the number of edges that cross the edge  $e$  corresponding to  $v$ . This is true, because all edges that cross  $e$  need to be considered in order to determine on which page  $e$  should be placed. For an input graph  $\mathcal{G}$ , in the worst case, an edge is crossed by every other edge in  $\mathcal{G}$ , so that the worst case complexity of `ChooseBestPage` may be measured as  $O(|E(\mathcal{G})|)$ , and since the loop (lines 5–10) in which this computation is located is executed for every crossing-vertex of the intersection graph of  $\mathcal{G}$ , the total worst case complexity for the loop is  $O(|E(\mathcal{G})|^2)$ . Now, since not every edge in a graph can cross every other edge (after all, adjacent edges never cross in book layouts), there is a possibility that the bound  $O(|E(\mathcal{G})|^2)$  is overly pessimistic. Thus, it must be shown that there exists a graph for which this bound is attained. Consider a two-page layout of a complete graph  $\mathcal{K}_n$  — for every choice of four vertices in  $\mathcal{K}_n$ , a crossing may be obtained (since there will be a pair of alternating edges to which these vertices are incident). The total number of alternating edges must therefore be counted as  $\binom{n}{4}$ . Now, since the graph has  $\binom{n}{2}$  edges, the number of alternating edges expressed in terms of the number of edges in  $\mathcal{K}_n$ , is  $O(|E(\mathcal{K}_n)|^2)$ .

It is rather more difficult to put forth a bound on the number of times that the main loop (lines 2–12) would be executed. In a worst case scenario, the number of crossings during each iteration of this main loop would decrease only by 1. Furthermore in a worst case, the initial crossing configuration would be such that it realises the maximum number of crossings for any layout in the intersection graph  $\mathcal{X}_{\mathcal{G},A}$ ; denote this value by  $\nu_{\mathcal{X}_{\mathcal{G},A}}^{(\max)}$ . Denote the minimum value for the number of crossings that are permitted by  $\mathcal{X}_{\mathcal{G},A}$  as  $\nu_{\mathcal{X}_{\mathcal{G},A}}$ . Thus, at most  $\nu_{\mathcal{X}_{\mathcal{G},A}}^{(\max)} - \nu_{\mathcal{X}_{\mathcal{G},A}}$  can be performed by the main loop, and therefore, the worst case running time of the algorithm would be  $O((\nu_{\mathcal{X}_{\mathcal{G},A}}^{(\max)} - \nu_{\mathcal{X}_{\mathcal{G},A}})|E(\mathcal{K}_n)|^2)$ . It should be noted that  $\nu_{\mathcal{X}_{\mathcal{G},A}}^{(\max)} - \nu_{\mathcal{X}_{\mathcal{G},A}} < \binom{|V(\mathcal{G})|}{4}$ . This is because the maximum crossing number of the complete graph in standard one-cross form is  $\binom{|V(\mathcal{G})|}{4}$  — this may be seen drawing all the pages of the complete graph on a single page of a book, where, for every choice of four vertices, there is a pair of crossing edges.

With this said, the algorithm seems to perform quite well in practice. Empirical results for the average number of iterations required for graphs of varying sizes may be found in the next chapter. In the worst case layout, all edges are placed on the same page. In this situation, when an edge is shifted to the opposite page, the number of crossings is likely to decrease by far more than 1; this is especially true for dense graphs, where each edge is crossed by a number of other edges. It is therefore unlikely that the worst case running time will become problematic.

#### 6.2.1.2 The Cimikowski–Shope neural network algorithm

There is not much that needs to be said about the Cimikowski–Shope algorithm, since the theory for the algorithm was discussed in § 4.3.2.2. This section merely provides a concrete reference implementation<sup>3</sup> of the algorithm which also employs the notion of intersection graphs.

<sup>2</sup>His neural network algorithm does continue until convergence is reached. However, the intention is not to compare this simple iterating algorithm to the neural network algorithm.

<sup>3</sup>This implementation ought to be more understandable than its C, or Python computer implementations, which may be found in § B.1.2. References are made to the notation and concepts in § 4.3.2.2, and it is recommended that this section is read before attempting to understand this algorithm.

**Algorithm 6.4**  $b_v^{(\downarrow)}$  or  $b_v^{(\uparrow)}$ **Input:** An intersection graph  $\mathcal{X}$ , a vertex  $v \in V(\mathcal{X})$ , and a neural page configuration  $\mathbf{V}$ .**Output:** The number of crossings that  $v$  would be involved in, in a page layout corresponding to  $\mathbf{V}$ .

- 1:  $\nu^* \leftarrow 0$
- 2: **for all** incident edges  $\{v, u\}$  of  $v$  in  $\mathcal{X}$  **do**
- 3:    $\nu^* \leftarrow \nu^* + \mathbf{V}_u$
- 4: **end for**
- 5: **return**  $\nu^*$

**Implementation:** An implementation of this algorithm may be found in § B.1.2.

The main algorithm is called **Cimikowski-Shope**; its main loop spans lines 6–33 in Algorithm 6.5. This loop is executed until convergence is achieved, or until a specified maximum number of iterations  $t_{\max}$  have been completed. As with the **GreedySide** algorithm, the variable  $c$  indicates convergence, and as with **GreedySide**, it is initially set to “FALSE” at line 5, so that the main loop will be executed at least once. Furthermore, for each iteration of the main loop, it is initially assumed that convergence will occur, until a counter-example is found — hence,  $c$  is set to “TRUE” at the start of the main loop, which is line 7. The variable  $t$  maintains the iteration count, and it is updated at line 8.

The vectors  $\mathbf{U}^{(\uparrow)}$  and  $\mathbf{U}^{(\downarrow)}$  represent the function values of the like-named functions in § 4.3.2.2; since the boundary conditions require that these functions have random values at time  $t = 0$  for all edges, they are thus assigned values in the first loop, which spans lines 1–4. The binary valued vectors  $\mathbf{V}^{(\uparrow)}$  and  $\mathbf{V}^{(\downarrow)}$  indicate whether an edge should be drawn on respectively the upper or lower page, by having the respective entries for the edge (intersection-vertex) set to 1 (which, of course, means that ambiguity over page on which the edge occurs arises when its entries in both vectors are 1, or when both are 0). All of these vectors are indexed by means of the intersection-vertices of the intersection graph. For each iteration of the main loop, the vectors  $\mathbf{V}^{(\uparrow)}$  and  $\mathbf{V}^{(\downarrow)}$  have to be updated in accordance with the function values contained in the vectors  $\mathbf{U}^{(\uparrow)}$  and  $\mathbf{U}^{(\downarrow)}$ . This is done in the loop spanning lines 9–20.

The heart of the algorithm is the loop spanning lines 22–32. It is here that the function values contained in  $\mathbf{U}^{(\uparrow)}$  and  $\mathbf{U}^{(\downarrow)}$  are updated. The **if** statement between lines 23–25 is only executed if there is an intersection-vertex that is not unambiguously assigned to a page. If this is the case, then the algorithm has not yet converged, and the value of  $c$  is set to boolean value “FALSE.” The function values are updated by Euler’s method at lines 27 and 28.

The values of  $a_v$  and  $c_v$  are computed as they are in § 4.3.2.2. The way in which  $b_v^{(\downarrow)}$  and  $b_v^{(\uparrow)}$  are computed deserves some attention. The definitions provided for these variables in § 4.3.2.2 do not yield particularly efficient methods for computing their values; the method described here instead uses the intersection graph. Essentially, the value of  $b_v^{(\uparrow)}$  is equal to the number of edges that  $v$  would cross if it were to be drawn on the upper page. It will only cross those edges that have their entries in the vector  $\mathbf{V}^{(\uparrow)}$  set to 1. Therefore, if all of the vertices adjacent to  $v$  are enumerated, then taking the sum of their values in  $\mathbf{V}^{(\uparrow)}$  yields the number of crossings in which  $v$  would be involved, if drawn on the upper page. The same reasoning holds for the vector  $\mathbf{V}^{(\downarrow)}$ , for drawings on the lower page.

Algorithm 6.4 is responsible for calculating either  $b_v^{(\uparrow)}$  or  $b_v^{(\downarrow)}$ . The input vector  $\mathbf{V}$  corresponds either to  $\mathbf{V}^{(\uparrow)}$  or to  $\mathbf{V}^{(\downarrow)}$ , depending on whether respectively  $b_v^{(\uparrow)}$  or  $b_v^{(\downarrow)}$  is being considered. The loop between lines 2–4 enumerates the vertices adjacent to the intersection-vertex  $v$  under consideration, and the variable  $\nu^*$  maintains the sum of the crossings in which  $v$  is involved.

**Algorithm 6.5** Cimikowski-Shope

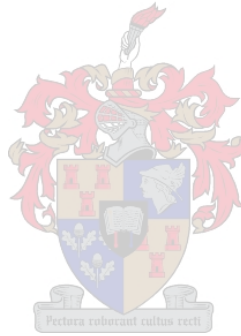
**Input:** An intersection graph  $\mathcal{X}_{\mathcal{G},A'}$ , neural network parameters  $A, B, C, dt, t_{\max}$  and random initializer  $\omega$ .

**Output:** The variable  $c$  to indicate convergence, and the vectors  $\mathbf{V}_v^{(\uparrow)}$  and  $\mathbf{V}_v^{(\downarrow)}$  from which the page assignments for edges may be computed.

```

1: for all  $v \in V(\mathcal{X}_{\mathcal{G},A})$  do
2:    $\mathbf{U}_v^{(\uparrow)} \leftarrow$  random uniform value in  $[-\omega, 0)$ 
3:    $\mathbf{U}_v^{(\downarrow)} \leftarrow$  random uniform value in  $[-\omega, 0)$ 
4: end for
5:  $c \leftarrow$  FALSE
6: while  $c \neq$  TRUE and  $t < t_{\max}$  do
7:    $c \leftarrow$  TRUE
8:    $t \leftarrow t + 1$ 
9:   for all  $v \in V(\mathcal{X}_{\mathcal{G},A})$  do
10:    if  $\mathbf{U}_v^{(\uparrow)} > 0$  then
11:       $\mathbf{V}_v^{(\uparrow)} \leftarrow 1$ 
12:    else
13:       $\mathbf{V}_v^{(\uparrow)} \leftarrow 0$ 
14:    end if
15:    if  $\mathbf{U}_v^{(\downarrow)} > 0$  then
16:       $\mathbf{V}_v^{(\downarrow)} \leftarrow 1$ 
17:    else
18:       $\mathbf{V}_v^{(\downarrow)} \leftarrow 0$ 
19:    end if
20:  end for
21:
22:  for all  $v \in V(\mathcal{X}_{\mathcal{G},A})$  do
23:    if  $\mathbf{V}_v^{(\uparrow)} = \mathbf{V}_v^{(\downarrow)}$  then
24:       $c \leftarrow$  FALSE
25:    end if
26:
27:     $\mathbf{U}_v^{(\uparrow)} \leftarrow \mathbf{U}_v^{(\uparrow)} + [a_v A + B(-b_v^{(\uparrow)} + b_v^{(\downarrow)}) + c_v C] dt$ 
28:     $\mathbf{U}_v^{(\downarrow)} \leftarrow \mathbf{U}_v^{(\downarrow)} + [a_v A + B(-b_v^{(\downarrow)} + b_v^{(\uparrow)}) + c_v C] dt$ 
29:
30:     $\mathbf{U}_v^{(\uparrow)} \leftarrow \mathbf{U}_v^{(\uparrow)} / \max\{1, |\mathbf{U}_v^{(\uparrow)}|\}$ 
31:     $\mathbf{U}_v^{(\downarrow)} \leftarrow \mathbf{U}_v^{(\downarrow)} / \max\{1, |\mathbf{U}_v^{(\downarrow)}|\}$ 
32:  end for
33: end while
34: return  $c, \mathbf{V}^{(\downarrow)}, \mathbf{V}^{(\uparrow)}$ 

```



**Implementation:** An implementation of this algorithm may be found in § B.1.2.

The final two lines of importance are lines 30 and 31 in Algorithm 6.5. They simply ensure that the values of  $\mathbf{U}^{(\uparrow)}$  and  $\mathbf{U}^{(\downarrow)}$  remain within the range of  $[-1, 1]$ , by “capping” the maximum value to 1, and the minimum to  $-1$ . Finally, upon completion at line 34, the algorithm returns the variable  $c$ , along with the vectors  $\mathbf{V}^{(\uparrow)}$  or  $\mathbf{V}^{(\downarrow)}$ , from which the page values of intersection-vertices may be computed (if convergence has occurred).

If convergence does not occur, then another simple optimization algorithm may simply be ap-

plied after the execution of the neural algorithm. For this thesis, the **GreedySide** algorithm is applied under such circumstances. In fact, it is always applied, since even if the neural algorithm **Cimikowski-Shope** converges, local improvements might still be possible.

### Algorithmic complexity of the Cimikowski-Shope algorithm

For an input graph  $\mathcal{G}$ , the initial assignment of random function values to  $\mathbf{U}^{(\uparrow)}$  and  $\mathbf{U}^{(\downarrow)}$  takes  $O(|V(\mathcal{X}_{\mathcal{G},A})|)$  time, or equivalently  $O(|E(\mathcal{G})|)$  time, since an assignment must be made for each vertex in the intersection graph, and each assignment takes constant time.

The main loop, which spans lines 6–33 is executed at most  $O(t_{\max})$  times. Inside this main loop, the first loop between lines 9 and 20 is executed for each crossing–vertex, but the operations within it may be verified to require constant execution time. Thus, the time complexity of this inner loop is  $O(|V(\mathcal{X}_{\mathcal{G},A})|)$ . For the second loop inside of the main loop (lines 22–32), all operations except for those on lines 27 and 28 execute in constant time. As with the procedure **ChooseBestPage** in the algorithm **GreedySide** (Algorithm 6.3, § 6.2.1.1), the time taken for the operations on lines 27 and 28 depends on the degree of each crossing–vertex  $v$ ; this may be seen in Algorithm 6.4 from the fact that the adjacent vertices of  $v$  all need to be considered in order to compute the number of crossings that  $v$  would incur when it is to be drawn on either the upper or lower page (depending on whether respectively  $b^{(\uparrow)}$  or  $b^{(\downarrow)}$ ). Thus as in the algorithm **GreedySide**, the worst case computational running time for the second loop is of the order  $O(|E(\mathcal{G})|^2)$ .

Within the main loop, the running time of the second loop (lines 22–32) dominates that of the first loop (lines 9–20), and therefore each iteration of the main loop executes in  $O(|E(\mathcal{G})|^2)$  time. Therefore, the overall worst case running time of the algorithm is  $O(t_{\max} |E(\mathcal{G})|^2)$ .

#### 6.2.1.3 A hybrid genetic and local optimization algorithm

Darwin’s ground breaking book *On the Origin of Species* [Dar59], which was first published in 1859, detailed what was to become the foundation for the modern theory of evolution. His innovation was not the concept of evolution per se<sup>4</sup>, but the concept of *natural selection*. Natural selection refers to the fact that certain individuals in a population are more likely to survive and consequently to reproduce, than others, due to individual traits that work to their advantage, or in the parlance of popular evolution theory, traits that make them *fitter* (from here the oft quoted phrase “survival of the fittest”). Natural selection occurs when the following three properties are present in a population of individuals:

1. individuals in the population can *make copies* of themselves (typically by means of the “mating” of two individuals),
2. the process of copying is *imperfect*,
3. the copying errors influence the ability of offspring to *survive* and make copies of themselves.

Thus, the fittest individuals are more likely to survive, and to have their genes transferred to the next generation. The errors that occur in the copying process, ensure that locally optimal chromosomes are avoided. This process is not confined only to living entities. Sets of candidate

<sup>4</sup>At least part of the academic community involved in biological research believed that a process of evolution occurred in living species, although they lacked a mechanism with which to explain its action (page 25, [EH01]).

solutions for optimization problems may be made to display the same properties, if a mechanism for producing new offspring is provided. It is exactly this observation that led to the development of *genetic algorithms*.

### Algorithmic aspects

Not surprisingly, the terminology used for genetic algorithms is derived from the nomenclature of biology. A set of solutions is called a *population*; its individuals are referred to as *chromosomes*, which consist of *genes*. A genetic algorithm proceeds through successive *generations* of a population. Offspring are created by means of either *crossover* or by *mutation*. The exact semantics of these two operations may differ widely from problem to problem, but a basic description using bit strings should serve as a sufficient example.

Let  $X = x_1x_2 \cdots x_t$  and  $Y = y_1y_2 \cdots y_t$  represent two binary bit strings of length  $t$  each. Two offspring may be produced from the pair, using the traditional *crossover operator*, which works as follows: a random index  $i$ , called the *crossover point*, is chosen, such that  $1 < i < t$ , then the two offspring are defined as  $O_1 = x_1 \cdots x_i y_{i+1} \cdots y_t$  and  $O_2 = y_1 \cdots y_i x_{i+1} \cdots x_t$ . This operation is depicted in Figure 6.6, where the crossover point is  $i = 4$ , and is depicted by the X between the two strings.

Mutation is a transition that is applied to a single chromosome. In the case of bit strings, a random point is chosen, and the bit is flipped to its opposite value. Mutation is typically applied with a certain probability (say 0.2%) to the offspring generated by a crossover.

$$\begin{array}{cccccccccccccccc} X = & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & \rightarrow & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & = O_1 \\ & \\ & \\ Y = & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \rightarrow & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & = O_2 \end{array}$$

Figure 6.6: An illustration of the operation of the crossover operator.

In the classical genetic algorithm, the population size is kept constant, and the entire population is replaced at each iteration by the offspring born generally of the fittest genes. There are a myriad of ways to achieve this goal, but only the method used in this thesis is discussed here. The method of *tournament selection* is a popular and effective selection technique for genetic algorithms. Consecutive (non-intersecting) sets of  $n$  individuals from the population are considered at a time, and the best of each set is chosen. If the population has size  $p$ , then a total of  $p$  individuals have to be selected for mating, since each pair of parents produces two offspring. This means that  $n \times p$  individuals will be selected during the course of choosing parents, with the effect that selection will “wrap around” and recommence the with the selection of the first individuals in the population. When such “wrapping around” occurs, it is therefore beneficial to randomize the order of the chromosomes in the population, so that not all of the same chromosomes will be chosen as in the previous enumeration. When the  $p$  desired parents have been selected, they are mated sequentially with one another in pairs. After mating, the children are mutated with a low probability.

Pseudo-code for the genetic algorithm described above is shown in Algorithm 6.6. The code in the algorithm is admittedly quite vague, but this is due to the fact that many aspects of the algorithm are highly dependent on the structure of the problem to be solved. The variable  $c^*$  maintains the best chromosome that has been found in all generations during the algorithm’s course of execution, whilst the variable  $f$  contains the fitness value of the chromosome represented by  $c^*$ . Line 1 of the algorithm may translate into a host of different instructions; typically it will consist of a simple loop enumerating the population and executing the fitness function for each



of the constituent chromosomes. This initial computation is necessary, because the fitness values must be known at the start of each generation of chromosomes. The main loop, of which each iteration is a generation, spans lines 2–17. The condition that is tested by the loop at line 2 is simply called “continue,” to reflect the fact that there are a variety of ways in which to terminate a genetic algorithm. Probably the most common methods are to terminate

1. after a specified number of generations,
2. if, for a specified number of generations, no individuals of a higher fitness than the best individual over all generations have been found,
3. when a certain fitness threshold has been achieved.

The first of these was used for the genetic algorithm implementation in this thesis, since it gives an assurance of maximum running time, where the second may run for an arbitrarily long time, and the third may never terminate (in fact, the third should in all sensibility be combined with a condition such as the first in the list). At line 3, the  $|P|$  fittest individuals are stored in the variable  $M$  (where the fitness depends on the manner of selection). The loop between lines 4–17 considers all chromosomes of  $M$  in consecutive pairs, as they are to be mated. The creation of offspring only really occurs at lines 5 and 6. Again, the fitness of the individuals is required at the next generation when parents of that generation are to be selected, and so their fitness values<sup>5</sup> are computed at line 7. It may be seen from both conditional **if** structures, spanning lines 8–15, that their function is only to determine whether the two respective new offspring are fitter than the previously best found chromosome, and if so, to store whichever fitter offspring into the variable  $c^*$ , and its fitness into  $f$ . The new generation is stored into the population variable  $P'$ , where the offspring are appended in turn to  $P'$  at line 16, and finally after the creation of all offspring,  $P$  is replaced by  $P'$  at line 18.

Genetic algorithms are typically applied to difficult optimization problems, with solutions spaces that are too large to enumerate sequentially. The preservation of good genes progressively leads to the progressive improvement of good solutions whilst the mutation ensures that local optima may be avoided, thereby driving the search into unexplored regions of the solution space. A good balance between these two traits of “zooming in on good solutions” and “scouting for new solutions” is the hallmark of a good heuristic algorithm, and it is also found in the form of “aspiration” and “diversification” in the Tabu search method, described later.

### Applying genetic algorithms to the edge layout problem

Genetic algorithms are the most effective for problems where the combination of sub-solutions from two existing solutions may yield a new, better (fitter) solution. These problems display what might be called the “good to near-optimal sub-solution trait”, which may loosely be thought of as a relaxation of the optimal sub-solution trait for which *dynamic algorithms* are often useful.

Now consider the edge layout problem. A seemingly sensible encoding for the problem would be to use the bit string scheme from the previous section, such that each gene (bit) corresponds to the page layout for a unique edge in the input graph  $\mathcal{G}$  (for example a value of 0 would indicate that the corresponding edge would be drawn on the upper page, and a value of 1 that it would be drawn on the lower page).

---

<sup>5</sup>The fitness values are generally stored in an associative array (the association being between chromosomes and fitness values), or inside the structure which contains the chromosome.

**Algorithm 6.6 GeneticAlgorithm****Input:** A population  $P$  of chromosomes.**Output:** The updated population  $P$  and the fittest chromosome  $c^*$ .

```

1: Evaluate the fitness of each chromosome in  $P$ 
2: while continue do
3:    $M \leftarrow \{|P| \text{ fittest parents selected from tournaments}\}$ 
4:   for all consecutive pairs of elements  $p_1, p_2$  in  $M$  do
5:      $c_1, c_2 \leftarrow \text{CrossOver}(p_1, p_2)$ 
6:     possibly mutate  $c_1$  and  $c_2$ 
7:     compute  $\text{Fitness}(c_1)$  and  $\text{Fitness}(c_2)$ 
8:     if  $\text{Fitness}(c_1) > f$  then
9:        $f \leftarrow \text{Fitness}(c_1)$ 
10:       $c^* \leftarrow c_1$ 
11:    end if
12:    if  $\text{Fitness}(c_2) > f$  then
13:       $f \leftarrow \text{Fitness}(c_2)$ 
14:       $c^* \leftarrow c_2$ 
15:    end if
16:     $P' \leftarrow P' \cup \{c_1\} \cup \{c_2\}$ 
17:  end for
18:   $P \leftarrow P'$ 
19: end while

```

**Implementation:** An implementation of this algorithm may be found in § B.1.3.

Unfortunately, this encoding scheme has the deficiency that sub-solutions cannot be isolated effectively into sub-strings in the bit string, so that the mating of two chromosomes generally would not yield a better chromosome. To understand why this is so, it must be noted that a single edge  $e$  may be crossed by many other edges. Now, if the bits representing these other edges (or rather the pages for these edges) are all placed adjacent to the bit representing  $e$ , then this entire substring would need to be preserved if the crossing configuration for  $e$  is to be preserved. But, of course, these edges generally will not cross only  $e$ , but also some other edges in  $\mathcal{G}$ . Clearly, it would not be possible to pack together bits of all edges that cross one another. The solution is “dispersed” across the bit string, and the recombination of two substrings  $X$  and  $Y$  generally will not yield an improvement, since the quality of the chromosome  $A$  of which  $X$  is a substring is due to the way that some bits in  $X$  depend on the values of bits that are not in  $X$  — these bits would probably be different in  $Y$ , causing new crossings, and *vice versa* for  $Y$ .

It is not possible to find an encoding which would resolve this problem entirely, but its effect may be minimized. In an encoding, edges that occur entirely to the right of some edge  $e$  should preferably be placed to the right of the encoding of  $e$  in the chromosome; the same considerations hold for edges to the left of  $e$ . As for edges that cross  $e$ , it would be preferable to keep them as close to  $e$  as possible, whilst considering that this property should also hold for all other edges.

Consider the book layout of  $\mathcal{K}_8$  shown in Figure 6.7(a) — for each edge  $e = \{u, v\}$  in the drawing, one could assign a coordinate to  $e$  by taking the average of the positions of its two incident vertices  $u$  and  $v$  — *i.e.*, if  $u$  is at position  $x$ , and  $v$  at position  $y$ , then  $e$  is assigned the coordinate  $(x + y)/2$ . If such coordinates are used to encode page numbers into a chromosome, then edges which occur entirely to the right, or entirely to the left of an edge  $e$  (*i.e.*, those edges that do not intersect  $e$ ) might be represented by bits in the chromosome which occur respectively either to the right or to the left of the bit corresponding to  $e$ . Edges which are likely to cross  $e$

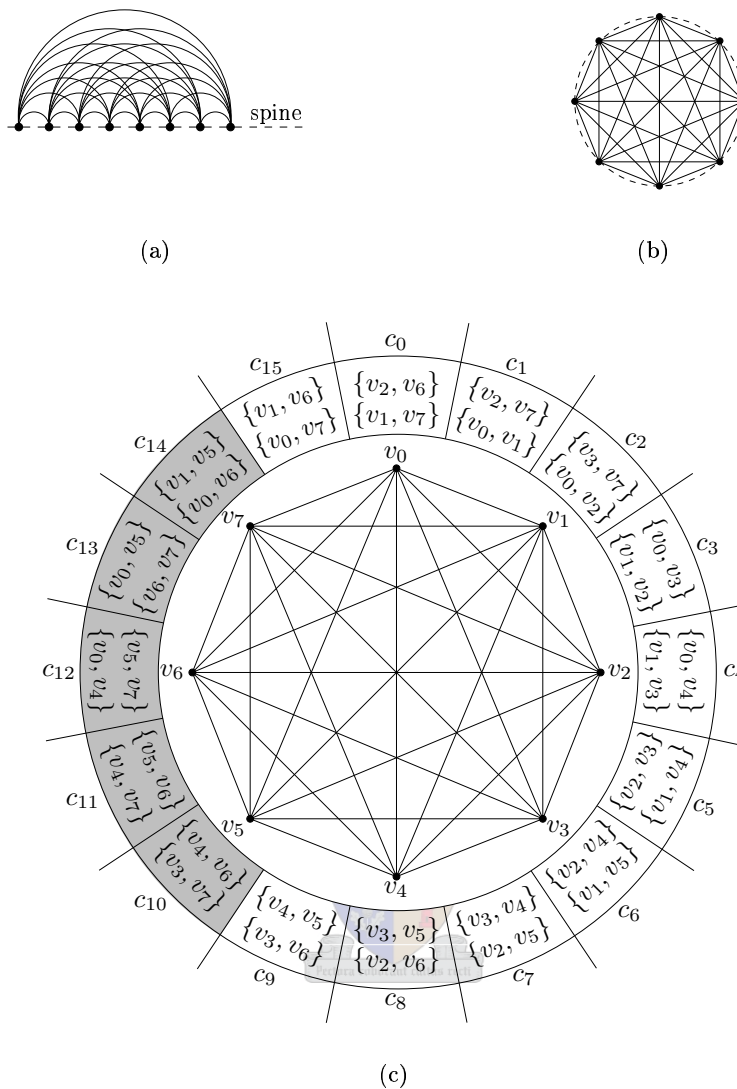


Figure 6.7: A change in perspective from a book layout to a circular layout enables a “cell” encoding, which is more effective with the genetic crossover operator than a simple bit string encoding of edge positions.

also occur “close” to  $e$  (although this may be seen more clearly in the circular encoding, which is described in the following paragraphs). Clearly, a number of edges may occupy the same coordinate — in this case they are treated as an entity, with the convention that a crossover cannot be made in the middle of an entity.

A better encoding scheme will be presented presently, but first a change in perspective is required. From the book layout depicted in Figure 6.7(a), it might be concluded that more crossings appear in the middle, and one could assume that this would influence the encoding. However, since book layouts are equivalent to circular layouts (§ 3.1.3.2); the layout depicted in Figure 6.7(b) is equivalent to the layout in Figure 6.7(a); however in Figure 6.7(b), the crossings seem to be “distributed” around the circle.

Now, instead of assigning the coordinates relative to the position of the midpoint of an edge on

the spine, coordinates are assigned by taking the average of the *minimum* distance between two vertices on the circle. In Figure 6.7(c), there are sixteen “cells,” labelled  $c_0$  to  $c_{15}$ ; each of these cells represents a unique coordinate (there are 8 vertices, and a total of 16 coordinates due to the fact that an average of an even and an odd vertex position gives a position between two vertices); edges which have the same coordinate are grouped together in a cell.

The cell units now represent genes, and this scheme is sufficient to avoid the problem of a crossover occurring in the middle of a group of genes which represent edges with the same coordinate. In Figure 6.7(c), the cells that are coloured gray are supposed to illustrate a section of another chromosome that is mated with the white cells.

### Issues surrounding circular encoding

The first problem that arises with such a circular chromosome encoding scheme is a relatively minor one. When the input graph has an even number of vertices, as is the case for  $\mathcal{K}_8$  shown in Figure 6.7(c), then edges for which the incident vertices are equally far apart ( $\{v_0, v_4\}$ ,  $\{v_1, v_5\}$ ,  $\{v_2, v_6\}$  and  $\{v_3, v_7\}$ ) will appear in two cells, since these edges may ambiguously have two coordinates. A simple and pragmatic solution is simply to use only the smallest coordinate for an edge. Another point to raise is that edges which span two vertices which occur adjacent to one another on the circle have been included; however, they can never make a difference to an edge layout, since they are not crossed, and they may therefore be ignored.

The second problem is rather more fundamental, and has to do with the fact that there is still some “dispersion” of the solution present in circular encodings. For example, consider again Figure 6.7(c), and let the greyed cells be a sub-solution to be mated with the white cells. Clearly the edges in cells  $c_{10}$  and  $c_{14}$  in the greyed region, and the edges in the cells  $c_9$  and  $c_{15}$  in the white region also depend on edges outside of their regions for their crossing configurations.

It does not seem sensible to ignore this, since the values corresponding to these edges are not really part of a good sub-solution. The approach taken for this thesis was to apply a simple heuristic such as the **GreedySide** to these “boundary” edges. Because of the application of such a local optimization method, the algorithm is not a genetic algorithm in the purest sense. Nevertheless, such a pragmatic approach is required to mitigate dispersion problems.

### Implementation of the crossover operator

The implementations of most genetic algorithms are simple, and many follow the same general structure as presented in the pseudo-code of Algorithm 6.6. However, it is important to give a more precise specification of the crossover operator. The cell encoding described in the previous sub-sections may be implemented as a list of lists — the main list is indexed by the cell coordinate, and the contents of each list position is a list containing the crossing-vertices for that coordinate. It is a simple matter to mate two such chromosomes — the only difference between this model and the bit string model is that lists are copied to the offspring instead of bits.

The one point which deserves attention, is the fact that the edges of a chromosome that have incident vertices occurring beyond the crossover point (*i.e.*, the edges that cause the solution “dispersion”) need to be identified, so that they may be assigned pages by a heuristic method. A simple approach was taken for this thesis — in each list (*i.e.*, each cell) of the list of lists, all the crossing-vertices are enumerated, and those for which the corresponding edge has an incident vertex occurring outside the crossover point, are marked. After they have all been marked, a modified version of the algorithm **GreedySide** (Algorithm 6.3, § 6.2.1.1) is applied to all the marked crossing-vertices.

### Algorithmic complexity of the genetic algorithm

From a population  $P$ , the time taken to select chromosomes for the purposes of mating is proportional to the size of  $P$ , and thus of the complexity order  $O(|P|)$ . For each offspring, the crossover operation takes  $O(|E(\mathcal{G})|)$  time for an input graph  $\mathcal{G}$ , since the list of lists constituting a chromosome contains an entry for each edge, and the assembly of a new chromosome is by the process of copying the various list entries from its parents. The time taken to identify edges which straddle the two crossover points takes  $O(|E(\mathcal{G})|)$  time, since all entries in the list of lists are enumerated. In the worst case, there will be  $O(|E(\mathcal{G})|)$  edges that straddle the crossover point, and due to the fact that the algorithm **GreedySide** is used, the running time is  $O(c|E(\mathcal{G})|^2)$ , where  $c$  is the difference between the maximum and minimum number of crossings realizable by a two–page layout, using the current vertex arrangement. However, as noted in § 6.2.1.1,  $c$  is generally not very large, and in practice  $c \ll |E(\mathcal{G})|$ . Finally, the time that it takes to compute the fitness of the offspring is the time taken to compute the crossing number that would result from a particular partitioning of the vertices of the intersection graph, which has been shown to take  $O(|E(\mathcal{G})|^2)$  time.

Thus, most time is likely to be spent in the crossover, which has a complexity of  $O(|E(\mathcal{G})|^2) + O(c|E(\mathcal{G})|^2) = O(c|E(\mathcal{G})|^2)$ . Two crossover operations are performed for each pair of parents, to produce two offspring. Therefore, the total time taken to create a new generation is  $O(c|P||E(\mathcal{G})|^2)$ , and the total running time of the algorithm is  $O(t_{\max} c |P||E(\mathcal{G})|^2)$ .

#### 6.2.2 Vertex arrangement heuristics

The vertex arrangement problem has not received quite as much attention in the literature as the edge layout problem. The problem of vertex arrangements has been studied extensively for the crossing minimization of  $k$ –layered drawings [MM97, BML00, Mut01, JLMO97, SSSV97b], because, as noted in § 3.1.3.4, the number of crossings depend only on the orderings of the vertices on the various levels. Such solutions are, however, of little use for the general crossing number problem, except for providing upper bounds to the crossing number values of layered graphs. By using graph bisection (§ 4.2.2.4), it is possible to obtain a vertex arrangement for a graph  $\mathcal{G}$  that is within a  $\log |V(\mathcal{G})|$  factor of its crossing number  $\nu(\mathcal{G})$ , but one cannot be sure of the constant factor involved in the bound, and it will almost certainly be possible to improve such a layout afterwards.

The heuristic method known as the tabu search method due to Glover [Glo86], which is introduced in this section, intelligently enumerates the search space by combining short–term “greedy,” or “aspiration” behaviour, with longer term “diversification.” This method may be seen as generalization of the *neighbourhood search* method, which is an entirely greedy. In practice, this method performs well with permutation based problems ([Ree93]), hence its choice for the computation of vertex arrangements.

At the end of the section, pre–optimization methods for the generation of initial vertex arrangements are discussed. The first method is based on Nicholson’s heuristic (§ 4.3.2.2) and the second method is essentially the Hamiltonian cycle heuristic that was used by Cimikowski [Cim02] (§ 4.3.2.2).

#### Neighbourhood Search

The terminology required for the methods of neighbourhood search and tabu search are introduced in this subsection. Combinatorial optimization algorithms rarely peruse their search spaces

randomly. Instead, it is typical to have a well defined transition by which a feasible solution  $A$  may be altered in order to form another solution  $A'$ ; the algorithms generally move from one solution to the next by such transitions (in a genetic algorithm, this transition is defined as the crossover of two chromosomes). For example, in permutation problems, such a transition might be the swapping of a pair of elements. In the parlance of tabu search theory, a transition or alteration is called a *move*. The set of solutions which may be obtained from a solution  $A$  by all moves defined for such a solution, is called the *neighbourhood* of  $A$ , and is denoted  $N(A)$ .

The neighbourhood search method is a local optimization method. The idea is quite simple in the context of the crossing number problem: with a vertex arrangement  $A$ , the arrangement  $A' \in N(A)$  which permits the edge layout with the fewest crossings, is chosen; if  $A'$  permits fewer crossings than  $A$ , then  $A$  is replaced by  $A'$  and the process is repeated; otherwise the search is terminated. It is clear that this process will converge to a local optimum; the greedy behaviour of this approach is manifested in the fact that it always makes the most improving move.

Algorithm 6.7 is an example of a simple neighbourhood search algorithm for the problem of determining vertex arrangements in the context of crossing minimization of two–page layouts. The solution in this case is the arrangement of vertices on the spine, and the only move that is defined, is the shifting of a vertex from its position on the spine to a new position.

The loop spanning lines 5–20 enumerates each position  $i$  on the spine  $A$ , and for each  $i$  it calls the nested loop spanning lines 6–19. This second loop moves the currently examined vertex to every other position  $j$  on the spine (if  $i = j$ , then no movement will take place, and this case is ignored; hence the `if` statement at line 7) and computes (line 10) the number of crossings resulting from the application of an edge layout algorithm (line 9) — any edge layout algorithm will suffice, where `PageLayout` would be replaced by the relevant algorithm. If it finds that the current layout improves the best crossing number found thus far (line 11), it stores the position from which vertex  $v$  is to move in  $i^*$  and the position to which the vertex  $v$  should be moved as the variable  $j^*$  (lines 12–13), so that this move can be applied after the two loops terminate. It also notifies the algorithm that it has found an improving neighbouring solution (line 15), and that the search should thus continue for a better solution.

If an improving move was found, then the update is made in lines 21–23. Finally, lines 1–2 are only responsible for computing the initial upper bound  $c'$  on the crossing number. Clearly this algorithm will terminate when a local optimum is found, since no neighbours would represent improved solutions.

### Algorithmic complexity of the neighbourhood search method

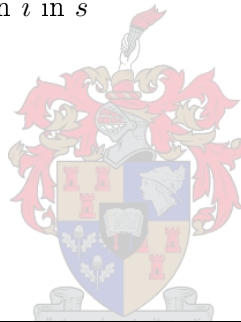
The two nested loops spanning respectively lines 5–20 and lines 6–19 together execute  $O(|V(\mathcal{G})|^2)$  times, for an input graph  $\mathcal{G}$ . Inside these loops, the most time consuming operations are at lines 8, 9, 10 and 17 — the rest of the operations require constant execution time. The movement of a vertex from one position on the spine to the next (lines 8 and 17) takes  $O(|V(\mathcal{G})|)$  time. This is because the vertices are stored in a vector, and when a vertex is moved from its position  $i$  to a new position  $j$ , a number of vertices have to be shifted to make space at  $j$ , and to fill the position  $i$ . The average distance between two randomly selected positions on the spine is  $O(|V(\mathcal{G})|)$ , and therefore, on average  $O(|V(\mathcal{G})|)$  vertices must be moved along on the spine. The complexity of the two–page layout operation (line 9) depends on the layout algorithm used, but it certainly takes at least time  $O(|E(\mathcal{G})|)$  (each edge must, after all, be considered at least once), although it may be seen from the edge layout algorithms in this thesis (§ 6.2.1.1, § 6.2.1.2 and § 6.2.1.3) that the running time is likely to be at least  $O(|E(\mathcal{G})|^2)$ . Denote the running time of the edge layout

**Algorithm 6.7** NeighbourhoodSearch**Input:** A graph  $\mathcal{G}$ , and a spine arrangement  $A$  of  $\mathcal{G}$ 's vertices.**Output:** The possibly updated spine arrangement  $A$  and number of crossings  $c'$ .

```

1:  $\ell \leftarrow \text{PageLayout}(\mathcal{X}_{\mathcal{G},A})$ 
2:  $c' \leftarrow \text{NumberOfCrossings}(\mathcal{X}_{\mathcal{G},A}, \ell)$ 
3: while improving = TRUE do
4:   improving  $\leftarrow$  FALSE
5:   for all  $i \in \{1, \dots, |V(\mathcal{G})|\}$  do
6:     for all  $j \in \{1, \dots, |V(\mathcal{G})|\}$  do
7:       if  $i \neq j$  then
8:         Move  $A_i$  to position  $j$  in  $s$ 
9:          $\ell \leftarrow \text{PageLayout}(\mathcal{X}_{\mathcal{G},A})$ 
10:         $c \leftarrow \text{NumberOfCrossings}(\mathcal{X}_{\mathcal{G},A}, \ell)$ 
11:        if  $c < c'$  then
12:           $i^* \leftarrow i$ 
13:           $j^* \leftarrow j$ 
14:           $c' \leftarrow c$ 
15:          improving  $\leftarrow$  TRUE
16:        end if
17:        Move  $A_j$  back to position  $i$  in  $s$ 
18:      end if
19:    end for
20:  end for
21:  if improving = TRUE then
22:    Move  $A_{i^*}$  to position  $j^*$  in  $A$ 
23:  end if
24: end while
25: return  $A$ ,  $\ell$  and  $c'$ 

```




---

**Implementation:** No implementation for this algorithm is given, since it may be seen as a special case of the tabu search algorithm described later in the chapter.

---

algorithm by  $D$ . Finally, as has been stated before, the time taken to compute the number of crossings in a two–page layout is  $O(|E(\mathcal{G})|)$ . Thus, the time taken by the two nested loops is  $O(D|V(\mathcal{G})|^2)$ . It is not possible to estimate how many times these nested loops will be executed, since the neighbourhood search algorithm runs until no more improvement is possible.

### Incorporating non–greedy behaviour

A major drawback of neighbourhood search, is that it might converge to a weak local optimum. This occurs when it is initialized with the wrong choice of an initial solution (of course, it cannot be known in advance whether an initial solution is “wrong”). The problem may be ameliorated somewhat by performing a number of trials of the neighbourhood search algorithm with random initial solutions, but the main problem is that, in general, good optima may not necessarily be achievable only via moves which are most improving. Of course, this problem itself is not entirely insurmountable, since the neighbourhood search algorithm could be designed to choose randomly amongst a number of improving moves. Nevertheless, a more systematic exploration of the search space is desirable, and this may be achieved by a method such as tabu search.

Tabu search has its antecedents in methods that were designed to violate feasibility barriers or

optimal search directions by the strategic application or release of constraints, so as to permit exploration of regions which would otherwise be forbidden/unreachable. In this thesis, a very one-sided view of the tabu search method is taken, whereby the method may be seen as a generalization of the neighbourhood search method. A tabu search algorithm maintains a memory structure  $H$  of *attributes* of moves that were performed in a number of previous iterations (an example of such an attribute, is: vertex  $v_1$  was moved to position 2 on the spine), and it employs  $H$  to classify certain moves that would result from the current neighbourhood  $N(A)$ , as *tabu* (hence the name of the method). Such moves are not considered by the algorithm; therefore, certain solutions in  $N(A)$  will not be reachable, and therefore the neighbourhood is also a function of  $H$ , and written as  $N(A, H)$ . This avoidance of moves has the effect of occasionally forcing the algorithm to make a number of non-improving moves, thereby facilitating escapes from local optima. An attribute (incorporated in  $H$ ) is active for a number of steps, called its *tenure* and may only play a role in classifying a move as tabu when it is active. An attribute is enabled when its tenure is positive, and this tenure is decreased after every move. Another often used technique is to store previously found local optima — otherwise called elite solutions — and to make moves to such solutions available in  $N(A, H)$  when no improving moves have been found for a certain number of prior moves.

### Attribute based memory

As mentioned above, the memory of the tabu search algorithm is based on *attributes* of solutions. In this context, an attribute is any aspect of a solution that may change during a move. The set of attributes for a solution  $A$  is denoted  $\alpha(A)$ . When a move has one or more attributes with positive tenure values, it might be classified as tabu, depending on how the algorithm was designed — for some problems, a single active attribute would suffice to designate a move as tabu, whereas in some other problems, a host of attributes need to be active for such a designation. The property of being tabu, is not, in general, a binary property — that is to say, a move is not merely tabu or not tabu, but instead, it is assigned a *tabu value* which varies in proportion to the tenure values of the move's attributes.

As an example of how attributes might be defined, consider again the problem of finding good vertex arrangements on the spine of a book. Two types of attributes are defined (without any justification at the present moment). First, when a particular vertex  $v$  has been moved to a new position  $i$  on the spine, it should maintain its position  $i$  for a number of steps. The attribute for this case is defined as being active when  $v$  might be moved *away from* its position. Second, once vertex  $v$  has been moved to a certain position  $i$  on the spine, that position should be “poisoned” so that  $v$  will not be moved there again for a certain number of steps during future iterations. This attribute is thus active when  $v$  is to be moved *to* position  $i$ .

These two attribute types were chosen to highlight the two types of attributes that exist: *from attributes* and *to attributes*. The former define moves that “cancel” attributes, and the latter define moves that “activate” attributes. More technically, let  $A$  be some solution, and let  $A' \in N(A, H)$ . Then, *from attributes* of the move that will transform  $A$  to  $A'$ , are defined as the set  $\alpha(A) \setminus \alpha(A')$ , *i.e.*, all attributes that are considered to be active for  $A$ , but not for  $A'$ . Conversely, *to attributes* of the move are defined as the set  $\alpha(A') \setminus \alpha(A)$ .

It would be impossible to detail a general data structure for the storage of tenure values for attributes, since this is very specific to the problem structure. Of the example attributes, the first type may be stored in a vector, indexed by vertices, and of which the entries are the tenure values (since it depends only on the vertex to be moved away). The second type may be stored using a matrix where rows might specify spine positions, and the columns the vertices. The



entries in the matrix positions contain the tenure values. This latter data structure is used in the tabu search algorithm example below.

An important consideration for the efficiency of a tabu search algorithm, is the time complexity of updating the tenure values of attributes after each iteration. The first of the two example attributes requires a vector, and thus to enumerate the entire vector for decreasing the tenure values is not a very expensive operation. In contrast, the second example attribute requires a matrix, and clearly the enumeration of this structure requires  $O(|V(\mathcal{G})|^2)$  time. This is normally excessive, since such a matrix is likely to be relatively sparsely populated. In this case, it is better to maintain a separate list, which contains the row/column index pairs of the entries which are non-zero. This list is enumerated at each iteration of the tabu algorithm to find non-zero entries in the matrix. When a tenure value of an entry reaches zero, its corresponding row/column pair is removed from the list. Such a list is called a *tabu active list*, an example of which may be seen in the tabu search algorithm example below.

### The selection of solutions from a neighbourhood

In tabu search algorithms, the neighbourhood of a solution need not only contain moves that are obtainable by applying normal moves to the solution. As mentioned above, a common strategy is to maintain a list  $L$  of the best solutions reached so far, commonly called elite solutions, and to define the neighbourhood  $N'(A, H) = N(A, H) \cup L$  for a solution  $A$  if, say, the tabu search algorithm has not made any progress within a specified number of steps. This enables the tabu algorithm to re-examine good solutions, and explore their direct neighbourhoods more closely. In the implementation of tabu search used in this thesis, the list  $L$  is pruned from time to time so that solutions which compare poorly with respect to the best solution encountered thus far are removed, where a solution is regarded as poor if it achieves a number of crossings in excess of 5% of the best known solution. The list is not pruned if  $L$  contains below a certain number of elements, so that diversification is ensured. In the implementation,  $L$  must contain at least two elements (the choice is arbitrary and may be varied). Another important property of a tabu search algorithm, is that tabu values are normally overridden in favour of moves that may reach particularly good solutions. The criteria by which this is decided, are called the *aspiration criteria*, and allow the algorithm to “zoom in” on good local optima. The most common aspiration criterion is to override the tabu status of a move which will lead to a solution that is better than the best solution found at that point. This is also the only aspiration criterion that was used in the tabu search algorithm described below.

A tabu search algorithm typically constructs a list (of maximum fixed length), called the tabu-list, which contains potential next moves from the neighbourhood  $N(A, H)$  of a solution  $A$  and attribute memory  $H$ . Moves are sorted in the list according to their quality (*i.e.*, in the context of two-page layouts, vertex arrangements which permit fewer crossings are higher quality solutions, and moves towards such solutions are thus higher quality moves) and if a move is of lower quality than that of the worst move in the list, it is simply not inserted into the list. Once the list has been constructed, the tabu search algorithm picks a non-tabu move of the highest quality. If all moves in the list are tabu, it chooses the move that is least tabu (*i.e.*, the move with the lowest tabu value).

In the implementation of tabu search used in this thesis, no actual tabu-list is maintained. When the algorithm enumerates the neighbourhood  $N(A, H)$ , then, until it encounters a non-tabu move, it records moves that have the lowest tabu values, and it only discriminates between a pair of tabu moves when their tabu values are equal, but the quality of their solutions differ. Once it encounters a non-tabu move, it begins to discriminate between moves solely on the basis

of their quality and it automatically ignores tabu moves. This is equivalent to a tabu list of infinite length, and has the added advantage of avoiding the process of list manipulation.

### An example of a tabu search implementation

To consolidate these concepts, a number of steps of a simplified tabu search algorithm for vertex arrangements are presented here. The only attribute that is considered, is where, if a vertex  $v$  is moved onto a position  $i$ , it avoids that position for 10 ensuing moves. The algorithm in illustration is assumed to have executed a number of steps already, and thus some of the attribute memory slots will have non-zero tenure values. A star next to a vertex label indicates that it was the most recent move.

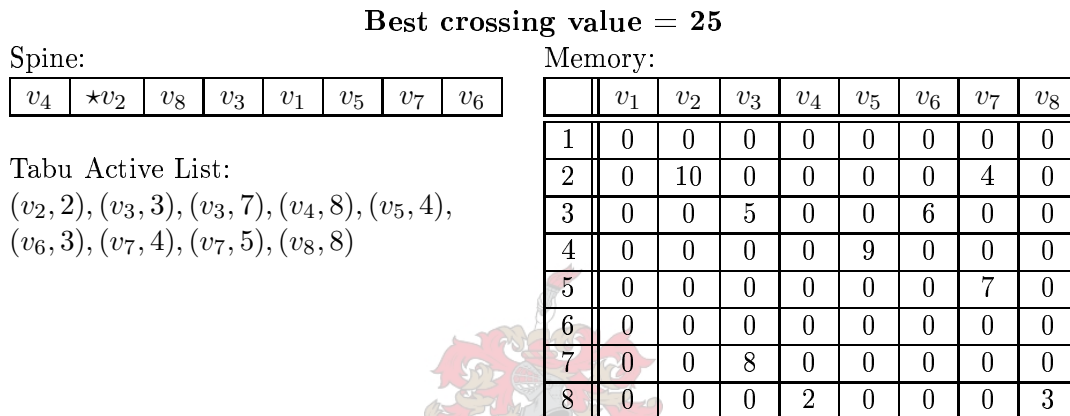


Figure 6.8: Step  $k$  of the tabu illustration.

In step  $k$  of the algorithm (depicted in Figure 6.8), it may be seen that the last move was the move of  $v_2$  to position 2 on the spine. This may be verified from the tabu memory — the entry for  $v_2$  at position 2 is 10, which is the highest tenure value. It is found that moving  $v_5$  to position 2 on the spine would lead to an improvement of the total number of crossings from 25 to 24. This move is performed, since the entry in cell  $(2, v_5)$  of the memory structure is 0.

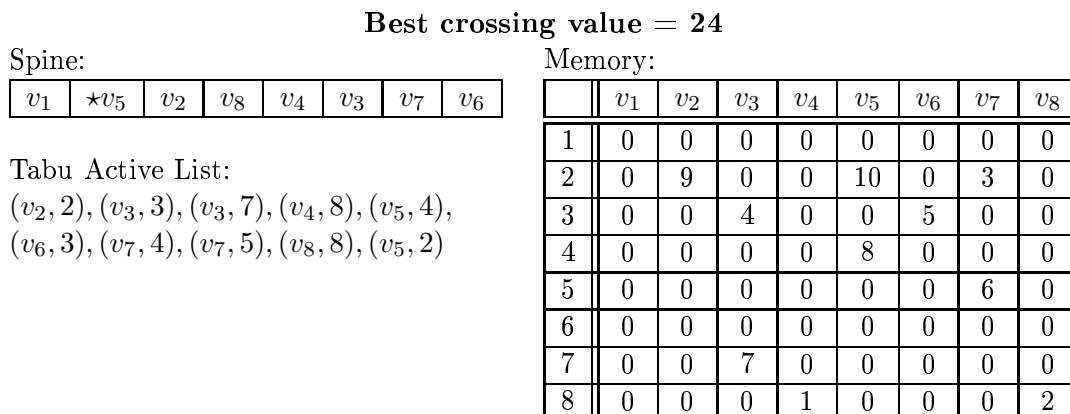


Figure 6.9: Step  $k + 1$  of the tabu illustration.

In step  $k + 1$  (depicted in Figure 6.9), the rest of the tabu tenure values have all been decreased by 1, and it may be seen that  $v_5$  now has a tenure of 10 for position 2 in the memory. The algorithm cannot find an improving move for the next move, and so chooses the least non-improving move,

which is to place  $v_4$  at position 8. Again it may be verified that all of the tabu tenure values have been decreased. The situation depicted in Figure 6.10 results.

**Best crossing value = 24**

Spine:

$v_1$	$v_5$	$v_2$	$v_8$	$v_3$	$v_7$	$v_6$	$\star v_4$
-------	-------	-------	-------	-------	-------	-------	-------------

Memory:

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
1	0	0	0	0	0	0	0	0
2	0	8	0	0	9	0	2	0
3	0	0	3	0	0	4	0	0
4	0	0	0	0	7	0	0	0
5	0	0	0	0	0	0	5	0
6	0	0	0	0	0	0	0	0
7	0	0	6	0	0	0	0	0
8	0	0	0	10	0	0	0	1

Tabu Active List:  
 $(v_2, 2), (v_3, 3), (v_3, 7), (v_4, 8), (v_5, 4),$   
 $(v_6, 3), (v_7, 4), (v_7, 5), (v_8, 8), (v_5, 2)$

Figure 6.10: Step  $k + 2$  of the tabu illustration.

### Frequency based memory

The usual attribute based memory is relatively short-term, in that an attribute only has an effect during its tenure. In order to avoid long-term cycling, and the repetition of particular types of moves, a longer term memory is required. Frequency based memory is used in this regard, and as the name suggests, it keeps track of the frequency with which individual moves are executed, and penalizes those moves which have been executed frequently in the past. The exact semantics of how moves are penalized is problem specific. For example, a move's tabu value may be increased by a factor which is a function of its frequency. Another possibility is to increase the tenure of a move in accordance with its frequency. The definition of how the frequency itself is measured is open to interpretation — that is, per which unit is frequency to be measured? Units that are simple to compute, are the maximum number of times that any move has been performed, and the total number of iterations already performed by the tabu algorithm. For a move  $s$ , denote the total number of times that  $s$  has been performed by  $t_s$ , and denote the value of the unit (*i.e.*, the denominator of the frequency term) by  $u$ . The advantage of letting  $u$  be the maximum number of times that a move has been performed, is that it guarantees that  $0 \leq t_s/u \leq 1$  for any move  $s$ . This makes it fairly easy to design penalty functions that are large when  $t_s/u$  is close to 1 (for example  $e^{c(t_s/u)}$ , where  $c$  is a positive real number), thereby forcing lesser frequent moves to be chosen. If  $u$  is chosen to be the number of iterations that have been performed, then it is virtually guaranteed that  $t_s/u$  will be much smaller than 1, unless a single move is repeatedly performed; it is difficult to estimate what the maximum value of  $t_s/u$  is likely to be. In this case, it is sensible to rely on average case behaviour, and to assume that in the long run, all moves should be executed the same number of times; those moves which are more often executed are then penalized, whilst those that are below the average are privileged. For this thesis, the former mechanism was chosen.

The frequency based memory component of an attribute is likely to be stored in the same data structure as the attribute's tenure based memory, since it has to be indexed via the same means. Thus, if the example tabu search algorithm above were to be augmented to include frequency based memory, it would also have used a matrix as its data structure. Also, instead of storing the frequencies with which moves are made in the frequency memory, only the total number of times that the moves have been executed, is stored — the frequency may readily be computed by dividing with the maximum frequency.

### A tabu search implementation for the vertex arrangement problem

The methods used for the implementation of the Tabu algorithm in this thesis have largely been expounded, and all that remains is to pull together these concepts. The implementation used for this thesis is fairly simple. It implements only a *to attribute*, which is similar to that of the example given above: whenever a vertex  $v$  moves to a position  $i$ , then this position becomes “poisoned”, such any vertex will avoid this position for a number of steps. This attribute is stored as a vector. The algorithm also implements a frequency based component, as described in the section on this type of memory. The actual implementation of the functions which update the memory are omitted, although the reader may refer to the programs in § B.1.2 for the implementations. This algorithm stores elite solutions (*i.e.*, the local optima that led to improved solutions), and the crossing number values achieved by these solutions, in a priority queue  $L$ , from which solutions are revisited when the algorithm does not find any improving moves for a number  $n_{\max}$  of moves. The reason that the crossing number values are also stored, is that it allows the quality of elite solutions to be evaluated when they are extracted from  $L$ . The priority queue prioritises solutions based on the number of times they have been revisited, with the least revisited solutions receiving higher priorities.

Algorithm 6.8 is clearly an augmentation of the neighbourhood search algorithm which is displayed in Algorithm 6.7. The first change is that the termination criterion of the Tabu search algorithm is based on it having reached the maximum number of iterations (as with a genetic algorithm, it is possible to use any of a number of termination criteria); the variable  $x$  maintains the number of steps, which is updated at line 5. At line 6, the number  $n$  is increased with the assumption that the best crossing number bound  $c'$  will not be improved in the current iteration; if, however, the assumption turns out to be false, then  $n$  is set to 0 at line 16. The variables  $c$  and  $t$  pertain to a current move and maintain, respectively, the crossing number bound for the solution generated by the move and the tabu value of the move. Variables marked by an asterisk relate to the neighbourhood:  $c^*$  and  $t^*$  maintain respectively the best crossing number and tabu value over the all moves in the neighbourhood, and  $i^*$ ,  $j^*$  maintain respectively the from- and to-positions on the spine for the best candidate move in the neighbourhood.

The code inside the conditional structure spanning lines 14–17 is executed when a current solution’s crossing number value improves upon the global best value (*i.e.*, the aspiration criterion is met); in this case the data structures pertaining to the solution are stored (line 15) and the best crossing number value is updated (line 16). The tabu value for this move is overridden due to the aspiration criterion, and thus  $t$  becomes 0 (also at line 16); finally, the value of  $r$  is set to “FALSE” to indicate that this solution is not in the elite solution list, and should therefore be inserted along with the number of crossings  $c$  that it achieved, into the priority queue  $L$  at line 26 if it is a local optimum (if  $r$  is “TRUE”, then the optimum is being revisited, and should not be reinserted into  $L$ ). The value of the number of crossings achieved via the solution is also stored, so that the quality of the solution may be compared to the best known solution when it is extracted from  $L$  — if it is of a low quality, it will be discarded and another solution will be extracted from  $L$ . The conditional structure between lines 18–20 is executed when a move is a better candidate than the foregoing moves (and should therefore replace the previous best move). The condition at line 18 requires some untangling to make its meaning clear. The two main mutually exclusive clauses are  $t = 0$  and  $t > 0$ . The former is true when there already was a previous move which had a tabu value of 0; thus, moves are compared solely on the basis of their quality, and hence the clause  $c < c^*$ . If  $t > 0$ , then no prior moves had non-zero tabu values. Therefore, when the tabu value of the current move matches a prior tabu value, it is a better candidate if its solution is of a higher quality, hence the clause ( $t = t^*$  and  $c < c^*$ ). If, however, it has a lower tabu value than all previous moves, it is seen as a better candidate, hence

**Algorithm 6.8** TabuSearch**Input:** A graph  $\mathcal{G}$ , and a spine arrangement  $A$  of the vertices of  $\mathcal{G}$ .**Output:** The possibly updated spine arrangement  $s$  and number of crossings  $c'$ .

```

1:  $\ell \leftarrow \text{PageLayout}(\mathcal{X}_{\mathcal{G},A})$ 
2:  $c' \leftarrow \text{NumberOfCrossings}(\mathcal{X}_{\mathcal{G},A}, \ell)$ 
3:  $H \leftarrow \text{InitializeMemory}(\mathcal{X}_{\mathcal{G},A}, A)$ 
4: for all  $x < x_{\max}$  do
5:    $x \leftarrow x + 1$ 
6:    $n \leftarrow n + 1$ 
7:   for all  $i \in \{1, \dots, |V(\mathcal{G})|\}$  do
8:     for all  $j \in \{1, \dots, |V(\mathcal{G})|\}$  do
9:       if  $i \neq j$  then
10:        Move  $A_i$  to position  $j$  in  $s$ 
11:         $\ell \leftarrow \text{PageLayout}(\mathcal{X}_{\mathcal{G},A})$ 
12:         $c \leftarrow \text{NumberOfCrossings}(\mathcal{X}_{\mathcal{G},A}, \ell)$ 
13:         $t \leftarrow \text{TabuValue}(H, i, j)$ 
14:        if  $c < c'$  then
15:           $\text{StoreResult}(\mathcal{X}_{\mathcal{G},A}, A, \ell)$ 
16:           $c' \leftarrow c, n \leftarrow 0, t \leftarrow 0, r \leftarrow \text{FALSE}$ 
17:        end if
18:        if  $(t = 0 \text{ and } c < c^*) \text{ or } (t > 0 \text{ and } ((t = t^* \text{ and } t < c^*) \text{ or } (t < t^*)))$  then
19:           $t^* \leftarrow t, i^* \leftarrow i, j^* \leftarrow j, c^* \leftarrow c$ 
20:        end if
21:        Move  $A_j$  back to position  $i$  in  $A$ 
22:      end if
23:    end for
24:  end for
25:  if  $n = 1$  and  $r = \text{FALSE}$  then
26:    Insert  $A, c$  into  $L$ 
27:  end if
28:   $\text{DecreaseTenure}(H)$ 
29:  if  $n \geq n_{\max}$  then
30:    repeat
31:      Extract a least revisited solution from  $L$  into  $A, c'$ 
32:      if  $c' > 1.05 \times c$  and  $|L| > 2$  then
33:        Discard  $A$ 
34:      end if
35:    until  $c' \leq 1.05 \times c$  or  $|L| \leq 2$ 
36:    Insert  $A, c'$  into  $L$ , increasing the revisit count of  $A$ 
37:     $n \leftarrow 0, r \leftarrow \text{TRUE}$ 
38:  else
39:     $\text{MemoryInsert}(H, i^*, j^*, n)$ 
40:    Move  $A_{i^*}$  to position  $j^*$  in  $A$ 
41:  end if
42: end for
43: return stored best solution

```

**Implementation:** An implementation of this algorithm is given in § B.2.2.

the clause  $t < t^*$ . At line 19, all the details of a superior move are stored.

Now, moving outside of the two nested loops, consider the line 25. This case can only occur if the previous move improved upon the overall best crossing number bound and was not achieved by revisiting a solution, but if the current best move is does not — in other words, if the current solution  $s$  (that is, before the application of the current move) is a newly found local optimum. This solution is then stored in the list  $L$  of elite solutions at line 26.

The tenure values of all active attributes are decreased at line 28. It is determined at line 29 whether the algorithm has made any improving moves within the maximum number of  $n_{\max}$  steps. If it has, the solution is updated (line 40) and the memory is updated (line 39) to reflect the move from of the vertex  $A_{i^*}$  to the position  $j$  on the spine  $A$ . On the other hand, If  $n \geq n_{\max}$ , a past elite solution is revisited from the priority  $L$  (line 31). If the extracted solution is of a low quality (*i.e.*, the number of crossings  $c'$  is in excess of 5% of the number of crossings for the best known solution), it is discarded (line 33), and another solution is extracted from  $L$ . This process is repeated (lines 30–35) until either a desirable solution is found, or until  $L$  contains less than three elements. When a solution has been found, it is reinserted into  $L$ , with a higher visit count (line 36). Finally,  $n$  is set to zero to indicate a new start and  $r$  is set to indicate that this is a revisited solution.

### Algorithmic complexity of the tabu search implementation

The computational complexity of actions performed within the main loop of the tabu algorithm, which spans lines 4–42, is, in fact, the same as that of the neighbourhood search method. Inside the two nested loops, spanning lines 7–24, the procedure `TabuValue` (line 13) requires a simple lookup in the memory (*i.e.*, the matrix which stores the attribute tenure values) for the tenure value corresponding to the move from  $i$  to  $j$ . This may be performed in constant time. The procedure `StoreResult` (line 15) stores all information regarding the current move, including the intersection graph. The time taken by the edge layout algorithm, depends on the number of edges in the intersection graph, and therefore, the complexity of the storage of this graph cannot be greater than the complexity of the edge layout algorithm. As in the case of the neighbourhood search method, denote the time complexity expression of the edge layout algorithm by  $D$ . Then, the time complexity of the two nested loops (lines 7–24) is  $O(D|V(\mathcal{G})|^2)$  as in the case of the two nested loops in the neighbourhood search method.

The priority queue  $L$  which maintains a list of elite solutions, may itself become very long. A priority queue is normally implemented using a heap data structure, which permits the extraction (line 31) and insertion (lines 26 and 36) of an element in  $O(\log_2 |L|)$  time. It is, of course, possible that  $L$  might become very large, but it is noted that local optima are typically few and far between — indeed, in the unlikely situation that 512 local optima are visited,  $\log_2 512 = 10$ . This value is, however, much smaller than the running time of  $O(D|V(\mathcal{G})|^2)$  for the two nested loops (lines 6–24). Therefore, in practice this is not a problem.

Finally, only the complexity of the procedure `MemoryInsert` (line 39) remains to be examined. As with the memory lookup required to compute the tabu value (line 13), this may be achieved in constant time, because only the tenure value of the current attribute value has to be updated, and this is tantamount to the addition of a single value in a matrix.

Therefore, the time complexity of the two nested loops dominates the execution time of the main loop (lines 4–42). This loop is executed  $x_{\max}$  times, so that the resultant time complexity for the tabu search algorithm is  $O(x_{\max} D|V(\mathcal{G})|^2)$ .

### Pre-optimization techniques

An “undesirable” initial vertex arrangement may impact negatively on the total running time of the tabu search method, and on its rate of convergence. For a graph  $\mathcal{G}$ , a vertex arrangement  $A$  is said to be “undesirable” when its corresponding intersection graph  $\mathcal{X}_{\mathcal{G},A}$  is dense (*i.e.*, contains a high number of edges relative to the number of vertices). It is very likely that the minimum number of crossings achievable by an optimal edge layout for a dense intersection graph will be larger than the number of crossing achievable by an optimal edge layout for a sparser intersection graph. This larger number of edges forces an edge layout algorithm to run more slowly, since all edges of the intersection graph need to be considered. Furthermore, a greater number of moves has to be performed by the tabu search algorithm in order to locate vertex arrangements with sparser corresponding intersection graphs. These problems may be mitigated by the application of pre-optimization techniques which generate initial vertex arrangements of “reasonable” quality.

The first method proposed is the pre-optimization phase of Nicholson’s Heuristic (§ 4.3.2.2). In this phase, a vertex arrangement is constructed one vertex at a time from an input graph  $\mathcal{G}$ , where at each step, an unplaced vertex from  $\mathcal{G}$  with the highest connectivity to the vertices that have already been placed, is selected, and inserted into a position in the arrangement where it induces the lowest number of crossings (its edges are placed so as to achieve this number of crossings). In the post-optimization phase of Nicholson’s heuristic, a vertex in the arrangement which would cause the greatest decrease in the number of crossings is moved to a position in the arrangement which would achieve this decrease, and the process is repeated, until no better improvement is possible.

Because the tabu search method behaves like the neighbourhood search method up to the point that the first local optimum is found, it fulfils the role of the post-optimization phase of Nicholson’s heuristic. Furthermore, where only the layouts of the edges of the vertex that has been moved are adjusted in the post-optimization phase of Nicholson’s heuristic, a new edge layout for the vertex arrangement is calculated in its entirety by the tabu search method. This generally leads to greater decreases in the number of crossings for each such move.

For the second pre-optimization method proposed, the vertices are placed on the spine in the order of the vertices of a Hamiltonian cycle of an input graph  $\mathcal{G}$ , or otherwise in the order of the vertices in the longest path that could be found in  $\mathcal{G}$ . In the latter case, the vertices of  $\mathcal{G}$  that are not in the path are placed randomly on the remainder of the spine. The algorithm for finding Hamiltonian cycles that was used for the programs in this thesis, is based on concepts that were developed by Pósa [P76]. A description and an implementation of the algorithm are discussed in § B.2.1.2.

## 6.3 A new Székely–esque upper bound algorithm

Székely’s algorithm, which is described in § 4.3.1.4, gives an elegant method for computing the independent-odd crossing number of a graph. The algorithm is insensitive to a particular vertex arrangement, although an arrangement must be fixed for the duration of the algorithm. The only information of which the algorithm keeps track, is the manner in which each edge “separates” the vertices that are not incident to it; this, in conjunction with the vertex ordering, specifies the independent-odd crossing number for the given configuration.

Vertex separation really describes the way that an edge “weaves” around the various vertices to which it is not incident. Consider the following mechanical procedure for constructing circular drawings, given the vertex separation information for each edge: every edge is drawn so that if it

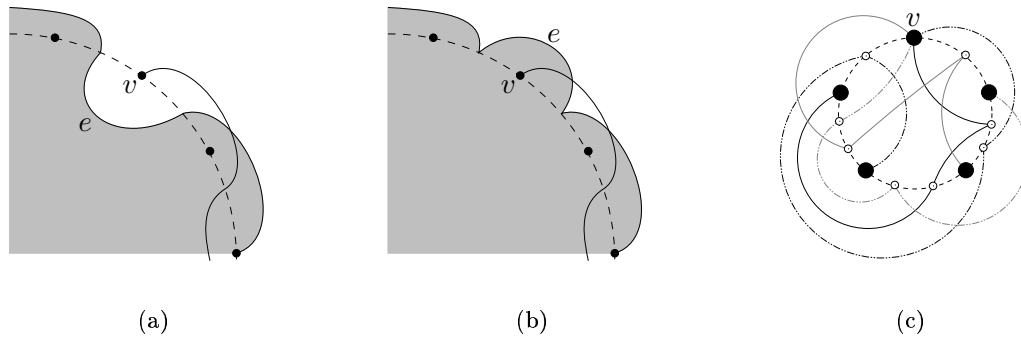


Figure 6.11: A mechanical method for drawing construction from Székely's algorithm.

does not pass over the (imaginary) circle between a pair of vertices  $u$  and  $v$  (which lie adjacent to one another on the circle), then it “touches down” on the circle between  $u$  and  $v$ . This scheme is depicted in Figure 6.11; in part (a) of the figure, the edge  $e$  “weaves” through the various vertices, but in part (b), it only touches down on the circle on both sides of  $v$ .

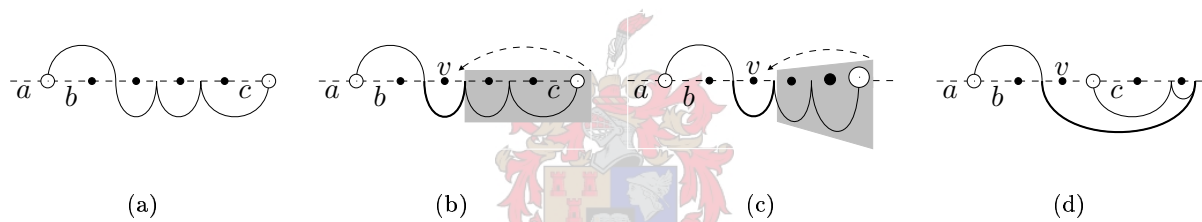


Figure 6.12: Edges need not touch the circle between every pair of adjacent lying vertices.

Whilst admittedly, this scheme hardly produces aesthetically pleasing results, it has the advantage of partitioning edges into sections (each of which is between two points that touch down on the circle) such that the shifting of such sections from the inside to the outside (or *vice versa*) of the circle produces the various vertex separation configurations. This is exactly what happens in Figure 6.11 — in part (a), the vertex  $v$  is separated from the other vertices, which are in the gray region, but in part (b), it is no longer separated from those vertices.

From this figure it is clear that the edge that is incident to  $v$  does not touch down on the circle between  $v$  and the vertex that lies just to the right of  $v$ . In fact, in a graph  $\mathcal{G}$ , an edge separates  $|V(\mathcal{G})| - 2$  vertices, since it cannot separate its own incident vertices, which means that each edge need only be separated into  $|V(\mathcal{G})| - 2$  sections, and this may be achieved by letting an edge touch  $|V(\mathcal{G})| - 1$  points on the circle. If, however, in the mechanical drawing method, each edge would have to touch down on the circle between every pair of vertices which lie adjacent on the circle, then each edge would have a total of  $|V(\mathcal{G})| + 1$  sections. Thus, at least three of these sections are redundant. It is a simple matter to show that an edge requires only  $|V(\mathcal{G})| - 2$  sections when the vertices that are incident to the edge lie adjacent to one another on the circle. Consider Figure 6.12(a), which is a flattened version of a circle. In this figure, the two white vertices lie adjacent to one another on the circle, and the circle section  $a$  is the part of the circle spanning between them. Now, starting with the leftmost white vertex, the edge may avoid touching down on the circle in the segment labelled  $b$ , since it could have no impact on the segmentation of any vertex. Likewise, for the segment labelled  $c$ . Now clearly, there would be no purpose to let the edge touch down in the segment labelled  $a$  either. It may be verified that



a single edge is dedicated to each of the remaining (black) vertices, and thus that the separation of each vertex may be handled independently. Suppose now that the two white vertices do not lie adjacent to one another on the circle. In this case, the configuration may be constructed from the former construction (*i.e.*, where the two white vertices lie adjacent to one another on the circle). Let  $v$  be the first vertex on the circle that falls before the second white vertex, as shown in Figure 6.12(d). Identify  $v$  with the vertex from the first scenario that falls on the same position, so that one has the vertex  $v$  in Figure 6.12(b). Now, everything to the right of the bold edge below  $v$ , which is marked by the gray block in Figure 6.12(b) is to be “swung around” to the left under the bold edge. Imagine the right most point where the bold edge section touches down on the circle to be a vertical pivot, and the gray section to be a door attached to the pivot. Figure 6.12(c) suggests this “swinging” action, and finally Figure 6.12(d) results. Now it may still be verified that each black vertex has an edge section. The bold edge section may now affect not only  $v$ , but also the two vertices to the right of the second white vertex. However, the situation where the bold vertex changes the separation of these two vertices may always be corrected by their switching the sides upon which their own edge segments fall.

Clearly this partitioning of edges may be achieved by subdividing the edges of a graph, such that the subdivision vertices are placed at the points where the edges would have touched down, or passed over the circle. But this transforms the layout into a combinatorial two–page book layout (since book layouts are equivalent to circular layouts — § 3.1.3.2). This is illustrated in full in Figure 6.11(c) for the star graph  $\mathcal{K}_{1,4}$ .

Unfortunately, the edge layout algorithms cannot be applied directly to such a book layout, since the way in which crossings are counted differs from that of the crossing number (afterall, Székely’s algorithm determines the independent–odd crossing number). The number of crossings that an edge  $e$  induces by being on a particular page, does not solely depend on the number of edges which alternate it on that page. It must be kept in mind that  $e$  is part of some subdivided edge  $E$ , and that when it crosses another edge  $f$ , it is also the case that  $f$  is itself part of a subdivided edge  $F$ . Thus,  $e$  may alternate the edge  $f$ , but due to the fact that other crossings may be present between sections of  $E$  and  $F$ ,  $E$  and  $F$  may cross an even number of times and hence  $e$  and  $f$ ’s crossing really has the effect of letting  $E$  and  $F$  cross an even number of times.

Thus, the following adjustment has to be made to the edge–layout algorithms described this chapter to make them usable as independent–odd crossing number algorithms: When an edge  $E$  is subdivided into a number of edges  $e_1, e_2, \dots, e_t$ , then each of the edges  $e_i$ ,  $1 \leq i \leq t$  store a reference to  $E$ . Now, when considering whether a sub–edge  $e_i$ ,  $1 \leq i \leq t$  should rather be drawn on the other page, it must first be determined how the number of crossings change with the *removal* of  $e_i$  from its current page (it is quite possible that the number of crossings might increase, since the removal of  $e_i$  might suddenly cause  $E$  to have an odd number of crossings with edges which were crossed by  $e_i$ ) — denote this difference by  $r_{e_i}$  (it is positive when the number of crossings increase). Then, when  $e_i$  is inserted into the other page, let  $p_{e_i}$  denote the change in the number of crossings that result; an improvement occurs if  $r_{e_i} + p_{e_i} < 0$ .

To determine whether two alternating edges  $e$  and  $f$  indeed cause a crossing (*i.e.*, whether they cause their “parent” edges to cross an odd number of times), references to their respective “parent” edges  $E$  and  $F$  are obtained — the parity of the number of crossings between the edges may be computed considering the total number of crossings that occur between the sub–edges of the two “parent” edges. This seems an expensive operation, since each edge of  $E$  would have to be considered in turn, and the number of times that edges from  $F$  cross each such an edge has to be computed. There is, fortunately an efficient method for computing the crossing parity that an edge  $e$  would incur by drawing it on the other page. A two dimensional matrix, of which the rows and columns are both indexed by “parent” edges, maintains the parity of the current

number of crossings between each pair of parent edges. Suppose that, for  $E$  and  $F$ , this is 1. Now, if  $e$  is to be drawn on the other page, the set of edges  $B$  which alternate  $e$  on the current page are first considered. For each edge  $g \in B$ , the parent edge  $G$  is obtained, and the parity of the entry corresponding to  $E$  and  $G$  in the parity matrix is changed (*i.e.*, the bit is flipped). This renders the parity situation after  $e$  has been removed from its current page, but before its insertion on the opposite page. The updating of the parity information for the insertion of  $e$  on the opposite page proceeds in the same manner as before — for each edge that alternates  $e$  on the new page, the parent edge is found, and the parity matrix is updated. If  $P_1$  is the parity matrix before removing  $e$  from its original page, and  $P_2$  the parity matrix after the removal of  $e$  from its page, then the matrix  $P_2 - P_1$ , obtained by normal subtraction (*i.e.*, not over modulo 2 operations, as is the case for the prior operations on these matrices), will have entries of  $-1$  where a crossing was rendered even by the removal of  $e$ , 1 where a crossing was rendered odd by the removal of  $e$ , and 0 otherwise. Thus, by summing the entries, one obtains the change in the number of crossings.

## 6.4 Constructing a planarized graph

So far, the issue of the generation of a drawing of a graph from a two–page layout solution has been ignored. This is of course a very important, and often overlooked aspect of the problem.

### 6.4.1 Spine drawings

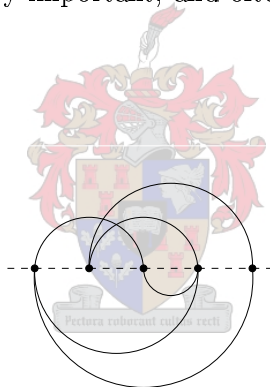


Figure 6.13: A normal spine drawing.

The simplest method by which to generate drawings, is to draw the vertices on a straight line in the order in which they appear on the spine, and to draw edges as half–circles whose diameters are equal to the distances of separation on the spine, between the vertices to which they are incident. Such half circles are then drawn either above the spine, or below it. This method has already been seen in this thesis, and is essentially that of Shahrokhi, Sýkora, Székely and Vrfo [SSSV96b], and of Cimikowski [Cim02]. An example is shown in Figure 6.13.

This method has the advantage of being very easy to implement. However, its main drawback is that drawings become hard to study when there are many edges, since crossings may lie close together. Difficulties also arise when moving vertices, since the crossing configurations may easily be disrupted. For this reason, it is more desirable to represent crossings as vertices, as may be seen in the following section.

### 6.4.2 Planarizing drawings

The problems posed by spine drawings of the previous section are easily mitigated by graph planarization. In such a planarization, crossings are represented as artificial vertices, making it

possible to apply planar graph layout algorithms, and thus to obtaining graph drawings that are less cluttered than would be the case with spine drawings.

If, for each edge, the order in which it is crossed by other edges is known, then Algorithm 5.3 (**ConstructGraph**) may be used to construct the planarized graph. The problem now remains to determine the order in which edges cross each other. It would certainly be beneficial if the set of lists of edge orderings could be computed as a function of the vertex arrangement.

First some basic nomenclature is required to reason about drawings. Let  $e = \{u, v\}$  be an arbitrary edge in a graph  $\mathcal{G}$ , and let  $A$  be a vertex arrangement of  $\mathcal{G}$ . As per convention, let  $u$  be the leftmost vertex in  $A$ , and  $v$  the rightmost. Each edge  $f = \{w, x\} \in E(\mathcal{G}) \setminus \{e\}$  has one of four states with respect to  $e$ :

1. if  $x$  occurs between  $u$  and  $v$ , and  $w$  occurs to the left of  $u$  in  $A$ , then  $f$  is called *left-going* with respect to  $e$ ,
2. if  $w$  occurs between  $u$  and  $v$ , and  $x$  occurs to the right of  $v$  in  $A$ , then  $f$  is called *right-going* with respect to  $e$ ,
3. if  $x$  and  $w$  both occur between  $u$  and  $v$  in  $A$  then  $f$  is called *inside* of  $e$ ,
4. if  $x$  and  $w$  both occur either to the left of  $u$  or to the right of  $v$ , then  $f$  is called *outside* of  $e$ .

Of course the edges which are either right-going or left-going with respect to  $e$  are exactly those edges which alternate  $e$ , and these are also the only edges of interest for the ensuing discussion.

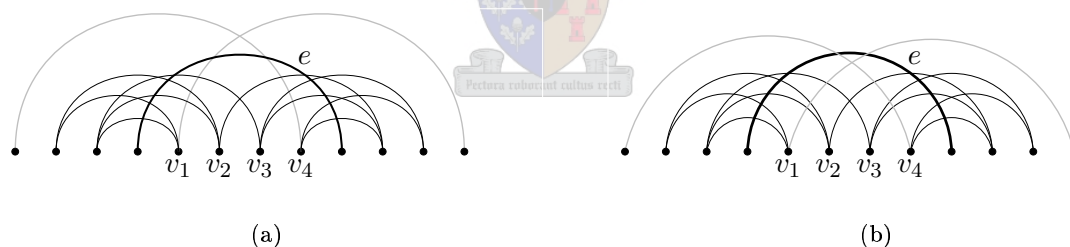


Figure 6.14: The ordering of vertices on the spine determines the order of edge crossings.

In Figure 6.14(a), if the two grayed edges are at first ignored, then the following simple pattern may be observed: All edges which are left-going with respect to  $e$ , cross  $e$  before any of the edges which are right-going with respect to  $e$ . In fact, the left-going edges all cross only the left half of the arc which represents  $e$ , and *vice-versa* for the right-going edges.

Furthermore, when considering only the left-going edges, it may be seen that all left-going edges incident to  $v_1$  cross  $e$  before the left-going edges incident to  $v_2$ . This is also true for  $v_2$  in relation to  $v_3$ . This same pattern is true for the right-going edges.

Finally, for each left-going edge  $f$  incident to  $v_1$ , the order in which  $f$  crosses  $e$  in relation to the other edges is determined by the distance of the opposite (to  $v_1$ ) incident vertex of  $f$ . The closer the distance, the closer to the beginning of  $e$  the crossing occurs. In fact, this is again true for all left-going edges of all vertices that lie between  $u$  and  $v$ . The situation is almost the same for right-going edges; the only difference being that right-going edges whose rightmost vertices lie further along the spine, cross before those whose rightmost vertices lie closer.

This order may be obtained by sorting the edges that cross  $e$  in the following way: First group together all left-going edges in one set,  $L$ , and group together all right-going edges in another set,  $R$ . Sort the edges in  $L$  primarily in ascending order of the positions of their vertices which fall between the vertices incident to  $e$ , and secondarily in descending order of the positions of their vertices which fall outside of  $e$ . The edges in  $R$  are also primarily sorted in ascending order of the positions of their vertices inside of  $e$ , but secondarily they are sorted in *ascending* order of the positions of their vertices which fall outside of  $e$ .

The drawing in Figure 6.14(a) is made by drawing edges as half circles, as described in § 4.3.2.1. Such drawings are used in the articles of Shahrokhi, Székely, Sýkora, and Vrton [SSSV96b] and of Cimikowski and Shope [Cim02, CS96].

Now, when the gray edges in Figure 6.14(a) are considered, the pattern just described, is broken. The right-going gray edge crosses  $e$  *before* the left-going edge, and it crosses the arc representing  $e$  on the left side of the arc. Fortunately, this pattern may be restored, if the edges are drawn in a different manner. Instead of using half circles, arcs from larger circles are used (*i.e.*, the arcs are “flatter”), as shown in Figure 6.14(b). Here it may be seen that all right-going edges cross  $e$  after all left-going edges — in fact, all left-going edges occur precisely in the left half of  $e$  and the right-going edges in the right half. By choosing the arcs from “large enough” circles, it can always be ensured that the desirable order is obtained — from this it can be deduced that the wider the vertices of an edge is apart, the “flatter” its arc is likely to be. Another point worth mentioning, is that due to this setup, no edges from a vertex  $v$  would ever cross  $e$  before the edges of a vertex  $u$  which is situated before  $v$  on the spine. For example, consider  $v_3$  and  $v_4$  in Figure 6.14(b) — the only way in which any of the right-going edges of  $v_4$  (say) could cross  $e$  before edges of  $v_3$ , would be if such an edge crossed one of the edges of  $v_3$  in its left half. By assumption, this is impossible, as the arcs were chosen exactly to avoid this situation.



## Implementation of the algorithm

Now it is possible to construct an algorithm from these ideas. Again, the intersection graph is used, since it readily provides the left-going and right-going edges for an edge  $e$  (which, as has been said, are exactly the edges which alternate  $e$ ). For an input graph  $\mathcal{G}$ , the only purpose of the algorithm is to construct the lists of alternating edges in the order that they cross each edge in the graph; after this point the algorithm **ConstructGraph** may be used to obtain the planarized drawing — the result is a very simple algorithm.

The procedure is called **ComputePlanarOrderings** and is shown in Algorithm 6.9. Essentially, the algorithm steps through every edge  $e$  in the input graph  $\mathcal{G}$  in the loop spanning lines 1–11. Although the intersection graph  $\mathcal{X}_{\mathcal{G},A}$  is not shown, it is used in lines 1 and 2 to determine respectively the left-going and right-going edges of  $e$ , which are stored into the variables  $L$  and  $R$ . In a computer implementation, both of these lines would expand to a number of lines of computer code, which would include finding crossing-vertex  $v$  of  $e$  in  $\mathcal{X}_{\mathcal{G},A}$ , the determination of the crossing-vertices which are adjacent to and in the same partition as  $v$ , and the mapping back of these crossing-vertices to edges in  $\mathcal{G}$ . Both  $L$  and  $R$  are sorted according to the method described in lines 4 and 5. These two ordered sets are then combined into the variable  $C$  (line 6), and this set is enumerated in the loop at line 7 to fill the crossing list  $\mathbf{L}_e$  of  $e$  (line 9) and the general list of crossings  $\mathbf{D}$  (line 8).

**Algorithm 6.9** ComputePlanarOrderings**Input:** A graph  $\mathcal{G}$ , a list of crossed edges  $\mathbf{D}$ , and the crossing configurations for the edges  $\mathbf{L}$ .**Output:** The graph  $\mathcal{G}$ , modified so that the crossings from  $\mathbf{D}$  are present as vertices of degree four, in the order given for the edges in  $\mathbf{L}$ .

```

1: for all  $e \in E(\mathcal{G})$  do
2:    $L \leftarrow \{f : f \text{ is a left-going edge of } e\}$ 
3:    $R \leftarrow \{f : f \text{ is a right-going edge of } e\}$ 
4:   sort  $L$  primarily in ascending order of the highest spine positions of the incident vertices
     of its edges, and secondarily in descending order of its edges the lowest spine positions of
     the incident vertices of its edges.
5:   sort  $R$  primarily in ascending order of the lowest spine positions of the incident vertices
     of its edges, and secondarily also in ascending order of the highest spine positions of the
     incident vertices of its edges.
6:    $C \leftarrow L \cup R$ 
7:   for all  $f \in C$  do
8:      $\mathbf{D} \leftarrow \mathbf{D} \cup \{e, f\}$ 
9:      $\mathbf{L}_e \leftarrow \mathbf{L}_e \cup \{f\}$ 
10:  end for
11: end for
12: ConstructGraph( $\mathbf{L}, \mathbf{D}, \mathcal{G}$ )

```

**Implementation:** An implementation of this algorithm may be found in § B.5.2.**Algorithm complexity of the algorithm** ComputePlanarOrderings

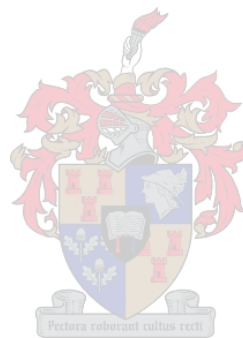
For an input graph  $\mathcal{G}$ , in the worst case, each edge crosses  $O(|E(\mathcal{G})|)$  edges. The sorting of a set of  $N$  items may be performed in  $O(N)$  time using a combination of Counting sort with Radix sort, or by using Bucket Sort [CLR97]. The set of edges is split into the two sets  $L$  and  $R$ , each of which will have  $O(|E(\mathcal{G})|)$  edges in the worst case scenario. The loop spanning lines 7–10 is executed for each crossed edge; the operations within the loop both take constant time. Therefore the loop is executed in  $O(|E(\mathcal{G})|)$  time. Thus, the computational complexity of the operations within the main loop (lines 1–11) is  $O(|E(\mathcal{G})|)$ , and because this is executed for each edge, the total running time of the main loop is  $O(|E(\mathcal{G})|^2)$ . As may be seen from § 5.2, the running time of the algorithm ConstructGraph is also  $O(|E(\mathcal{G})|^2)$ , and therefore, the running time of the algorithm ComputePlanarOrderings is  $O(|E(\mathcal{G})|^2)$ .

## 6.5 Chapter summary

In the first section, § 6.1, of the chapter, an algorithm for approximating lower bounds to the crossing numbers of graphs were developed, using the theory of graph-to-graph embedding, and basic concepts from the theory of shortest paths in graphs. It was shown that this technique has the peculiar property of improving solutions by feeding prior solutions back into itself.

The second section, § 6.2, is wholly concerned with the implementation of upper bound algorithms for the crossing number of a graph, using the two-page book layout framework. It was shown that the problem of determining edge layouts is equivalent to a vertex partitioning problem of the so-called intersection graph, which offers benefits in terms of computational efficiency. The edge layout algorithms in the section were all implemented to operate on the intersection graph. Two new edge layout algorithms were developed, one being a simple iterating greedy heuristic, and

the other being a more complex genetic algorithm. An implementation of the neural network algorithm of Cimikowski and Shope [CS96] was also given in terms of the intersection graph. The basic theory of the tabu search method was reviewed, and a tabu search algorithm for determining vertex arrangements was developed. Finally, an algorithm for obtaining drawings from given book layouts was developed.



# Chapter 7

## Computational Results

We think in generalities, but we live in details.

— *Alfred North Whitehead (1861–1947)*

This chapter is concerned with both the technical aspects of the algorithms that have been described in this thesis, and the output generated by these algorithms. The first section, § 7.1, focuses on the technical side, such as the convergence properties of the optimization algorithms that were described in Chapters 5 and 6. The second section, § 7.2, is dedicated to the consideration of outputs generated by the algorithms, and problem instances which cause some of the algorithms to perform poorly are also considered. A large catalogue of drawings of non-planar complete multipartite graphs with low crossing numbers is provided in this section.

### 7.1 Convergence of upper bound algorithms

In this section, the algorithmic complexity of the algorithm **GreedySide** is examined from a statistical perspective. The effects of varying the parameters of the genetic edge layout algorithm, and of the tabu search algorithm are also considered. The convergence properties of the neural network algorithm are not discussed in this thesis, as this topic has already been dealt with by Cimikowski and Shope [CS96], and their recommendation for the parameters of the neural network algorithm has already been mentioned in the thesis.

#### 7.1.1 Algorithmic complexity of the algorithm **GreedySide**

To determine the number of iterations that the algorithm **GreedySide** (Algorithm 6.3, § 6.2.1.1) would perform in the best, worst and average cases for a graph  $\mathcal{G}$ , the algorithm would have to be applied to each vertex arrangement  $A$  of  $\mathcal{G}$ , and for each  $A$ , all possible distinct edge layouts would have to be considered (since the initial edge layout configuration influences the execution of the algorithm). However, since there are a total of  $O((|V(\mathcal{G})| - 1)!)$  vertex arrangements, and a total of  $O(2^{|E(\mathcal{G})| - 1})$  edge layouts for any given vertex arrangement, this approach is not feasible for any but the smallest graphs. Therefore, only a relatively small number of vertex arrangements and edge layouts for a graph  $\mathcal{G}$  can be sampled, and the behaviour of the algorithm **GreedySide** must be inferred from the statistical data obtained from such trials.

The first aspect of the **GreedySide** algorithm to be considered, is the growth of its complexity as a function of the number of vertices of complete graphs, balanced bipartite graphs (*i.e.*, both

partite sets have the same cardinality) and balanced toroidal grid graphs (*i.e.*, the Cartesian product of cycles of equal order). With the first two classes of graphs, the number of edges grow with the square of the number of vertices, whilst in the case of the toroidal grid graphs, the number of edges increase linearly with respect to the number of vertices. The maximum, average, and minimum numbers of iterations that the algorithm was observed to have executed when it was applied to graphs from the three classes, are shown in Figure 7.1. For each graph to which the algorithm was applied, it was applied at least 500 times with random vertex arrangements and edge layouts.

The graphs seem to suggest that the average algorithmic complexity of the **GreedySide** algorithm is sub-linear with respect to the number of vertices in all three cases (although, of course, this does not mean that this behaviour may be extrapolated to infer the asymptotic running time of the algorithm). Furthermore, as the number of vertices in each of the classes of graphs increases, the observed maximum number of iterations that the algorithm would perform remains less than the number of vertices.

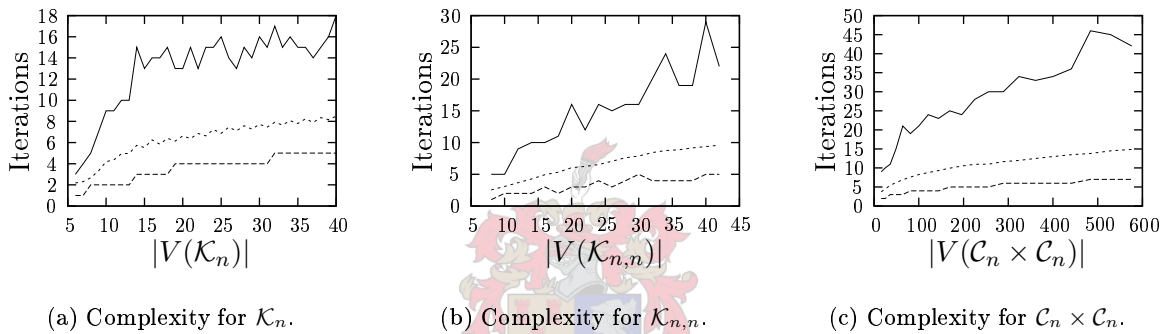


Figure 7.1: Increase in the algorithmic complexity of the **GreedySide** algorithm with respect to the number of vertices in complete graphs  $\mathcal{K}_n$ , balanced bipartite graphs  $\mathcal{K}_{n,n}$ , and balanced toroidal grid graphs  $\mathcal{C}_n \times \mathcal{C}_n$ . The solid black lines indicate the observed maximum number of iterations, the dotted lines indicate the observed average number of iterations performed by the algorithm and the stippled lines indicate the observed minimum number of iterations to be executed.

The *density* of a graph  $\mathcal{G}$  is defined as the value  $|E(\mathcal{G})|/\binom{|V(\mathcal{G})|}{2}$ , *i.e.*, the ratio between the size of  $\mathcal{G}$ , and the size of the complete graph on  $|V(\mathcal{G})|$  vertices. For a random graph  $\mathcal{G}$  on  $n$  vertices, it seems plausible that the complexity of the algorithm **GreedySide** would depend on the density of  $\mathcal{G}$ .

At least 500 random, connected graphs for each of the orders 20, 40 and 60 were generated, and the **GreedySide** algorithm was applied to each of these graphs with numerous ( $> 50$ ) initial random vertex arrangements and edge layouts. The observed maximum, average and minimum numbers of iterations required for convergence of the algorithm as a function of the numbers of edges in the three classes of graphs are shown in Figure 7.2, where  $\mathcal{R}(m, n)$  denotes a random graph of order  $m$ , and size  $n$ .

The number of iterations required for convergence initially increases as the density increases. However, the average number of required iterations decreases again beyond certain points. The only (highly speculative) explanation that the author can offer for this algorithmic behaviour, is that in a very dense graph, the movement of an edge from one page to another, generally causes a large change in the number of crossings in the configuration. This leads to opportunities for faster convergence. When a graph is less dense, the number of crossings in an edge layout might



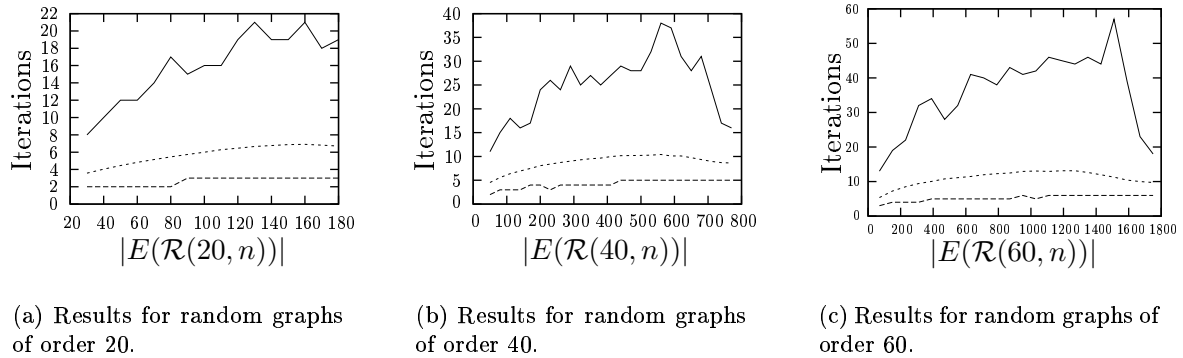


Figure 7.2: Increase in the algorithmic complexity of the **GreedySide** algorithm with respect to the density of random graphs on fixed numbers of vertices. The solid lines indicate the observed maximum number of iterations, the dotted lines indicate the observed average number of iterations performed by the algorithm and the stippled lines indicate the observed minimum number of iterations performed.

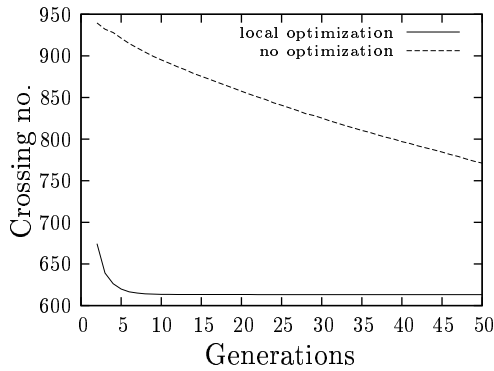
not be greatly affected by the movement of some edges to different pages, since these edges simply might not alternate a large number of other edges. Again it should be noted that the maximum number of observed iterations generally do not exceed the numbers of vertices in the three cases.

### 7.1.2 Convergence of the Genetic algorithm

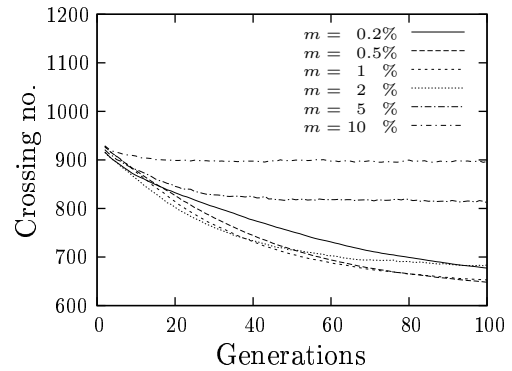
Despite the issues of “solution diffusion” that were discussed in the previous chapter, the genetic algorithm manages to obtain fairly reasonable solutions, even without utilizing local optimization. The main point that counts against it, is that it is computationally far less efficient than both the neural network algorithm, and the **GreedySide** algorithm. What does count in its favour, is that it is less likely to get trapped in weak local optima than the **GreedySide** algorithm, due to the fact that mutations on chromosomes facilitate escapes from such local optima. The genetic algorithm is quite sensitive with respect to the settings of its parameters. The five parameters that alter the behaviour of the genetic algorithm are

1. the population size,
2. the total number of generations,
3. the number of chromosomes involved per tournament,
4. the probability of the mutation of a bit (*i.e.*, the page on which the corresponding edge is to be drawn),
5. the option of disabling local optimization.

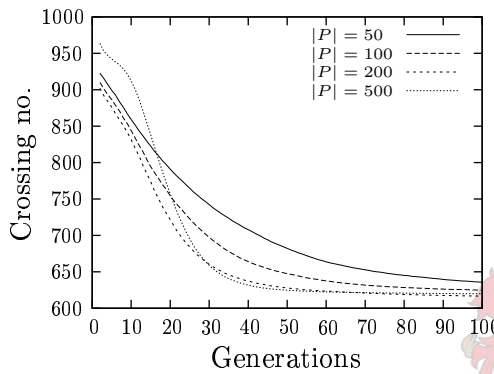
Two hundred random, connected graph of order 20 and size 120 were generated for the purpose of testing the effect of variations in the parameters of the genetic algorithm. In all of the experiments, the average crossing number bound over all 200 graphs was plotted against the generation number (*i.e.*, the iteration of the genetic algorithm). In the initial parameter setup for the experiments, the mutation rate was set to 0.2%, whilst the population size was 20, and the tournament size was set to three competitors. The parameter with the most significant impact on the rate of convergence of the genetic algorithm, was found to be the choice of whether local



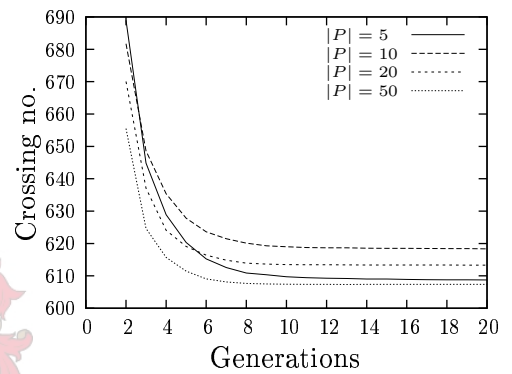
(a) Impact of local optimization.



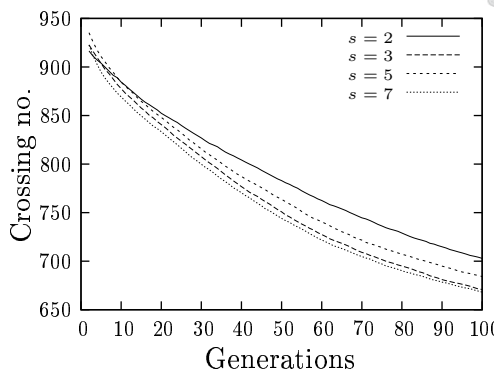
(b) Mutation of genes.



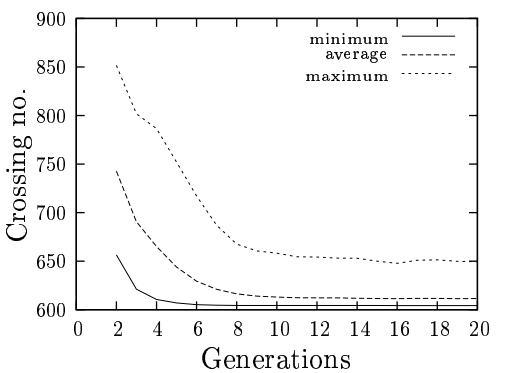
(c) Variation of population size under no local optimization.



(d) Variation of population size under local optimization.



(e) Number of tournament competitors.



(f) Application of the best parameters.

Figure 7.3: An illustration of the effects of the variation of the genetic parameters.

optimization should be applied or not. The difference between the two choices is illustrated in Figure 7.3(a), where local optimization leads to convergence within a few steps, whilst the non locally optimized version improves slowly. The mutation rate was also found to be an important parameter, and if it is set too high, the algorithm may never converge, as may be seen for the higher mutation probabilities in Figure 7.3(b), where  $m$  denotes the mutation probability. A mu-

tation rate of 1% or 0.5% results in acceptable convergence rates. When no local optimization is used, a large population size mitigates the solution diffusion properties of the genetic algorithm, since there is, presumably, a better chance for fit individuals to be created. The plot in Figure 7.3(c) suggests that a population size of 200 is a good choice when local optimization is not applied ( $\mathcal{P}$  denotes the population, and  $|\mathcal{P}|$  the size of the population). When local optimization is used, only very small populations are required for acceptable convergence rates. Populations as small as 5 members are sufficient for the genetic algorithm to converge. The impact of different population sizes on the rate of convergence for the case where local optimization is used, is shown in Figure 7.3(d), where again,  $\mathcal{P}$  denotes the population. A population of size 20 to 50 provides a better safeguard against the algorithm becoming trapped in local optima than in the case where the population is overly small. Finally, the number of competitors in a tournament has an effect on the rate of convergence. Although larger tournament sizes have a beneficial effect on convergence, it may be seen from Figure 7.3(e) that one soon reaches the point of diminishing returns with an increase in tournament size. In general, larger groups of competitors lead to a higher probability that high quality chromosomes will be selected (since the best chromosome in a tournament is selected), but it will also result in an increase in computational complexity, since the members of the population are considered more often in larger tournaments. The plot in Figure 7.3(e), suggests that a tournament size of 7 is a good choice (where  $s$  denotes the number of competitors involved in a tournament). The optimal parameter values were taken into consideration when the genetic algorithm was applied to the random graphs. Local optimization was used, with a populations of 50 members. Tournaments were between 7 competitors at a time, and the mutation rate was set to 0.5%. The averages of the minimum, average, and maximum fitness values (*i.e.*, crossing numbers bounds) of chromosomes were plotted against the number of generations. In general, only about seven generations were required for convergence, and it seems that at most ten generations are necessary to ensure convergence.

### 7.1.3 Convergence of the tabu search algorithm

First, the efficacy of the preconditioning techniques was tested. In addition to Nicholson's heuristic, and the Hamiltonian cycle heuristic, a preconditioner which generates random vertex arrangements was implemented. Each of these three methods were applied individually to the initial solutions to tabu searches that were executed for the hypercube  $Q_6$  of order 64, the balanced toroidal grid graph  $C_8 \times C_8$  of order 64, and a random, connected graph  $\mathcal{G}^*$  of order 32 and size 80. The results of these trials are shown in Figure 7.4.

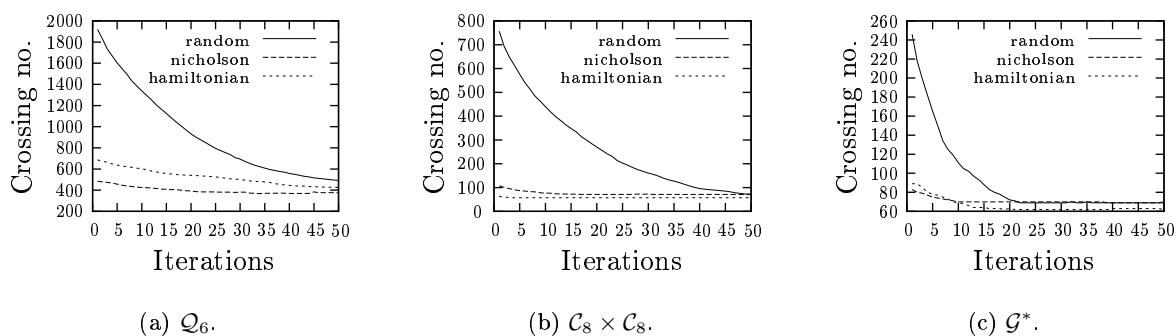
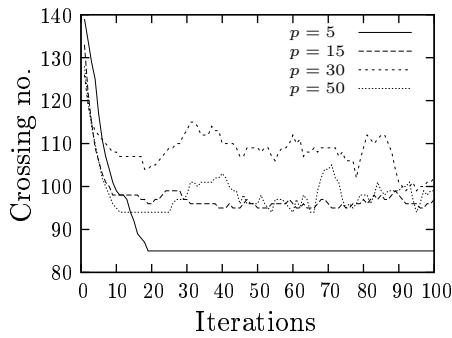


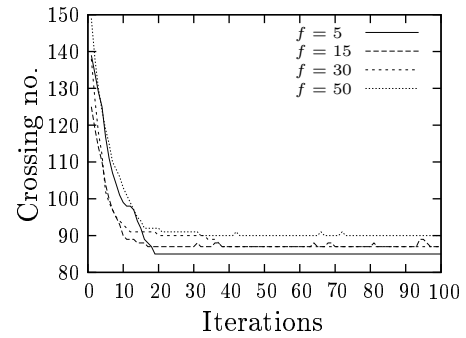
Figure 7.4: An illustration of the effects of preconditioning methods on tabu searches.

As may be seen from Figure 7.4, it is hardly ever beneficial to commence a tabu search with a

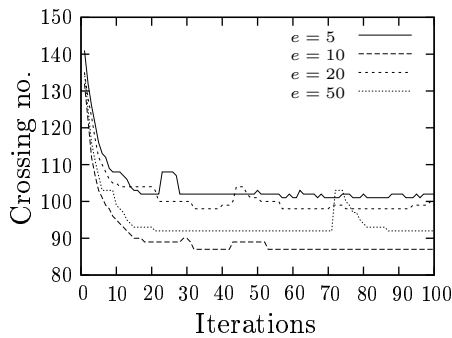
random vertex arrangement. As to the question of which of the other two heuristics is the best, no definite answer can be given. The heuristic of Nicholson performed better on the hypercube in Figure 7.4(a) than the Hamiltonian cycle heuristic. However, this situation is turned around for the toroidal grid graph, as may be seen in Figure 7.4(b). Nicholson’s heuristic has the advantage of being computationally quite efficient, as opposed to the Hamiltonian cycle heuristic, where many trials might have to be performed before a satisfactory cycle is found.



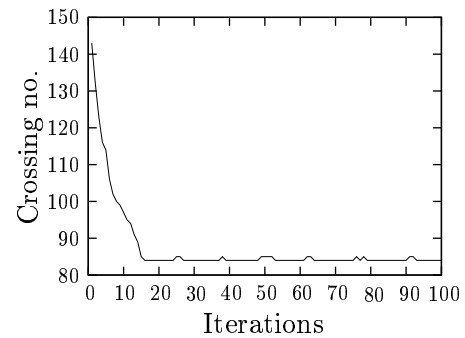
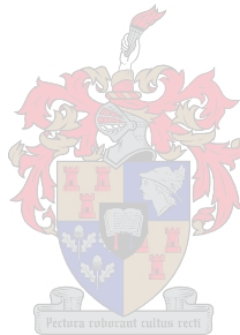
(a) Position poisoning tenure.



(b) High frequency penalties.



(c) Steps before elite solution reversion.



(d) Utilization of optimal parameters.

Figure 7.5: Illustration of the effect of the variation of tabu parameters on the convergence of solutions of a random, connected graph of order 32, and size 80.

It is difficult to provide a general set of parameters for the tabu algorithm that will yield acceptable results for all inputs. This is really a matter of experimentation for different classes of graphs. Denote the poison tenure of a position by  $p$ , the high frequency penalty as  $f$ , and the number of non-improving steps after which the tabu search algorithm will revert to an elite solution as  $e$ . Letting  $p = 5$ ,  $f = 5$  and  $e = 10$  works relatively well for small graphs. This parameter setup was mostly used in the enumeration of the drawings of the complete multipartite graphs of the next section.

If positions are poisoned for too long, convergence was found to be hampered. This is clearly visible in the different rates of convergence in Figure 7.5(a). The same occurs if the penalty for high frequency moves is too severe, as illustrated in Figure 7.5(b), although the effect is less pronounced than for position poisoning.

Elite solutions are revisited when no improving moves are found for a certain number of steps. This number of steps should preferably be quite large, since otherwise, the tabu algorithm will constantly be forced to return to old solutions, without having the chance to explore new regions

of the search space. The spikes in Figure 7.5(c) are indicative of where the tabu algorithm reverted to elite solutions. The fact that it performed relatively poorly for the cases where  $e = 20$  and  $e = 50$ , is probably not due to the choices of these parameters. In fact, the line representing the case  $e = 50$  seems to indicate that the algorithm was trapped in a local optimum. However, the case where  $e = 5$ , almost certainly does not provide the algorithm with enough time to scout, and hence the tiny oscillations that are due to constant reversions to elite solutions, that occurred in this case after 50 iterations.

The convergence for the tabu algorithm, with the parameters  $p = 5$ ,  $f = 5$  and  $e = 10$  is shown in Figure 7.5(d). In this case, it converged quickly, and it spent the rest of the time revisiting old solutions, which is evident from the fact that the spikes occur roughly at distances of 10 apart.

## 7.2 Algorithmic output

The output of the algorithms that were implemented, are considered in this section. The first subsection demonstrates a type of problem case that causes the **GreedySide** algorithm to perform poorly. In the second subsection, the output of the lower algorithm is considered. The third subsection provides an example of the output of the Garey–Johnson and finally, the last subsection contains a catalogue of drawings of all non–planar complete multipartite graphs of orders 6–13 realising upper bounds on their crossing numbers as determined by the tabu search algorithm.

### 7.2.1 Difficult problem instances for the GreedySide algorithm

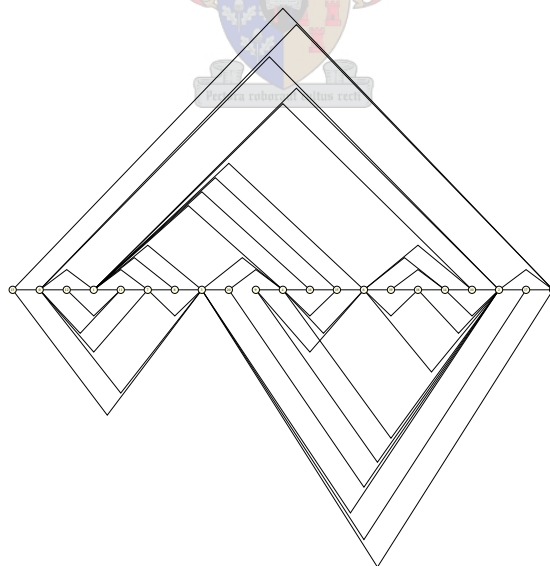


Figure 7.6:  $\mathcal{G}^{**}$

Although the **GreedySide** algorithm performs just as well as the Cimikowski–Shope algorithm in most instances, it performs very poorly on subdivided graphs. Consider the drawings in Figures 7.6 and 7.7, which were generated by the layout algorithm — the drawing of Figure 7.6 is a two–page layout of a graph  $\mathcal{G}^{**}$ , and the drawing of Figure 7.7 is a subdivided version of the graph  $\mathcal{G}^{**}$ , which was generated according to the subdivision method, described in § 5.3. The minimum edge layouts for each of these graphs will both contain the same number of crossings,

since by the method of construction of the subdivided graph, it cannot contain any additional crossings beyond those that were already present in  $\mathcal{G}^{**}$ .

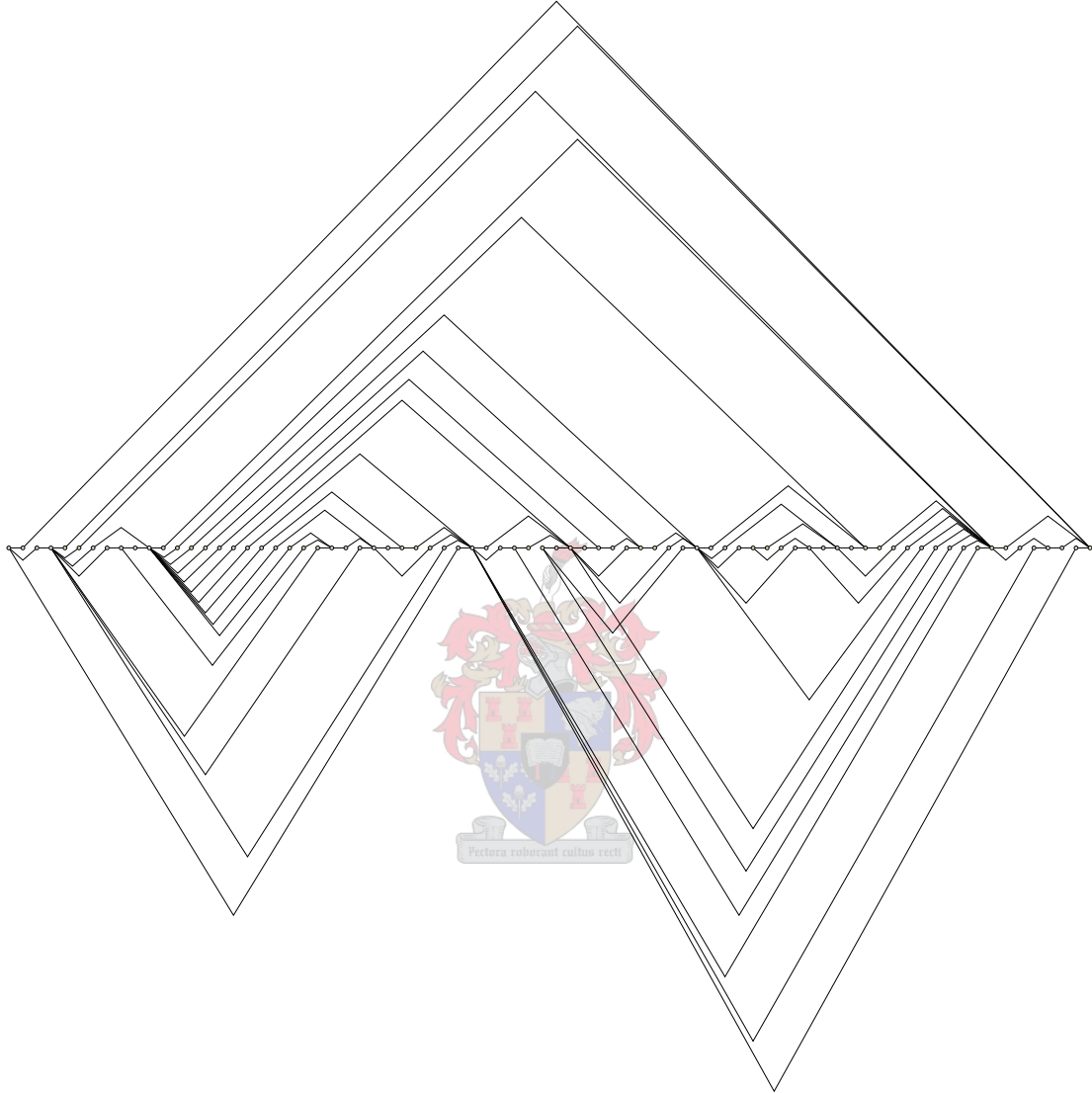


Figure 7.7: A subdivision of  $\mathcal{G}^{**}$

The edge layouts were randomized for both graphs, and the **GreedySide** and neural network algorithm were each applied to both graphs. For the graph  $\mathcal{G}^{**}$ , the **GreedySide** algorithm managed to find a layout with a single crossing, and neural network algorithm fared slightly worse, obtaining a layout with three crossings. However, the **GreedySide** algorithm could only find a layout for the subdivision of  $\mathcal{G}^{**}$  with 23 crossings. The neural network algorithm again found a layout with 3 crossings for the subdivided graph.

This indicates that the **GreedySide** algorithm is unsuitable for obtaining edge layouts in cases where it is possible that a subdivision of a graph  $\mathcal{G}$  would permit a book layout with a lower number of crossings than  $\mathcal{G}$  may be drawn within a book.

Dimension	8	9	10	11
Best known upper bound	8,192	36,032	153,088	636,160
Algorithmic result	306	2,395	6,787	18,329
Best known lower bound	597	1,536	3,840	84,787

Table 7.1: Lower and upper bound results for hypercubes.

### 7.2.2 Output of the lower bound algorithm

The lower bound algorithm was applied to the hypercubes of dimensions 8, 9, 10 and 11. Graph-to-graph embedding of the complete graphs into the hypercubes did not yield much success. The best results were achieved by repeated graph-to-graph embedding of bipartite graphs with 64–300 vertices into the hypercubes, and by feeding back their crossing number results as they improved, as explained in § 6.1.1.2. The results are shown in Table 7.1 and are compared to the best known upper and lower bounds for the hypercubes. The upper bound values were computed using the bound

$$\nu(\mathcal{Q}_n) \leq (165/1024)4^d - (2d^2 - 11d + 34)2^{d-3},$$

due to Faria and Herrera de Figueiredo [FdF00]. The lower bounds on  $\mathcal{Q}_8$  and  $\mathcal{Q}_9$  were computed using the bound

$$\frac{1}{24}d(d-1)2^d < \nu(\mathcal{Q}_d),$$

which is due to Madej [Mad91]. Finally, the lower bounds on  $\mathcal{Q}_8$  and  $\mathcal{Q}_9$  were computed with the bound

$$(1/20)4^d - (d^2 + 1)2^{d-1} < \nu(\mathcal{Q}_d),$$

which was derived by Sýkora and Vrto [SV93].

The results are not ground breaking by any means, but it must be remembered that, to the best knowledge of the author, these are the first results obtained by a general algorithmic method for bounding the crossing number of a graph from below. There are potentially many improvements that could be made (*i.e.*, improved edge mappings, better graphs for embedding into the hypercubes, *etc.*), and the fact that the algorithm has improved the best known lower bounds for  $\mathcal{Q}_9$  and  $\mathcal{Q}_{10}$  is hopefully proof that it has promise. Choosing an appropriate graph to graph-to-graph embed into the graph of which a lower bound needs to be computed is difficult, since the quality of the bound is highly sensitive to the graph that is embedded. For example, the graph  $\mathcal{K}_{58,60}$  has worked well in obtaining lower bounds to  $\mathcal{Q}_9$ , whilst the graph  $\mathcal{K}_{50,60}$  on the other hand, has delivered more mediocre results.

### 7.2.3 Output of the Garey–Johnson algorithm

The Garey–Johnson algorithm was implemented using the concept of independent crossing sets, and symmetry considerations for complete graphs. It was observed that the implementation of the algorithm consumed large amounts of memory and the computer that was used for simulations ran out of memory during the course of the execution of the algorithm for all but the smallest graphs. It has, however, been verified that the algorithm can find a drawing of  $\mathcal{K}_6$  realising its crossing number; such a drawing may be seen in Figure 7.8.

The algorithm first lists the edges involved in a crossing configuration, followed by the individual independent crossing subgraphs that may be constructed for the crossing configuration. Recall that all independent crossing subgraphs are initially contracted (§ 5.2.4.3), and that a crossing

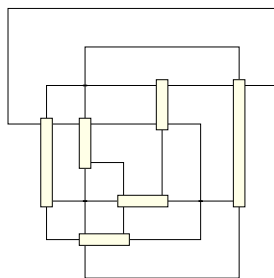


Figure 7.8: A drawing of  $\mathcal{K}_6$  generated by the Garey–Johnson algorithm

subgraph  $\mathcal{H}$  is only expanded when the graph in which  $\mathcal{H}$  is contracted, is found to be planar. Once  $\mathcal{H}$  is expanded, each of its edges are considered recursively, as described in § 5.2. The recursion level of the algorithm at a given point is indicated by the number of full stops preceding a line of output.

When the algorithm expands an independent crossing subgraph, it prints the line **Expanding independent crossing subgraph with edges**, followed by a line that lists the edges in the independent crossing subgraph. For each edge  $e$ , all permutations of crossings in which  $e$  is involved, are enumerated, and for each permutation, the next edge in the independent crossing subgraph is considered in the same fashion. For each permutation that is considered for an edge  $e$ , the algorithm prints the line **considering edge e with crossing permutation**, followed by the permutation of edges that cross  $e$ , on the next line. When the recursion process has descended to its lowest level, a subgraph corresponding to the independent crossing subgraph, as described in the previous paragraph, is constructed. The algorithm indicates this by printing the line **Constructing crossings for independent crossing subgraph with edges**, followed by a line containing the edges of the independent crossing subgraph.

An example of output from the Garey–Johnson algorithm, as applied to the graph  $\mathcal{K}_6$ , is provided below. In the example, the algorithm had to find a drawing of  $\mathcal{K}_6$  containing 5 crossing. The full output of the algorithm spans several hundred pages; therefore only the output corresponding to two crossing configurations is given.

The first example depicts a scenario where the given choice of edge crossings cannot lead to a planar configuration. The output was truncated, due to its length. One may clearly see how the different permutations of crossings of the edges  $e_1, e_2$  and  $e_9$  with the edge  $e_0$  is enumerated by the algorithm. This occurs at a recursion depth of two, *i.e.*, on lines which contain two full stops at the beginning.

Edges crossed:

```
e_0 is crossed by e_1, e_2, e_9
e_1 is crossed by e_0, e_12
e_2 is crossed by e_0
e_7 is crossed by e_11
e_9 is crossed by e_0
e_11 is crossed by e_7
e_12 is crossed by e_1
```

```
constructing independent crossing subgraphs with edges
e_9 e_0 e_1 e_12 e_2
```

```
constructing independent crossing subgraphs with edges
```



e\_7 e\_11

Expanding independent crossing subgraph with edges

e\_9 e\_0 e\_1 e\_12 e\_2

. considering edge e\_9 with crossing permutation

. e\_0

. . considering edge e\_0 with crossing permutation

. . e\_1 e\_2 e\_9

. . . considering edge e\_1 with crossing permutation

. . . e\_0 e\_12

. . . . considering edge e\_12 with crossing permutation

. . . . e\_1

. . . . . considering edge e\_2 with crossing permutation

. . . . . e\_0

. . . . . Constructing crossings for independent crossing subgraph with edges

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . . . Search was unsuccessful.

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . considering edge e\_1 with crossing permutation

. . . e\_12 e\_0

. . . . considering edge e\_12 with crossing permutation

. . . . e\_1

. . . . . considering edge e\_2 with crossing permutation

. . . . . e\_0

. . . . . Constructing crossings for independent crossing subgraph with edges

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . . . Search was unsuccessful.

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . considering edge e\_0 with crossing permutation

. . e\_1 e\_9 e\_2

. . . considering edge e\_1 with crossing permutation

. . . e\_0 e\_12

. . . . considering edge e\_12 with crossing permutation

. . . . e\_1

. . . . . considering edge e\_2 with crossing permutation

. . . . . e\_0

. . . . . Constructing crossings for independent crossing subgraph with edges

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . . . Search was unsuccessful.

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . considering edge e\_1 with crossing permutation

. . . e\_12 e\_0

. . . . considering edge e\_12 with crossing permutation

. . . . e\_1

. . . . . considering edge e\_2 with crossing permutation

. . . . . e\_0

. . . . . Constructing crossings for independent crossing subgraph with edges

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . . . . Search was unsuccessful.

. . . . . e\_9 e\_0 e\_1 e\_12 e\_2

. . considering edge e\_0 with crossing permutation

. . e\_2 e\_1 e\_9

. . . considering edge e\_1 with crossing permutation

. . . e\_0 e\_12

. . . . considering edge e\_12 with crossing permutation

. . . . e\_1

```

. . . . . considering edge e_2 with crossing permutation
. . . . . e_0
. . . . . Constructing crossings for independent crossing subgraph with edges
. . . . . e_9 e_0 e_1 e_12 e_2
. . . . . Search was unsuccessful.
. . . . . e_9 e_0 e_1 e_12 e_2
. . . considering edge e_1 with crossing permutation
. . . e_12 e_0
. . . . considering edge e_12 with crossing permutation
. . . . e_1
. . . . . considering edge e_2 with crossing permutation
. . . . . e_0
. . . . . Constructing crossings for independent crossing subgraph with edges
. . . . . e_9 e_0 e_1 e_12 e_2
. . . . . Search was unsuccessful.
. . . . . e_9 e_0 e_1 e_12 e_2
(output truncated)

```

In the second example, the crossing configuration yields a set of edge crossings that permits a planar graph  $\mathcal{G}$ , containing artificial vertices modelling crossings, to be constructed from  $\mathcal{K}_6$ . A plane drawing of  $\mathcal{G}$  then represents a drawing of  $\mathcal{K}_6$  with 5 crossings. At the recursion depth of five (*i.e.*, the lines which start with five full stops), the second independent crossing subgraph is expanded, after the algorithm found that the constructed subgraph corresponding to the first independent crossing subgraph was planar.

Edges crossed:

```

e_0 is crossed by e_1, e_2, e_9
e_1 is crossed by e_0
e_2 is crossed by e_0
e_3 is crossed by e_11
e_8 is crossed by e_12
e_9 is crossed by e_0
e_11 is crossed by e_3
e_12 is crossed by e_8

```



```

constructing independent crossing subgraphs with edges
e_9 e_0 e_1 e_2

```

```

constructing independent crossing subgraphs with edges
e_8 e_12 e_3 e_11

```

```

Expanding independent crossing subgraph with edges
e_9 e_0 e_1 e_2
. considering edge e_9 with crossing permutation
. e_0
. . considering edge e_0 with crossing permutation
. . e_1 e_2 e_9
. . . considering edge e_1 with crossing permutation
. . . e_0
. . . . considering edge e_2 with crossing permutation
. . . . e_0

```

```

. . . . . Constructing crossings for independent crossing subgraph with edges
. . . . . e_9 e_0 e_1 e_2
. . . . . Expanding independent crossing subgraph with edges
. . . . . e_8 e_12 e_3 e_11
. . . . . considering edge e_8 with crossing permutation
. . . . . e_12
. . . . . considering edge e_12 with crossing permutation
. . . . . e_8
. . . . . considering edge e_3 with crossing permutation
. . . . . e_11
. . . . . considering edge e_11 with crossing permutation
. . . . . e_3
. . . . . Constructing crossings for independent crossing subgraph
. . . . . with edges
. . . . . e_8 e_12 e_3 e_11

```

#### 7.2.4 Results for complete multipartite graphs

All complete multipartite graphs of orders 6–13 were generated, with the exception of the complete multipartite graphs that were isomorphic to the complete graphs. The tabu search algorithm was used to determine the upper bounds on the crossing numbers of these graphs in the plane, and it employed either the **GreedySide** algorithm, or Cimikowski and Shope’s neural network algorithm [CS96] to compute the edge layouts for the graphs, given vertex arrangements. After the initial application of the tabu search algorithm, the graphs were subdivided once (*i.e.*, a single subdivision vertex was inserted into each edge), after which the tabu search algorithm was applied again, except that it used only the neural network algorithm of Cimikowski and Shope for the subdivided cases, due to poor handling by the **GreedySide** algorithm of such cases (as demonstrated in § 7.2.1). Although the genetic algorithm is competitive with these two algorithms in terms of solution quality, its prohibitive computational complexity caused it to be deemed too inefficient for computations on this large data set. The graph layouts were computed by the orthogonal layout procedure in the LEDA [LED] software library.

With the exception of a single graph,  $\mathcal{K}_{1,2,2,2,2,2}$ , the subdivided graphs did not yield drawings with fewer crossings than their non-subdivided counterparts. In fact, for all of these graphs, both the **GreedySide** algorithm and the neural network algorithm found layouts achieving the stated bounds. In the case of  $\mathcal{K}_{1,2,2,2,2}$ , the subdivision drawing improves upon the upper bound of the non-subdivided counterpart only by a single crossing. The drawings were constructed using Algorithm 6.9.

A lower bound on the crossing number of a graph has the property of being hereditary under the supergraph relation. That is, if  $\mathcal{H}$  is a supergraph of  $\mathcal{G}$  (or equivalently, if  $\mathcal{G}$  is a subgraph of  $\mathcal{H}$ ), then a lower bound on  $\nu(\mathcal{G})$  is also a lower bound on  $\nu(\mathcal{H})$  — this is trivially true, because a drawing of  $\mathcal{H}$  must contain a drawing of  $\mathcal{G}$ . The subgraph relations of all of the generated complete multipartite graphs were computed using the method described in § B.5.3. For a complete multipartite graph  $\mathcal{H}$ , if a lower bound on  $\nu(\mathcal{H})$  was unknown, such a bound was computed as

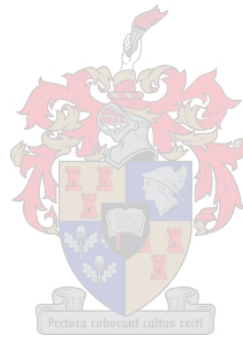
$$\nu(\mathcal{H}) \geq \max_{\mathcal{G} \text{ is a subgraph of } \mathcal{H}} \{\nu(\mathcal{G})\}.$$

This led to a recursive situation in cases where an analytical bound on  $\nu(\mathcal{G})$  was unknown. The lower bounds for crossing numbers that were used, were those for bipartite graphs and

for complete multipartite graphs in § 4.2.3.2, as well as the crossing number bounds for the complements of cycles in § 4.2.3.7, since these graphs are subgraphs of complete multipartite graphs that are obtained by the deletion of single edges.

In each of the drawings, vertices in the same partite set received the same label. Labelling always commenced at zero and the partite sets were labelled in increasing order with respect to their sizes, starting with the smallest partite sets.

The multipartite graph notation, in which a list of partite set cardinalities is written as the subscript of the letter  $\mathcal{K}$ , becomes unwieldy for larger graphs. For example, the graph resulting from the deletion of an edge from  $\mathcal{K}_{10}$  is a multipartite graph, denoted  $\mathcal{K}_{1,1,1,1,1,1,1,1,2}$ . For this reason, an alternative (non-standard) notation was employed in this thesis to ease readability. For  $n > 3$ , the presence of  $n$  multipartite sets of order  $p$  in a multipartite graph was denoted  $n \times p$  in the list of partite set cardinalities. Using this notation, the graph  $\mathcal{K}_{10}$  with an edge deleted, is denoted  $\mathcal{K}_{8 \times 1, 2}$ .



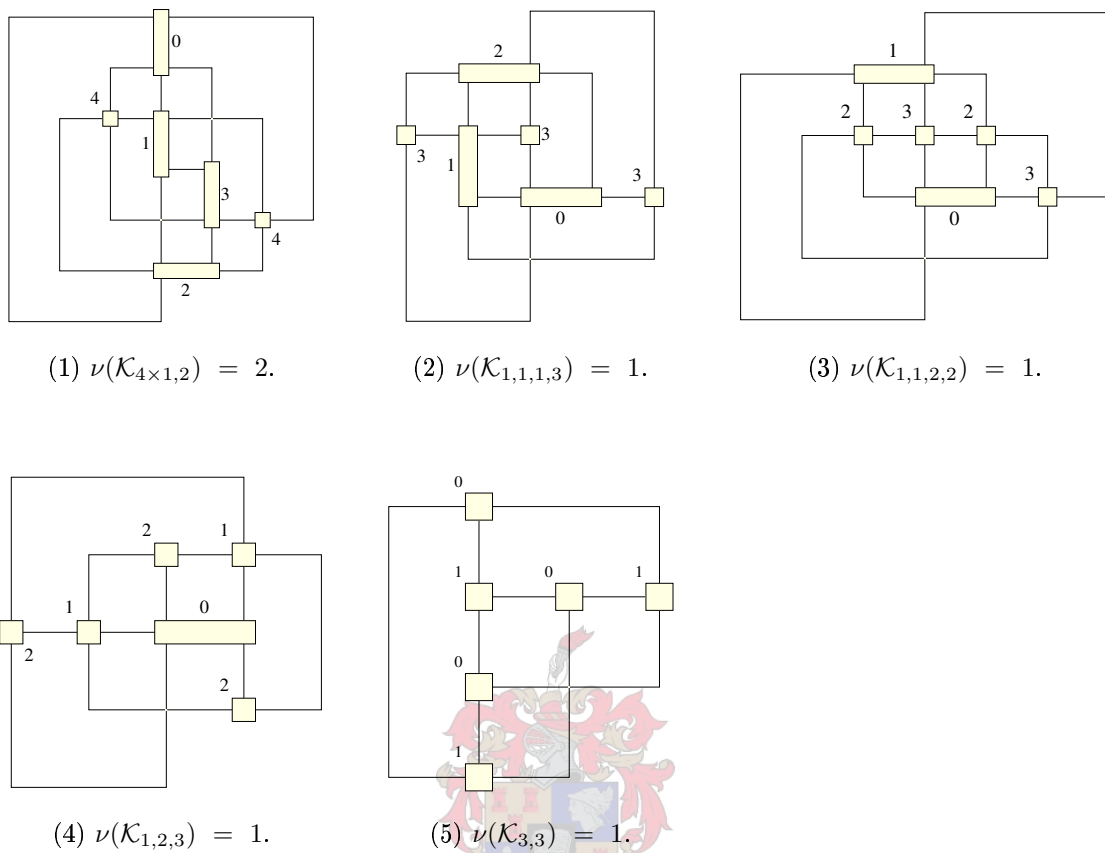


Figure 7.9: Non-planar complete multipartite graphs of order 6.

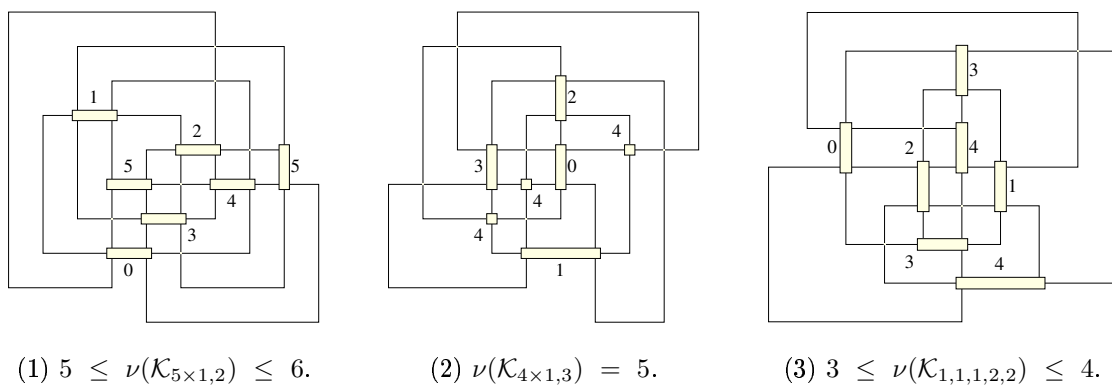
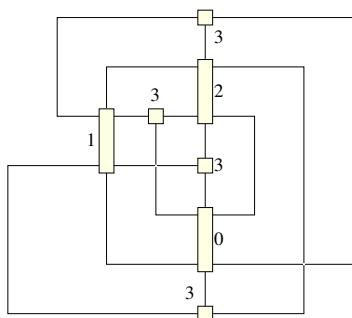
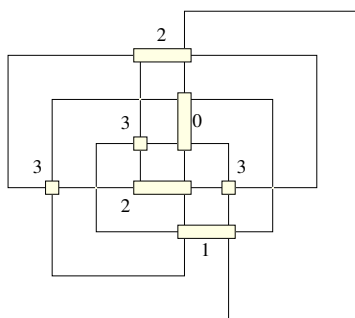


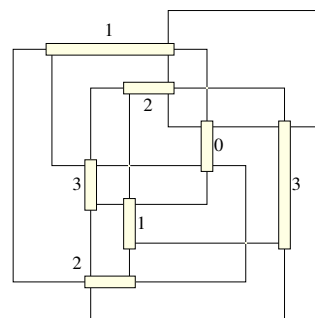
Figure 7.10: Non-planar complete multipartite graphs of order 7.



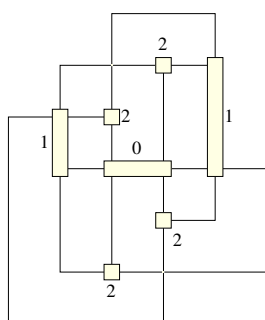
(4)  $\nu(\mathcal{K}_{1,1,1,4}) = 2$ .



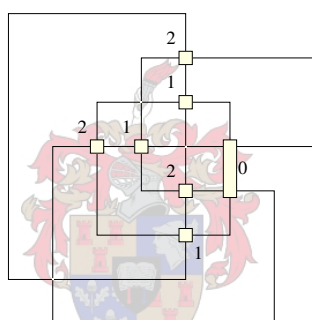
(5)  $\nu(\mathcal{K}_{1,1,2,3}) = 3$ .



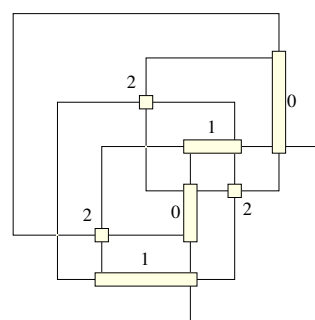
(6)  $2 \leq \nu(\mathcal{K}_{1,2,2,2}) \leq 3$ .



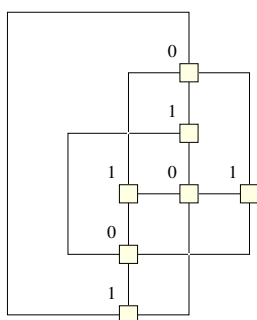
(7)  $\nu(\mathcal{K}_{1,2,4}) = 2$ .



(8)  $\nu(\mathcal{K}_{1,3,3}) = 3$ .

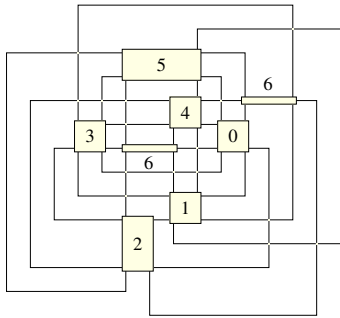


(9)  $\nu(\mathcal{K}_{2,2,3}) = 2$ .

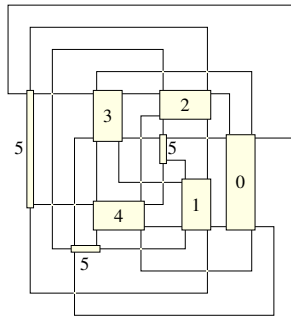


(10)  $\nu(\mathcal{K}_{3,4}) = 2$ .

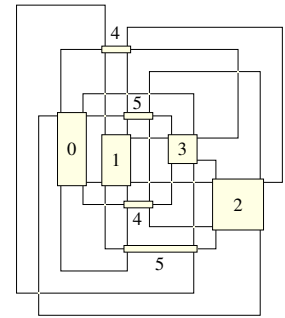
Figure 7.10 (continued): Non-planar complete multipartite graphs of order 7.



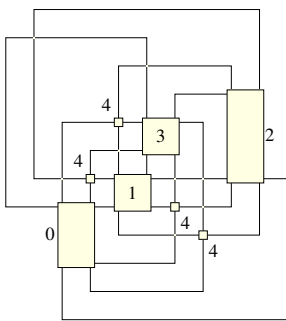
(1)  $9 \leq \nu(\mathcal{K}_{6 \times 1, 2}) \leq 15$ .



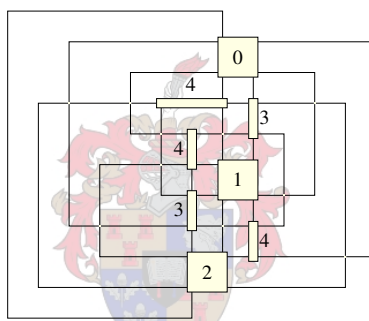
(2)  $8 \leq \nu(\mathcal{K}_{5 \times 1, 3}) \leq 12$ .



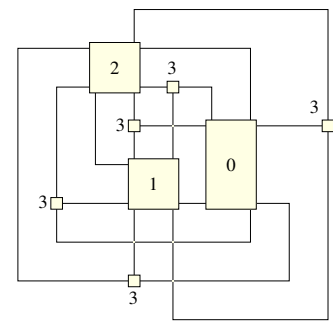
(3)  $8 \leq \nu(\mathcal{K}_{4 \times 1, 2, 2}) \leq 12$ .



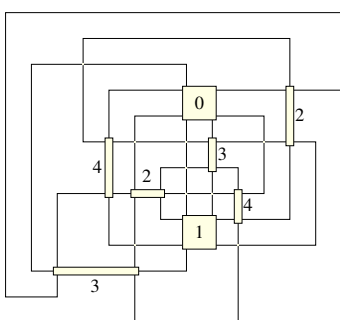
(4)  $\nu(\mathcal{K}_{4 \times 1, 4}) = 8$ .



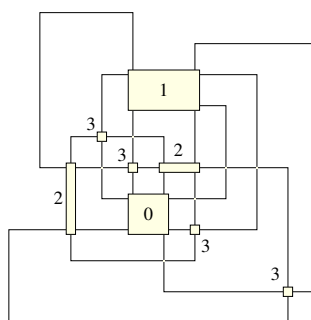
(5)  $7 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 3}) \leq 10$ .



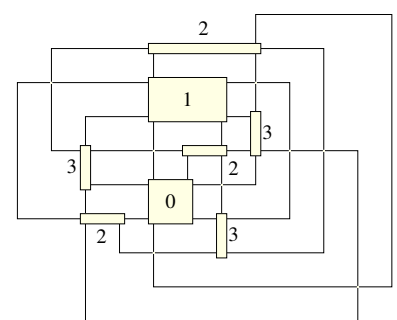
(6)  $\nu(\mathcal{K}_{1, 1, 1, 5}) = 4$ .



(7)  $7 \leq \nu(\mathcal{K}_{1, 1, 2, 2, 2}) \leq 9$ .

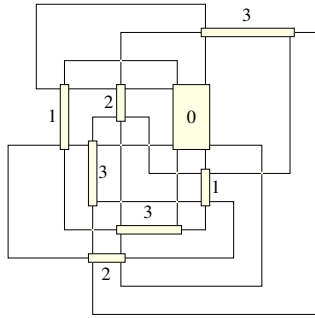


(8)  $\nu(\mathcal{K}_{1, 1, 2, 4}) = 6$ .

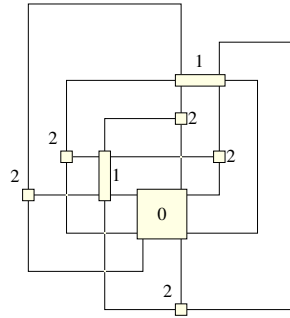


(9)  $7 \leq \nu(\mathcal{K}_{1, 1, 3, 3}) \leq 8$ .

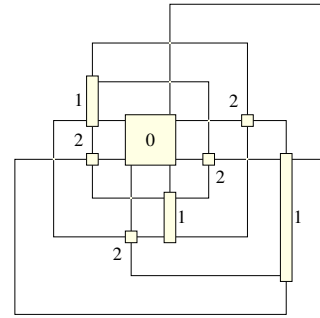
Figure 7.11: Non-planar complete multipartite graphs of order 8.



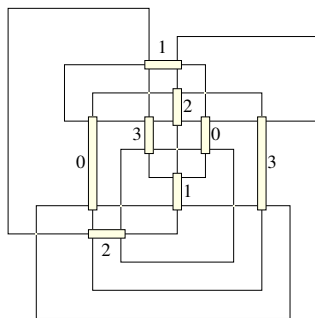
(10)  $7 \leq \nu(\mathcal{K}_{1,2,2,3}) \leq 8$ .



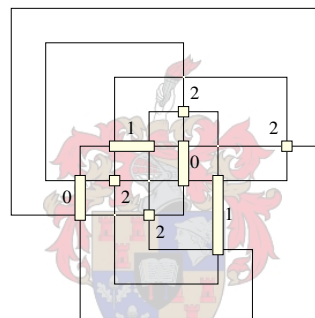
(11)  $\nu(\mathcal{K}_{1,2,5}) = 4$ .



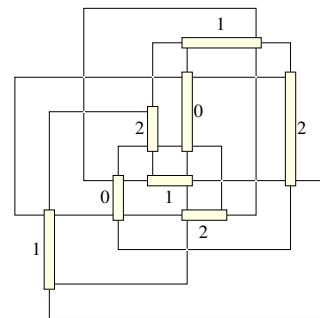
(12)  $\nu(\mathcal{K}_{1,3,4}) = 6$ .



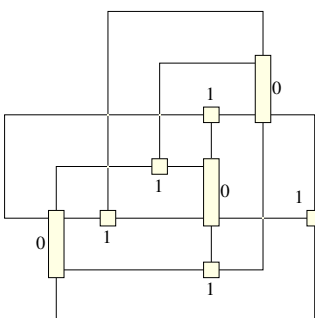
(13)  $4 \leq \nu(\mathcal{K}_{4 \times 2}) \leq 6$ .



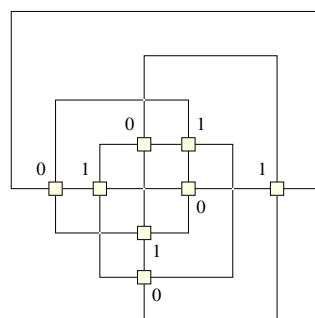
(14)  $\nu(\mathcal{K}_{2,2,4}) = 4$ .



(15)  $\nu(\mathcal{K}_{2,3,3}) = 7$ .



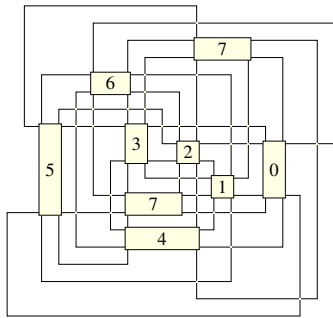
(16)  $\nu(\mathcal{K}_{3,5}) = 4$ .



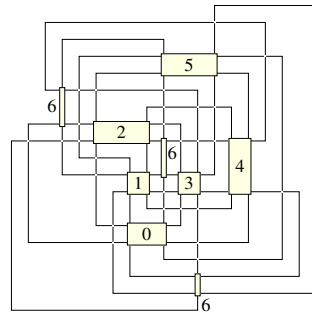
(17)  $\nu(\mathcal{K}_{4,4}) = 4$ .

Figure 7.11 (continued): Non-planar complete multipartite graphs of order 8.

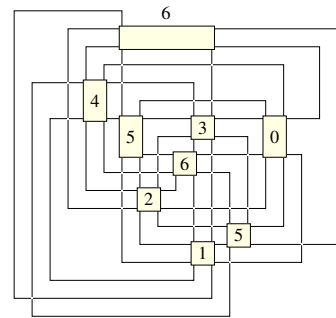




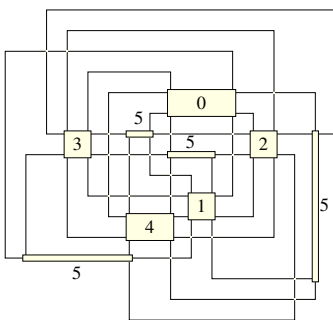
(1)  $18 \leq \nu(\mathcal{K}_{7 \times 1, 2}) \leq 30$ .



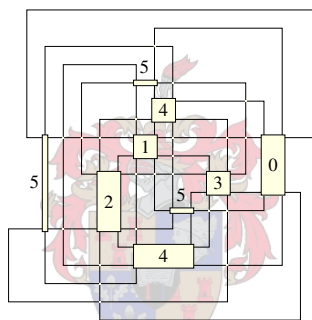
(2)  $13 \leq \nu(\mathcal{K}_{6 \times 1, 3}) \leq 27$ .



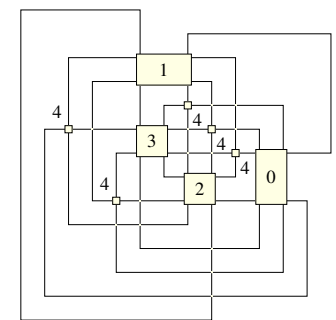
(3)  $13 \leq \nu(\mathcal{K}_{5 \times 1, 2, 2}) \leq 25$ .



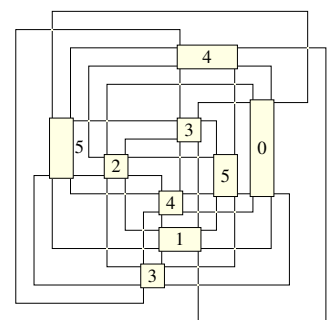
(4)  $13 \leq \nu(\mathcal{K}_{5 \times 1, 4}) \leq 19$ .



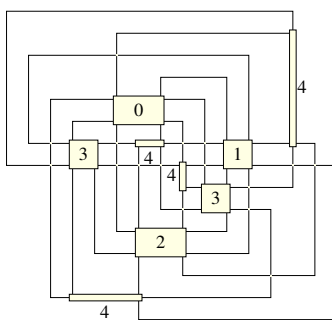
(5)  $13 \leq \nu(\mathcal{K}_{4 \times 1, 2, 3}) \leq 22$ .



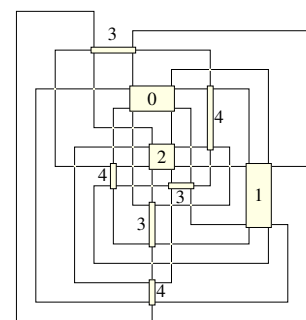
(6)  $\nu(\mathcal{K}_{4 \times 1, 5}) = 13$ .



(7)  $12 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 2, 2}) \leq 21$ .

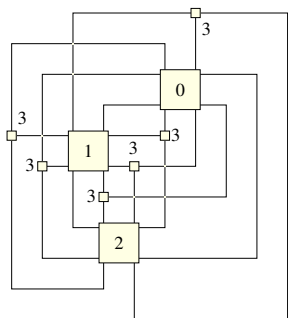


(8)  $12 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 4}) \leq 16$ .

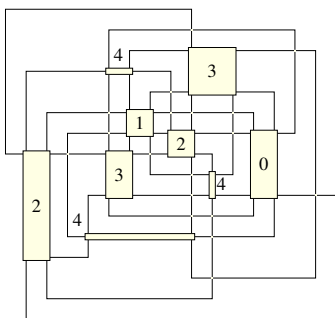


(9)  $12 \leq \nu(\mathcal{K}_{1, 1, 1, 3, 3}) \leq 20$ .

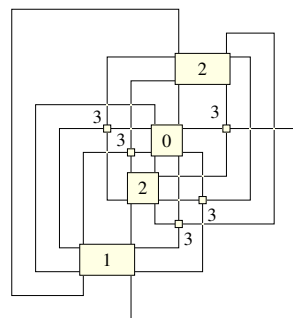
Figure 7.12: Non-planar complete multipartite graphs of order 9.



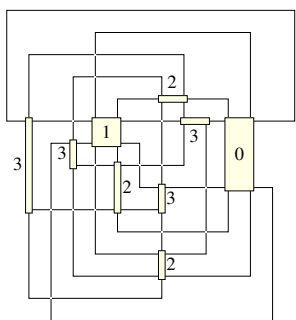
(10)  $\nu(\mathcal{K}_{1,1,1,6}) = 6$ .



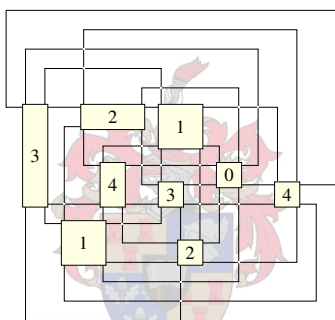
(11)  $12 \leq \nu(\mathcal{K}_{1,1,2,2,3}) \leq 18$ .



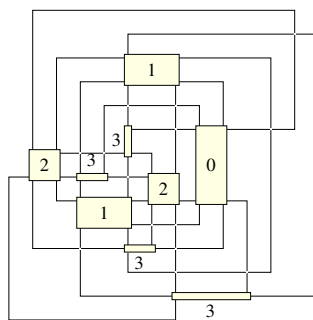
(12)  $\nu(\mathcal{K}_{1,1,2,5}) = 10$ .



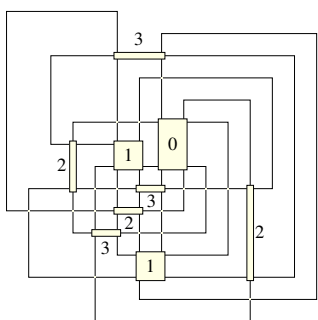
(13)  $12 \leq \nu(\mathcal{K}_{1,1,3,4}) \leq 14$ .



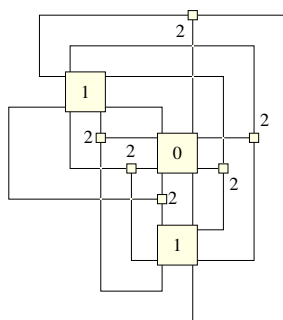
(14)  $12 \leq \nu(\mathcal{K}_{1,4 \times 2}) \leq 18$ .



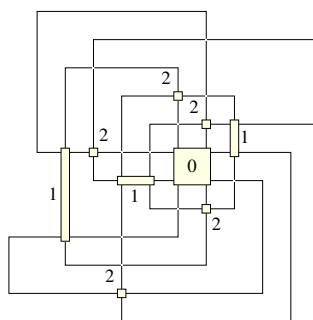
(15)  $12 \leq \nu(\mathcal{K}_{1,2,2,4}) \leq 14$ .



(16)  $12 \leq \nu(\mathcal{K}_{1,2,3,3}) \leq 16$ .

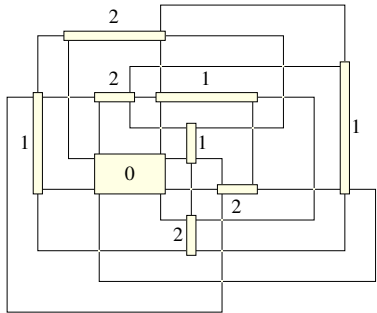


(17)  $\nu(\mathcal{K}_{1,2,6}) = 6$ .

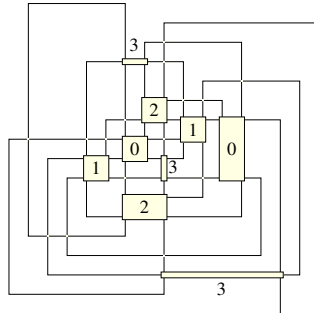


(18)  $\nu(\mathcal{K}_{1,3,5}) = 10$ .

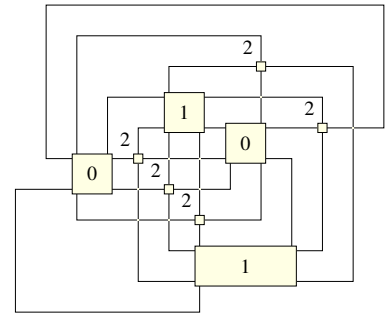
Figure 7.12 (continued): Non-planar complete multipartite graphs of order 9.



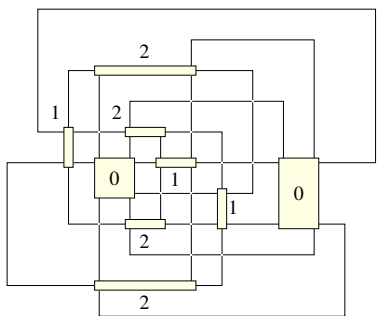
(19)  $8 \leq \nu(\mathcal{K}_{1,4,4}) \leq 12$ .



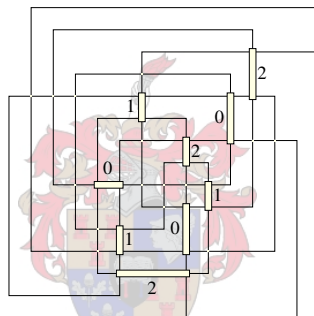
(20)  $12 \leq \nu(\mathcal{K}_{2,2,2,3}) \leq 15$ .



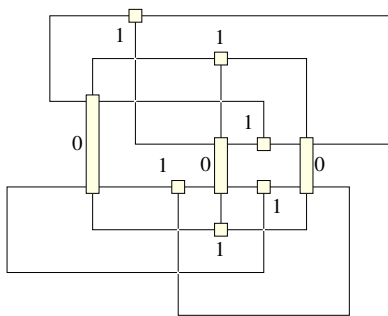
(21)  $\nu(\mathcal{K}_{2,2,5}) = 8$ .



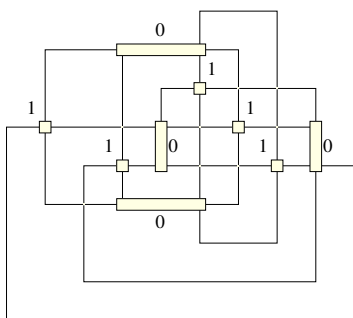
(22)  $\nu(\mathcal{K}_{2,3,4}) = 12$ .



(23)  $7 \leq \nu(\mathcal{K}_{3,3,3}) \leq 15$ .

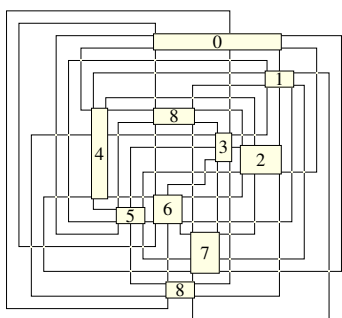


(24)  $\nu(\mathcal{K}_{3,6}) = 6$ .

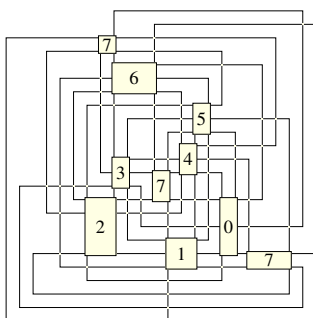


(25)  $\nu(\mathcal{K}_{4,5}) = 8$ .

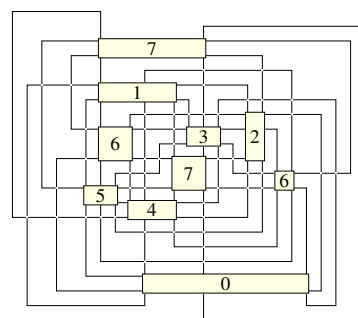
Figure 7.12 (continued): Non-planar complete multipartite graphs of order 9.



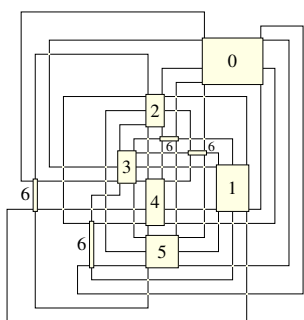
(1)  $36 \leq \nu(\mathcal{K}_{8 \times 1, 2}) \leq 54.$



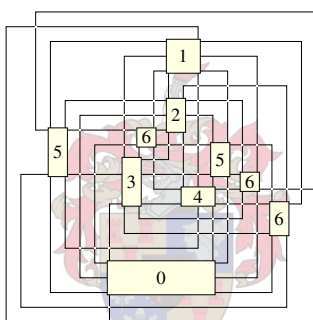
(2)  $21 \leq \nu(\mathcal{K}_{7 \times 1, 3}) \leq 48.$



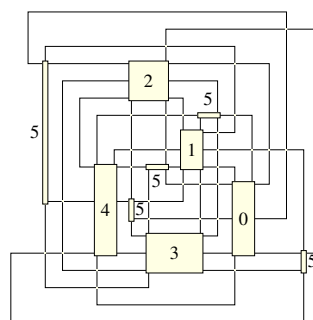
(3)  $21 \leq \nu(\mathcal{K}_{6 \times 1, 2, 2}) \leq 48.$



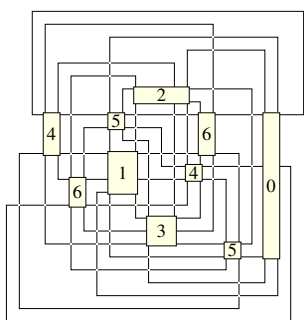
(4)  $21 \leq \nu(\mathcal{K}_{6 \times 1, 4}) \leq 39.$



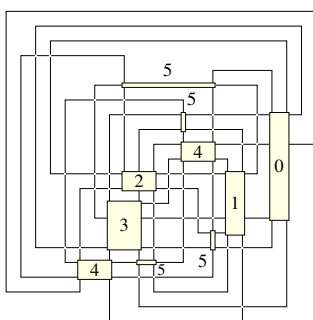
(5)  $21 \leq \nu(\mathcal{K}_{5 \times 1, 2, 3}) \leq 43.$



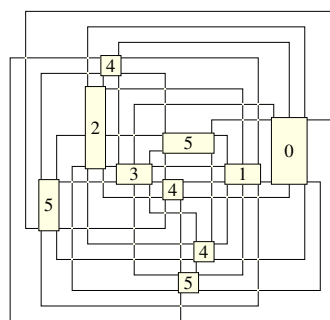
(6)  $21 \leq \nu(\mathcal{K}_{5 \times 1, 5}) \leq 29.$



(7)  $21 \leq \nu(\mathcal{K}_{4 \times 1, 2, 2, 2}) \leq 42.$

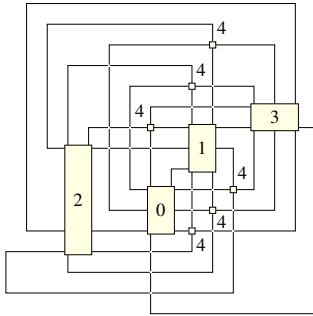


(8)  $21 \leq \nu(\mathcal{K}_{4 \times 1, 2, 4}) \leq 34.$

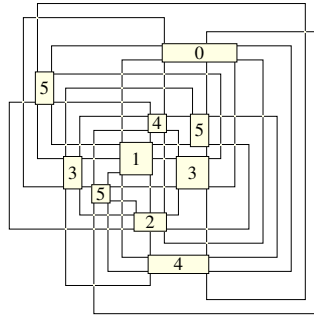


(9)  $21 \leq \nu(\mathcal{K}_{4 \times 1, 3, 3}) \leq 38.$

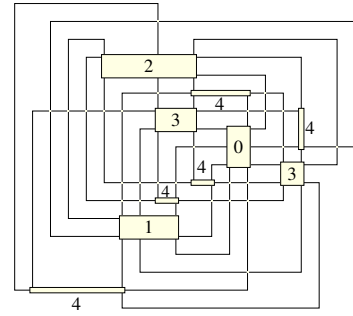
Figure 7.13: Non-planar complete multipartite graphs of order 10.



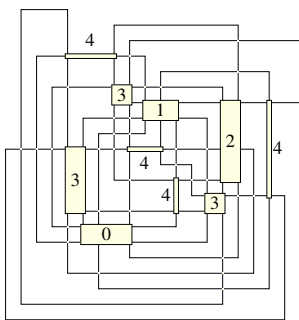
(10)  $\nu(\mathcal{K}_{4 \times 1, 6}) = 18$ .



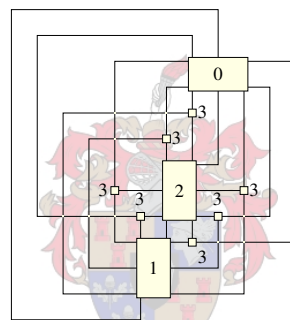
(11)  $21 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 2, 3}) \leq 38$ .



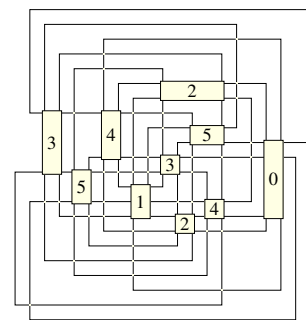
(12)  $21 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 5}) \leq 26$ .



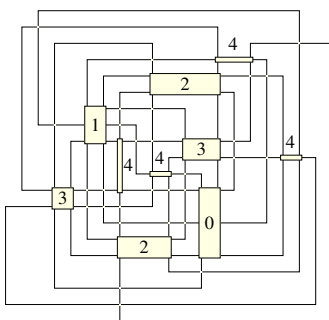
(13)  $21 \leq \nu(\mathcal{K}_{1, 1, 1, 3, 4}) \leq 31$ .



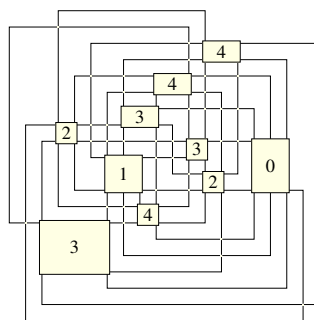
(14)  $\nu(\mathcal{K}_{1, 1, 1, 7}) = 9$ .



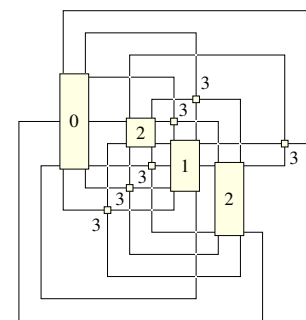
(15)  $21 \leq \nu(\mathcal{K}_{1, 1, 4 \times 2}) \leq 37$ .



(16)  $21 \leq \nu(\mathcal{K}_{1, 1, 2, 2, 4}) \leq 29$ .

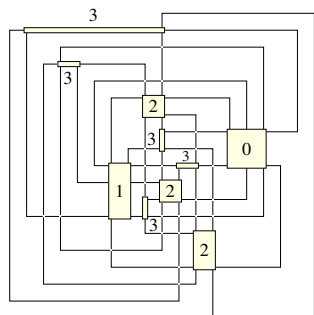


(17)  $21 \leq \nu(\mathcal{K}_{1, 1, 2, 3, 3}) \leq 34$ .

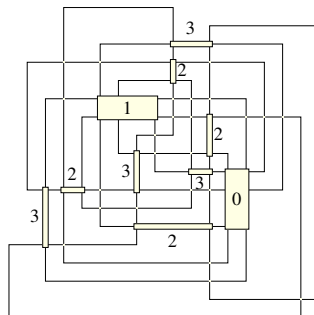


(18)  $\nu(\mathcal{K}_{1, 1, 2, 6}) = 15$ .

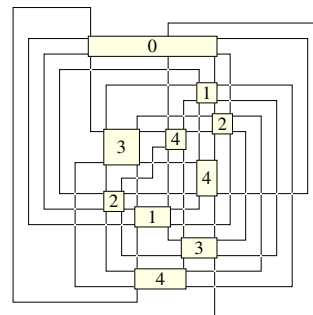
Figure 7.13 (continued): Non-planar complete multipartite graphs of order 10.



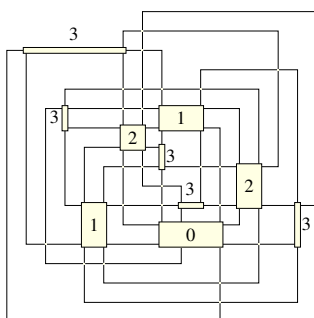
(19)  $21 \leq \nu(\mathcal{K}_{1,1,3,5}) \leq 23.$



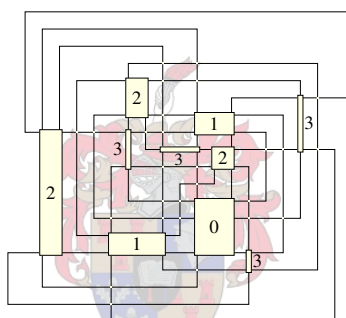
(20)  $16 \leq \nu(\mathcal{K}_{1,1,4,4}) \leq 24.$



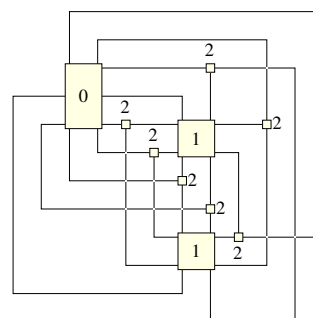
(21)  $21 \leq \nu(\mathcal{K}_{1,2,2,2,3}) \leq 33.$



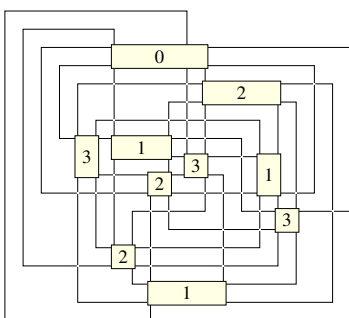
(22)  $21 \leq \nu(\mathcal{K}_{1,2,2,5}) \leq 23.$



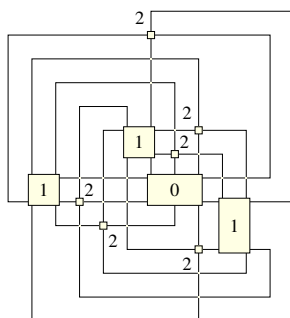
(23)  $21 \leq \nu(\mathcal{K}_{1,2,3,4}) \leq 27.$



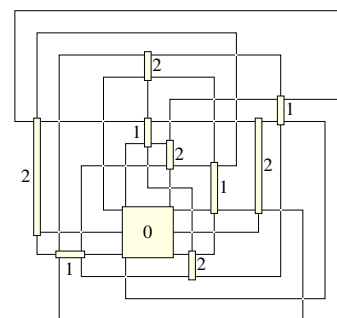
(24)  $\nu(\mathcal{K}_{1,2,7}) = 9.$



(25)  $15 \leq \nu(\mathcal{K}_{1,3,3,3}) \leq 30.$

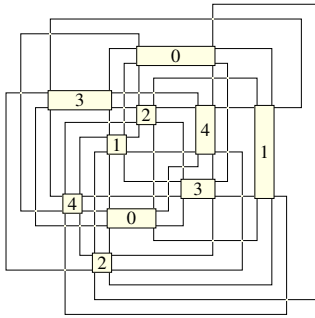


(26)  $\nu(\mathcal{K}_{1,3,6}) = 15.$

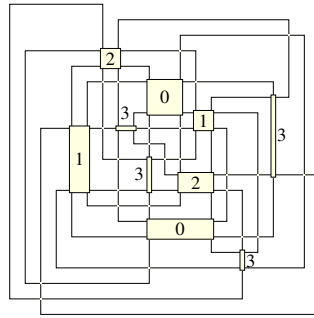


(27)  $16 \leq \nu(\mathcal{K}_{1,4,5}) \leq 20.$

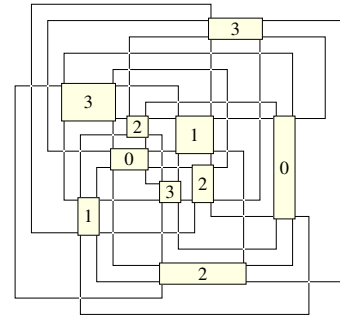
Figure 7.13 (continued): Non-planar complete multipartite graphs of order 10.



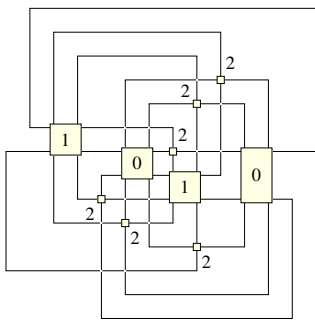
(28)  $12 \leq \nu(\mathcal{K}_{5 \times 2}) \leq 32$ .



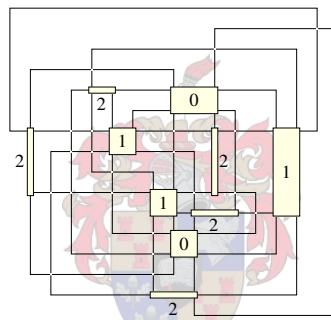
(29)  $12 \leq \nu(\mathcal{K}_{2,2,2,4}) \leq 24$ .



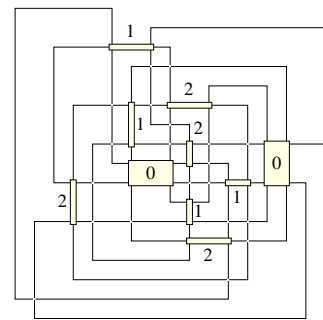
(30)  $21 \leq \nu(\mathcal{K}_{2,2,3,3}) \leq 30$ .



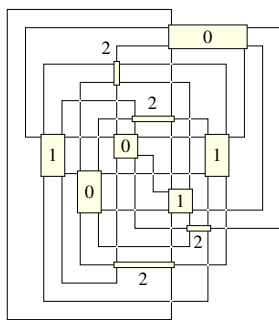
(31)  $\nu(\mathcal{K}_{2,2,6}) = 12$ .



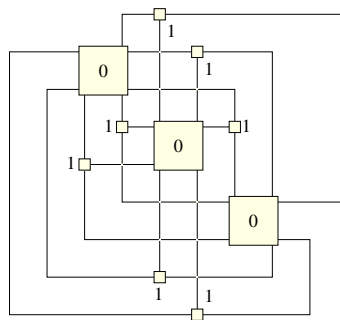
(32)  $\nu(\mathcal{K}_{2,3,5}) = 21$ .



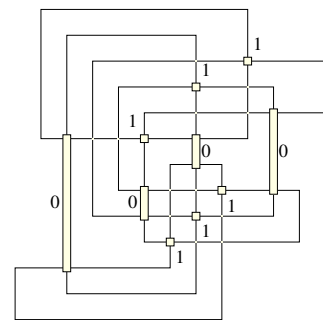
(33)  $12 \leq \nu(\mathcal{K}_{2,4,4}) \leq 20$ .



(34)  $12 \leq \nu(\mathcal{K}_{3,3,4}) \leq 25$ .

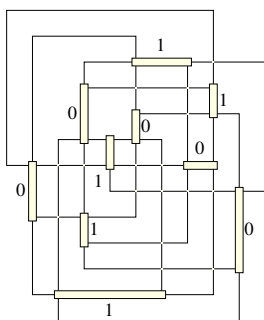


(35)  $\nu(\mathcal{K}_{3,7}) = 9$ .



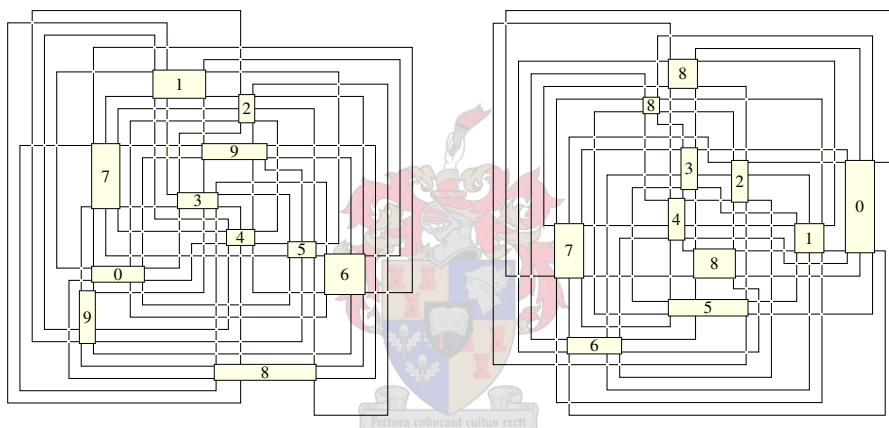
(36)  $\nu(\mathcal{K}_{4,6}) = 12$ .

Figure 7.13 (continued): Non-planar complete multipartite graphs of order 10.



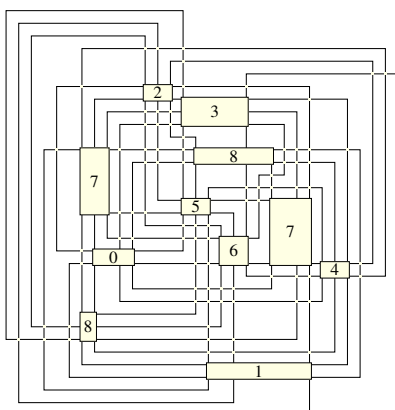
(37)  $\nu(\mathcal{K}_{5,5}) = 16$ .

Figure 7.13 (continued): Non-planar complete multipartite graphs of order 10.

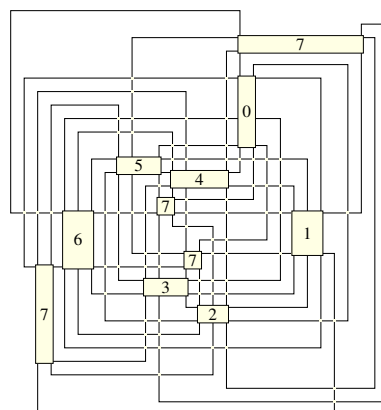


(1)  $79 \leq \nu(\mathcal{K}_{9 \times 1, 2}) \leq 90$ .

(2)  $36 \leq \nu(\mathcal{K}_{8 \times 1, 3}) \leq 84$ .



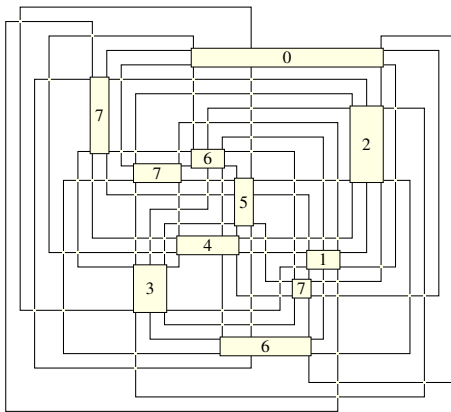
(3)  $36 \leq \nu(\mathcal{K}_{7 \times 1, 2, 2}) \leq 81$ .



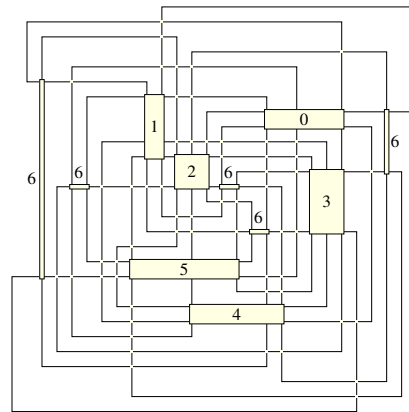
(4)  $30 \leq \nu(\mathcal{K}_{7 \times 1, 4}) \leq 69$ .

Figure 7.14: Non-planar complete multipartite graphs of order 11.

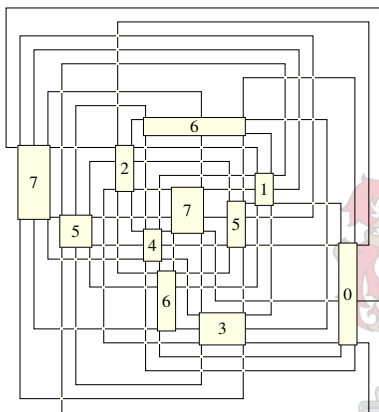




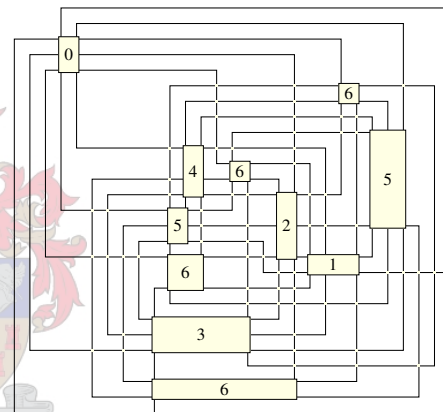
(5)  $30 \leq \nu(\mathcal{K}_{6 \times 1, 2, 3}) \leq 75.$



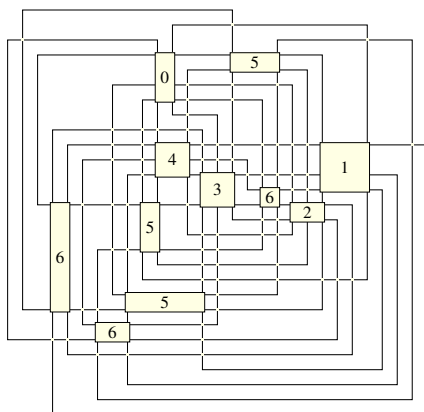
(6)  $30 \leq \nu(\mathcal{K}_{6 \times 1, 5}) \leq 57.$



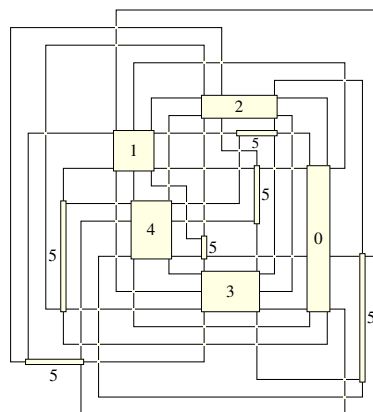
(7)  $30 \leq \nu(\mathcal{K}_{5 \times 1, 2, 2, 2}) \leq 73.$



(8)  $30 \leq \nu(\mathcal{K}_{5 \times 1, 2, 4}) \leq 62.$

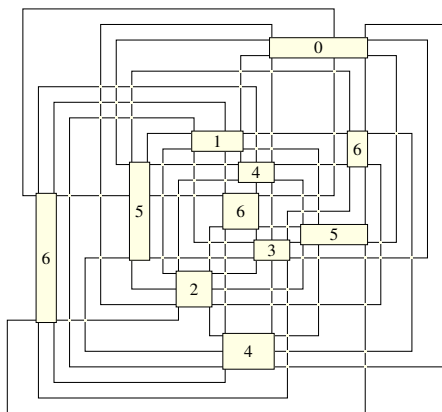


(9)  $30 \leq \nu(\mathcal{K}_{5 \times 1, 3, 3}) \leq 70.$

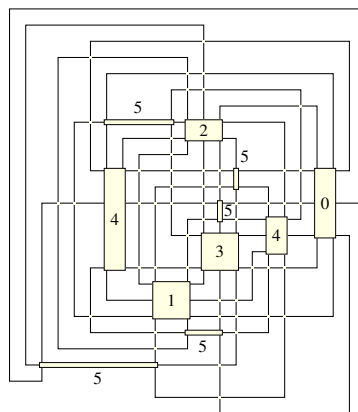


(10)  $30 \leq \nu(\mathcal{K}_{5 \times 1, 6}) \leq 40.$

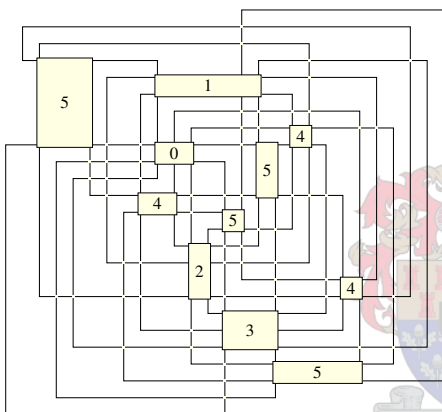
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



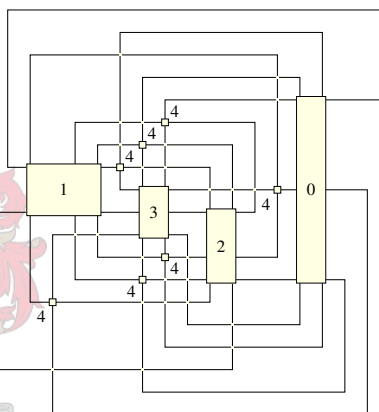
(11)  $30 \leq \nu(\mathcal{K}_{4 \times 1,2,2,3}) \leq 67$ .



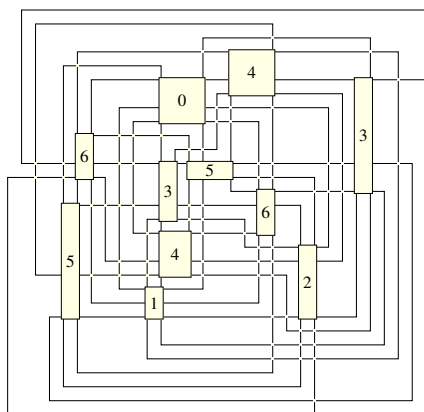
(12)  $30 \leq \nu(\mathcal{K}_{4 \times 1,2,2,5}) \leq 50$ .



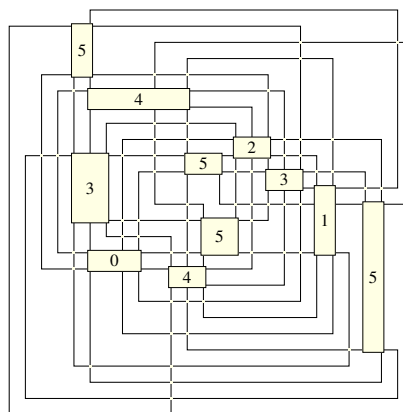
(13)  $30 \leq \nu(\mathcal{K}_{4 \times 1,3,3,4}) \leq 57$ .



(14)  $\nu(\mathcal{K}_{4 \times 1,7}) = 25$ .

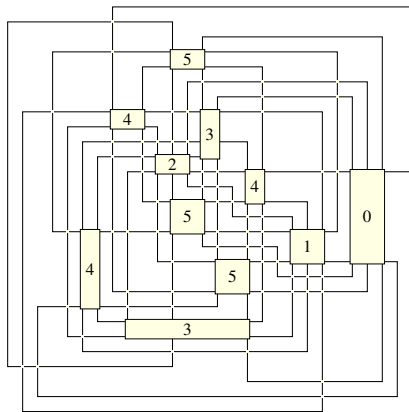


(15)  $30 \leq \nu(\mathcal{K}_{1,1,1,4 \times 2}) \leq 66$ .

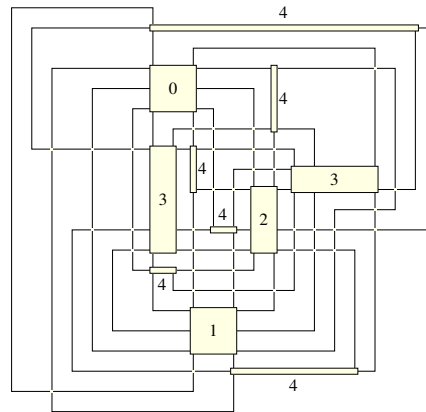


(16)  $30 \leq \nu(\mathcal{K}_{1,1,1,2,2,4}) \leq 56$ .

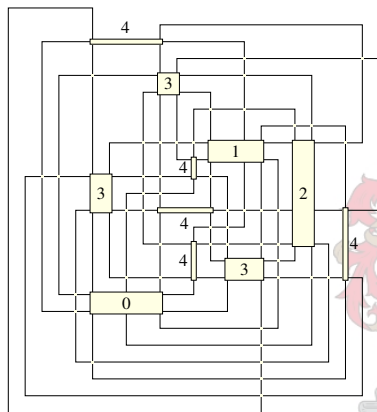
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



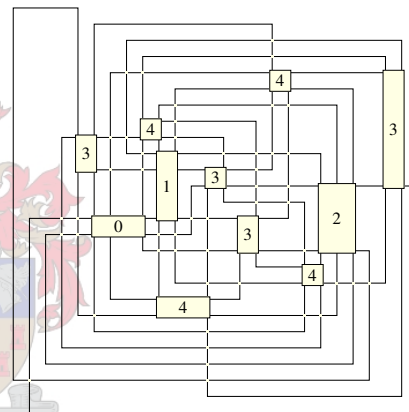
(17)  $30 \leq \nu(\mathcal{K}_{1,1,1,1,2,3,3}) \leq 62.$



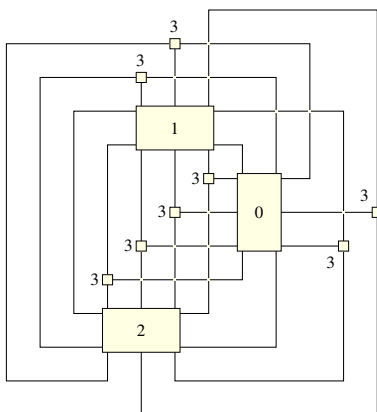
(18)  $30 \leq \nu(\mathcal{K}_{1,1,1,1,2,6}) \leq 36.$



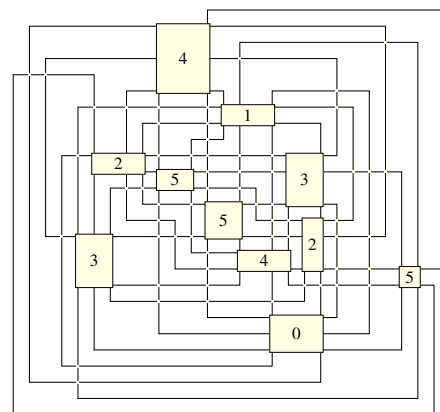
(19)  $30 \leq \nu(\mathcal{K}_{1,1,1,1,3,5}) \leq 47.$



(20)  $24 \leq \nu(\mathcal{K}_{1,1,1,1,4,4}) \leq 48.$

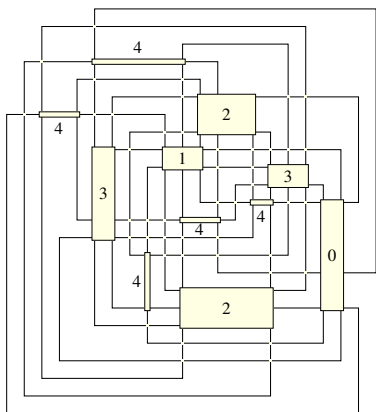


(21)  $\nu(\mathcal{K}_{1,1,1,8}) = 12.$

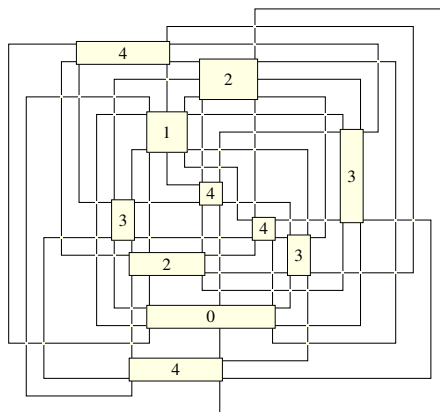


(22)  $30 \leq \nu(\mathcal{K}_{1,1,2,2,2,3}) \leq 60.$

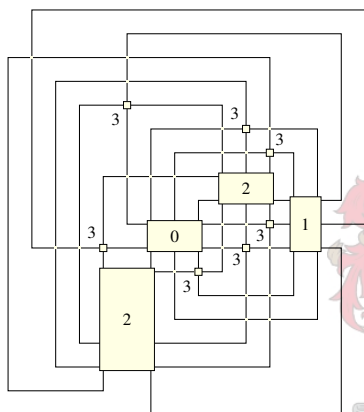
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



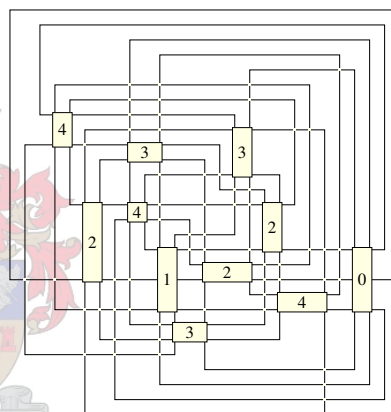
(23)  $30 \leq \nu(\mathcal{K}_{1,1,2,2,5}) \leq 44.$



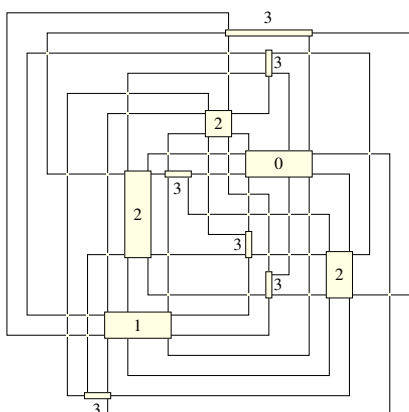
(24)  $30 \leq \nu(\mathcal{K}_{1,1,2,3,4}) \leq 51.$



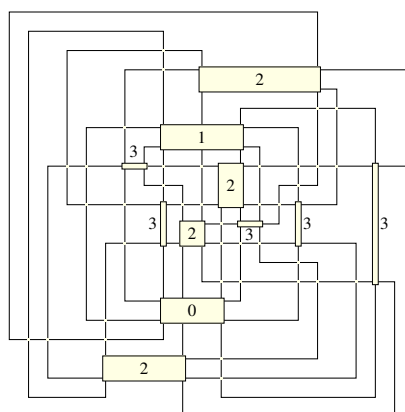
(25)  $\nu(\mathcal{K}_{1,1,2,7}) = 21.$



(26)  $30 \leq \nu(\mathcal{K}_{1,1,3,3,3}) \leq 58.$

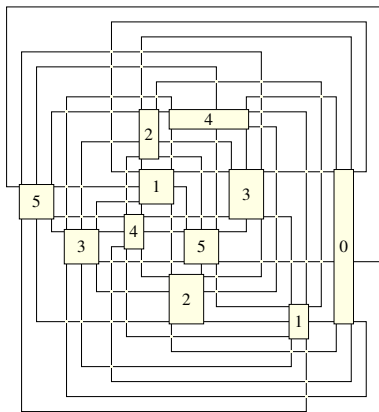


(27)  $30 \leq \nu(\mathcal{K}_{1,1,3,6}) \leq 33.$

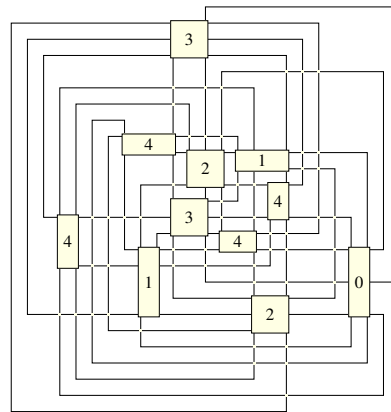


(28)  $24 \leq \nu(\mathcal{K}_{1,1,4,5}) \leq 38.$

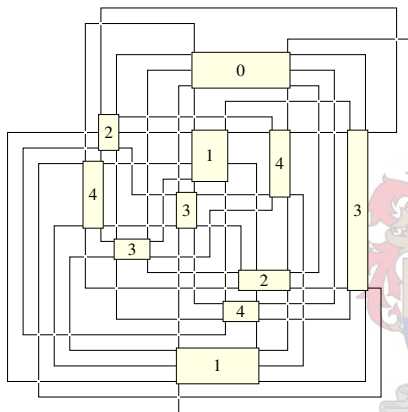
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



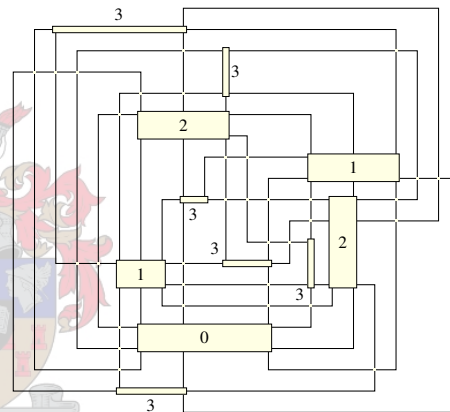
(29)  $30 \leq \nu(\mathcal{K}_{1,5 \times 2}) \leq 60.$



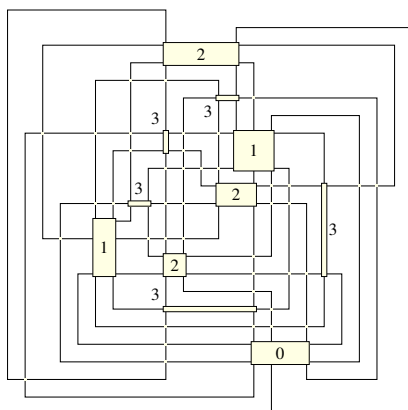
(30)  $30 \leq \nu(\mathcal{K}_{1,2,2,2,4}) \leq 51.$



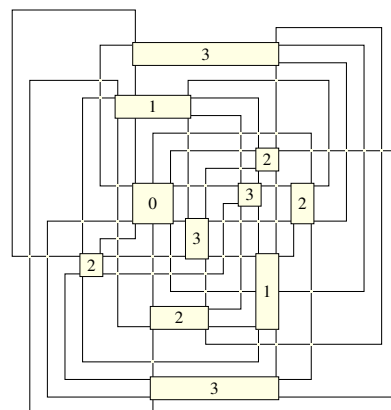
(31)  $30 \leq \nu(\mathcal{K}_{1,2,2,3,3}) \leq 55.$



(32)  $30 \leq \nu(\mathcal{K}_{1,2,2,6}) \leq 33.$

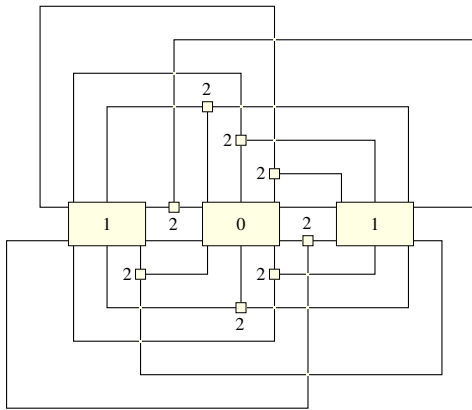


(33)  $30 \leq \nu(\mathcal{K}_{1,2,3,5}) \leq 41.$

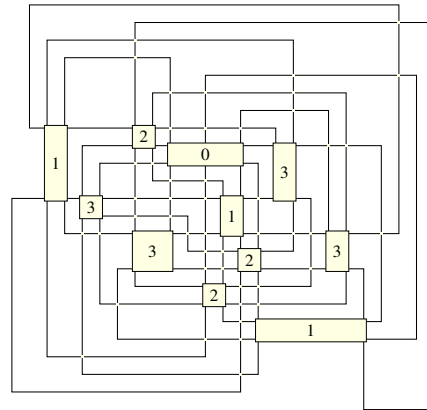


(34)  $24 \leq \nu(\mathcal{K}_{1,2,4,4}) \leq 44.$

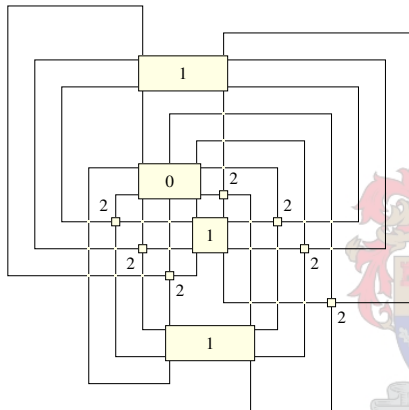
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



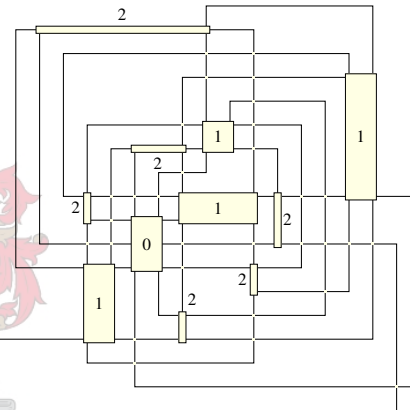
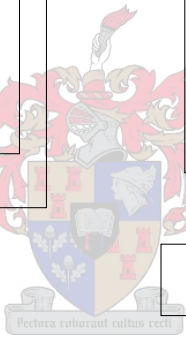
(35)  $\nu(\mathcal{K}_{1,2,8}) = 12$ .



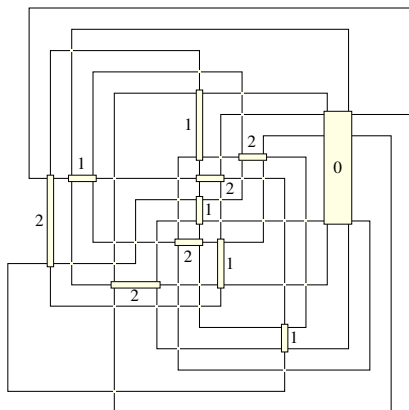
(36)  $24 \leq \nu(\mathcal{K}_{1,3,3,4}) \leq 47$ .



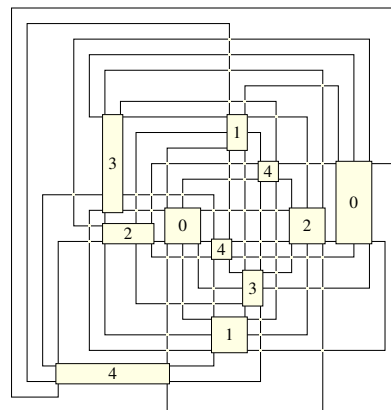
(37)  $\nu(\mathcal{K}_{1,3,7}) = 21$ .



(38)  $24 \leq \nu(\mathcal{K}_{1,4,6}) \leq 30$ .

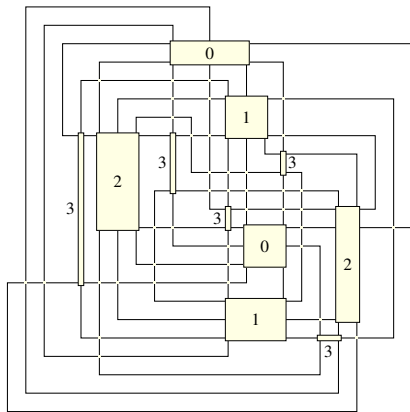


(39)  $24 \leq \nu(\mathcal{K}_{1,5,5}) \leq 32$ .

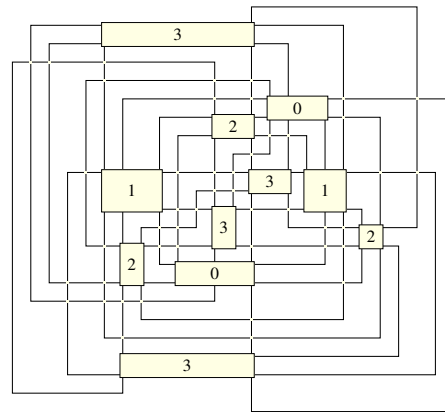


(40)  $30 \leq \nu(\mathcal{K}_{4 \times 2, 3}) \leq 54$ .

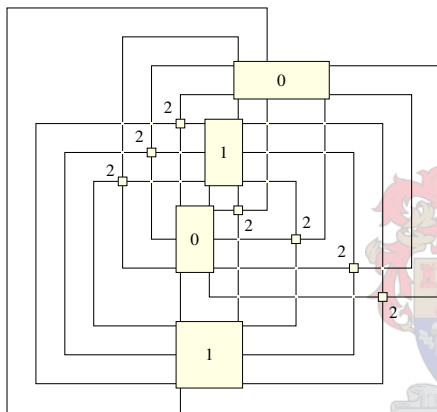
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



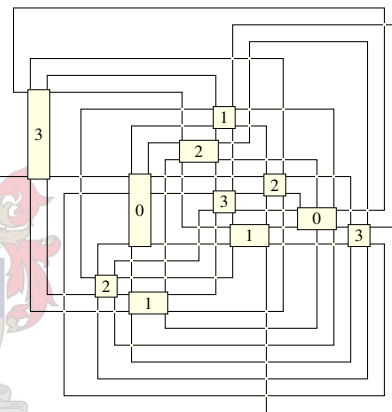
(41)  $24 \leq \nu(\mathcal{K}_{2,2,2,5}) \leq 39.$



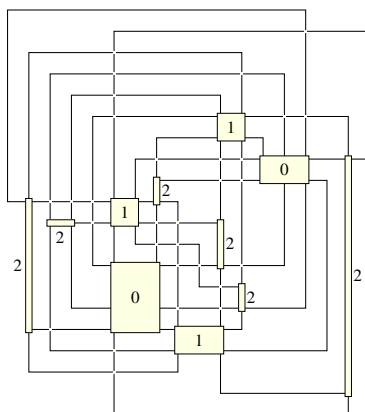
(42)  $30 \leq \nu(\mathcal{K}_{2,2,3,4}) \leq 46.$



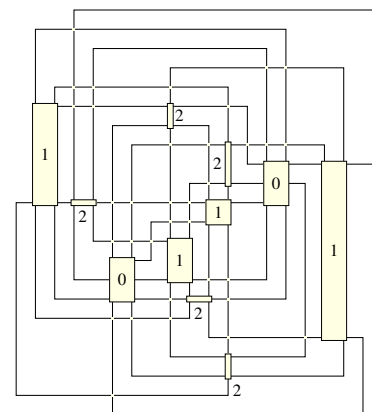
(43)  $\nu(\mathcal{K}_{2,2,2,7}) = 18.$



(44)  $30 \leq \nu(\mathcal{K}_{2,3,3,3}) \leq 51.$

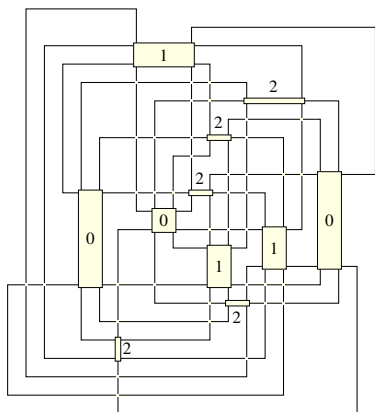


(45)  $\nu(\mathcal{K}_{2,3,3,6}) = 30.$

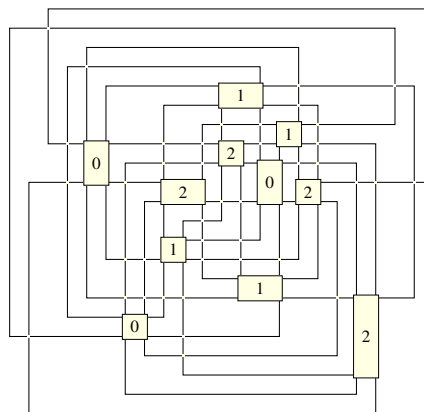


(46)  $24 \leq \nu(\mathcal{K}_{2,4,4,5}) \leq 34.$

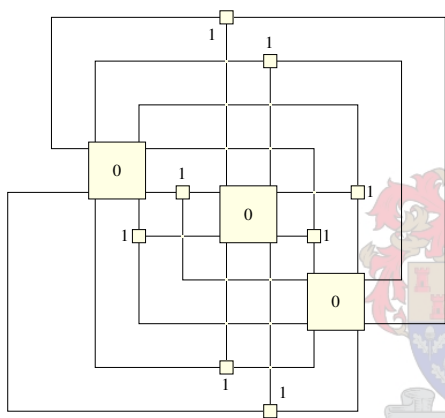
Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.



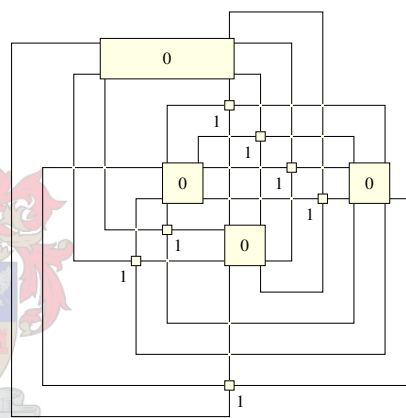
(47)  $24 \leq \nu(\mathcal{K}_{3,3,5}) \leq 39.$



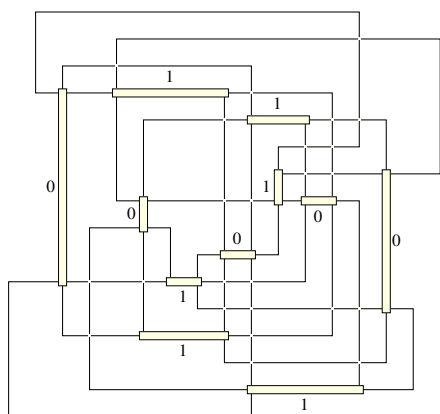
(48)  $18 \leq \nu(\mathcal{K}_{3,4,4}) \leq 40.$



(49)  $\nu(\mathcal{K}_{3,8}) = 12.$



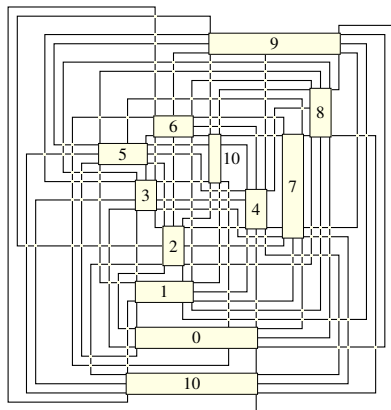
(50)  $\nu(\mathcal{K}_{4,7}) = 18.$



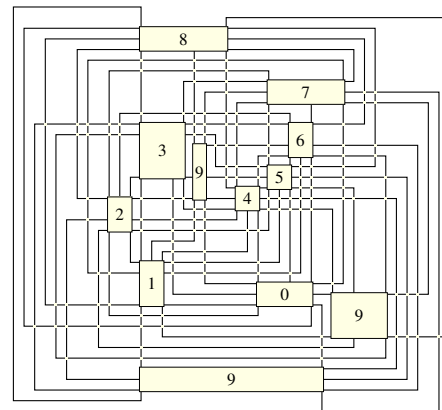
(51)  $\nu(\mathcal{K}_{5,6}) = 24.$

Figure 7.14 (continued): Non-planar complete multipartite graphs of order 11.

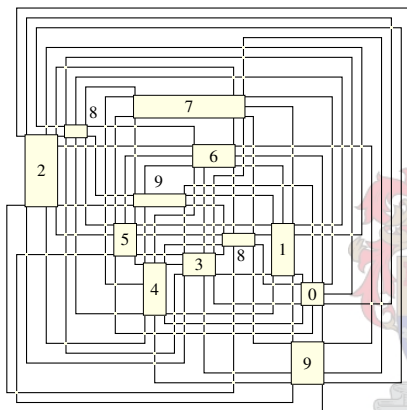




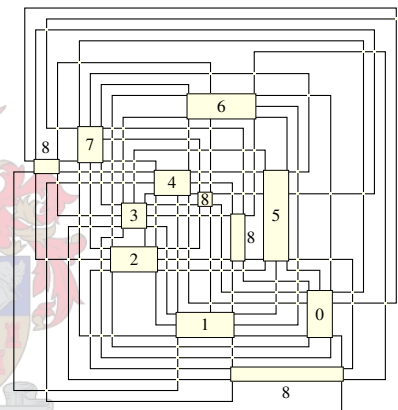
$$(1) 79 \leq \nu(\mathcal{K}_{10 \times 1, 2}) \leq 140.$$



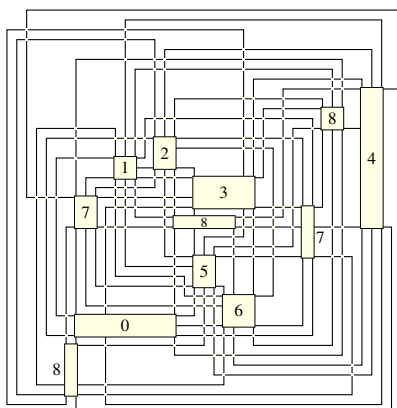
$$(2) 79 \leq \nu(\mathcal{K}_{9 \times 1, 3}) \leq 130.$$



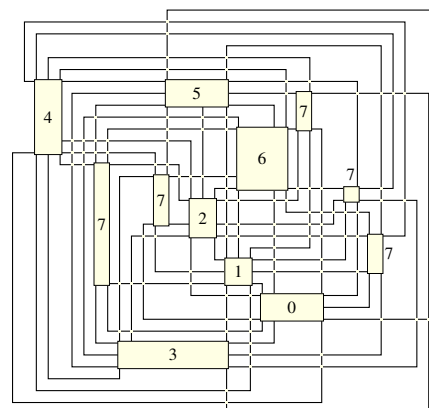
$$(3) 79 \leq \nu(\mathcal{K}_{8 \times 1, 2, 2}) \leq 130.$$



$$(4) 43 \leq \nu(\mathcal{K}_{8 \times 1, 4}) \leq 114.$$

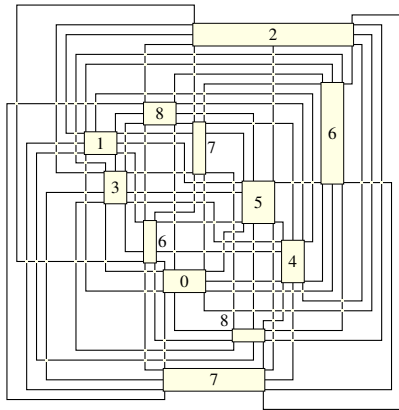


$$(5) 43 \leq \nu(\mathcal{K}_{7 \times 1, 2, 3}) \leq 121.$$

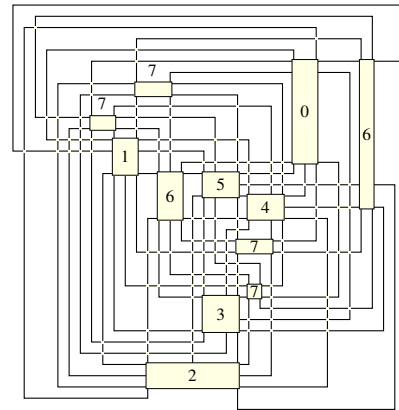


$$(6) 43 \leq \nu(\mathcal{K}_{7 \times 1, 5}) \leq 96.$$

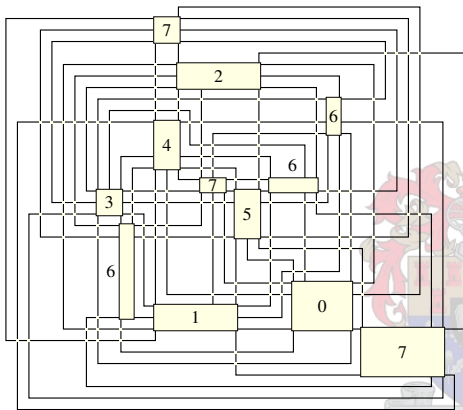
Figure 7.15: Non-planar complete multipartite graphs of order 12.



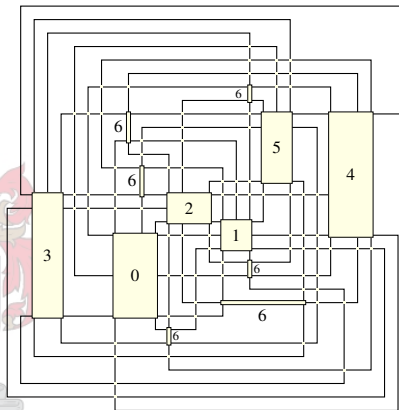
(7)  $43 \leq \nu(\mathcal{K}_{6 \times 1,2,2,2,2,2}) \leq 120$ .



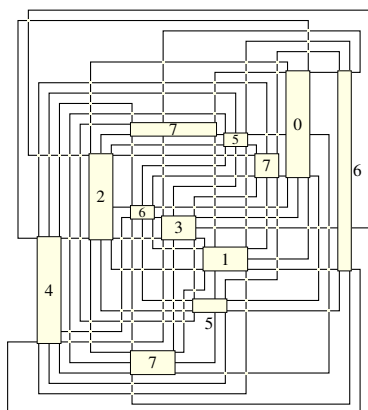
(8)  $43 \leq \nu(\mathcal{K}_{6 \times 1,2,2,4}) \leq 105$ .



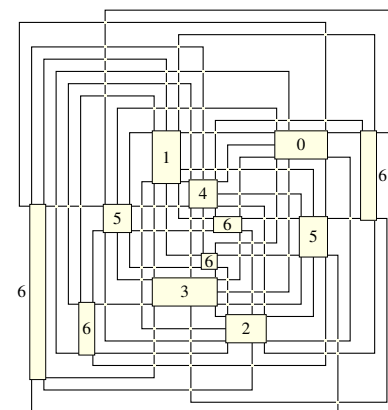
(9)  $43 \leq \nu(\mathcal{K}_{6 \times 1,3,3,3}) \leq 112$ .



(10)  $43 \leq \nu(\mathcal{K}_{6 \times 1,6}) \leq 75$ .

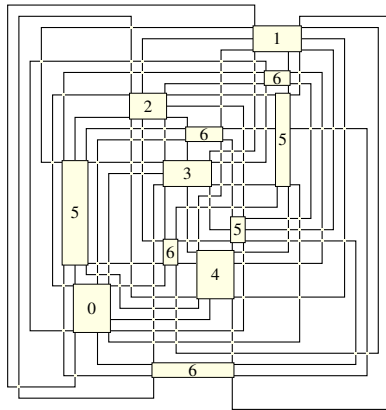


(11)  $43 \leq \nu(\mathcal{K}_{5 \times 1,2,2,3,3}) \leq 112$ .

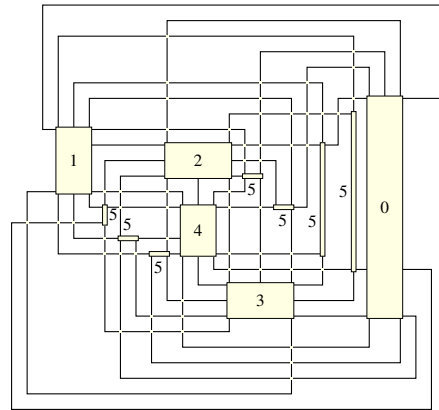


(12)  $43 \leq \nu(\mathcal{K}_{5 \times 1,2,5}) \leq 89$ .

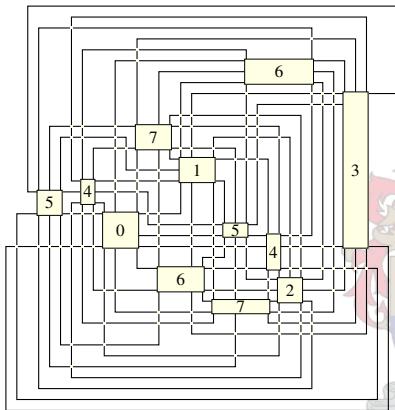
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



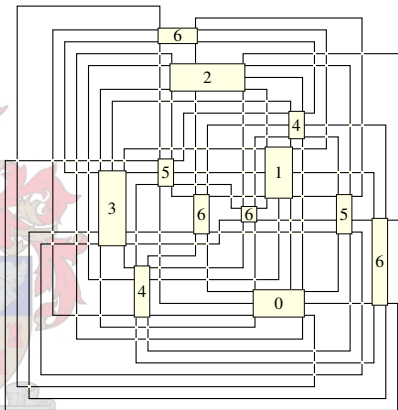
$$(13) \quad 43 \leq \nu(\mathcal{K}_{5 \times 1, 3, 4}) \leq 98.$$



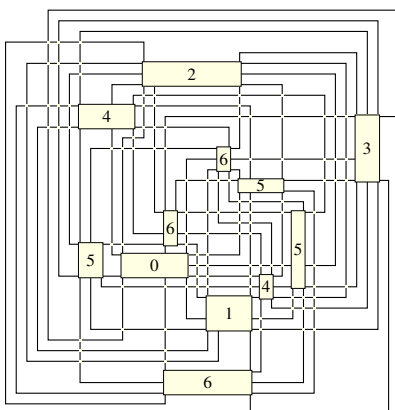
$$(14) \quad 43 \leq \nu(\mathcal{K}_{5 \times 1, 7}) \leq 54.$$



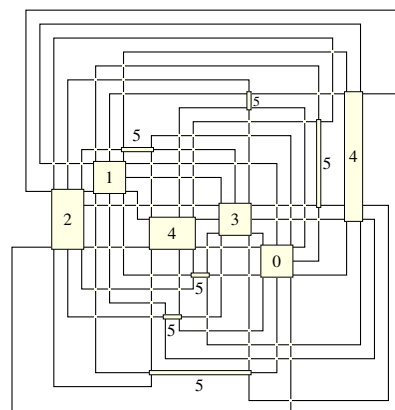
$$(15) \quad 43 \leq \nu(\mathcal{K}_{4 \times 1, 4 \times 2}) \leq 110.$$



$$(16) \quad 43 \leq \nu(\mathcal{K}_{4 \times 1, 2, 2, 4}) \leq 96.$$

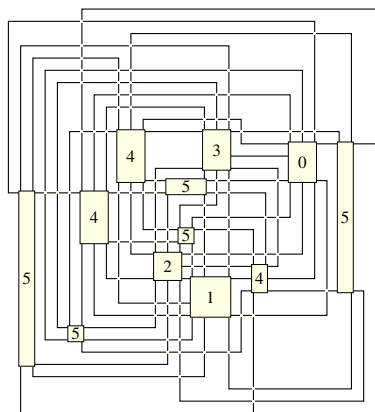


$$(17) \quad 43 \leq \nu(\mathcal{K}_{4 \times 1, 2, 3, 3}) \leq 104.$$

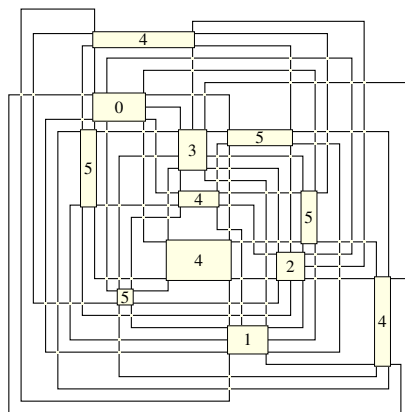


$$(18) \quad 43 \leq \nu(\mathcal{K}_{4 \times 1, 2, 6}) \leq 68.$$

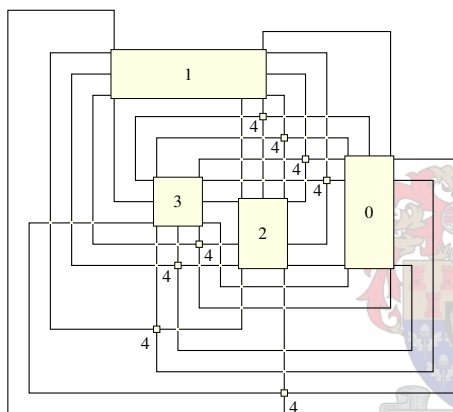
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



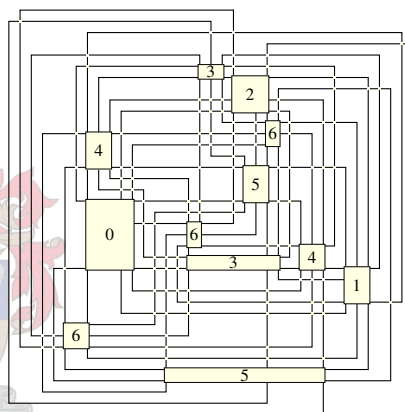
(19)  $43 \leq \nu(\mathcal{K}_{4 \times 1, 3, 5}) \leq 82.$



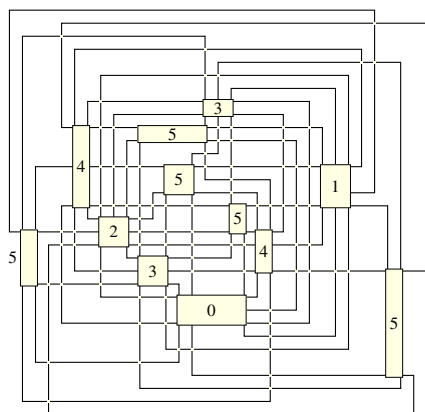
(20)  $36 \leq \nu(\mathcal{K}_{4 \times 1, 4, 4}) \leq 84.$



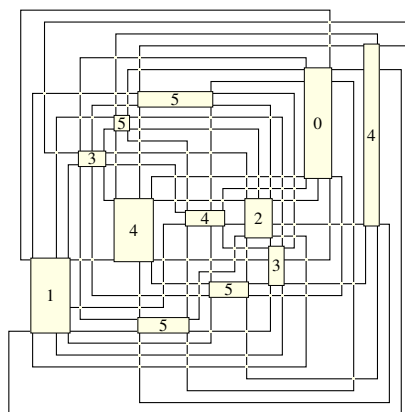
(21)  $\nu(\mathcal{K}_{4 \times 1, 8}) = 32.$



(22)  $43 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 2, 2, 3}) \leq 103.$

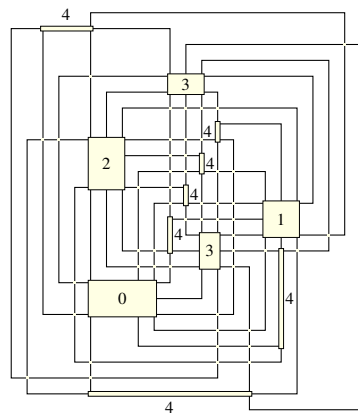


(23)  $43 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 2, 5}) \leq 82.$

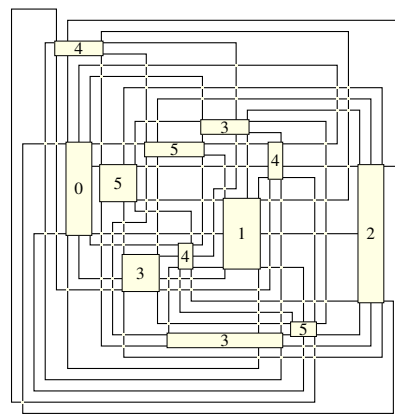


(24)  $43 \leq \nu(\mathcal{K}_{1, 1, 1, 2, 3, 4}) \leq 90.$

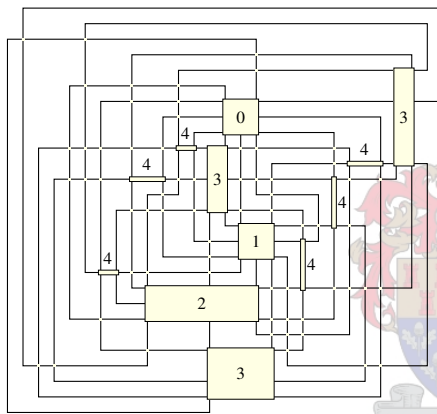
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



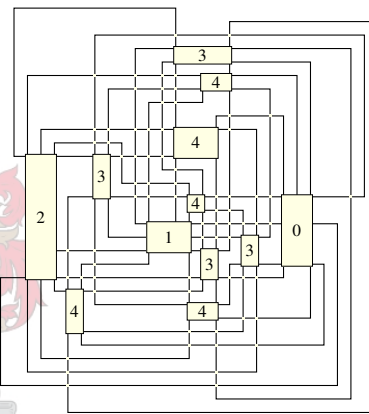
(25)  $43 \leq \nu(\mathcal{K}_{1,1,1,2,7}) \leq 50.$



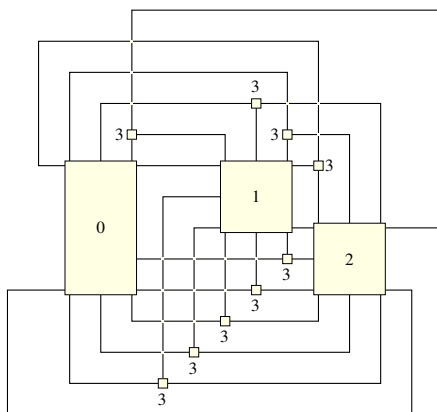
(26)  $43 \leq \nu(\mathcal{K}_{1,1,1,3,3,3}) \leq 96.$



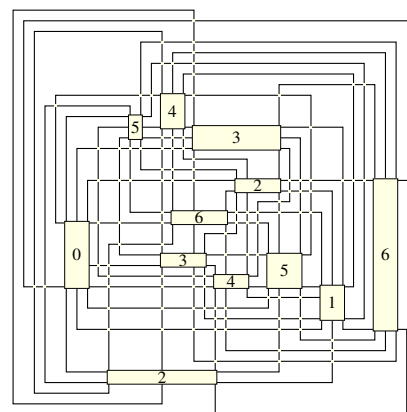
(27)  $43 \leq \nu(\mathcal{K}_{1,1,1,3,6}) \leq 64.$



(28)  $36 \leq \nu(\mathcal{K}_{1,1,1,4,5}) \leq 72.$

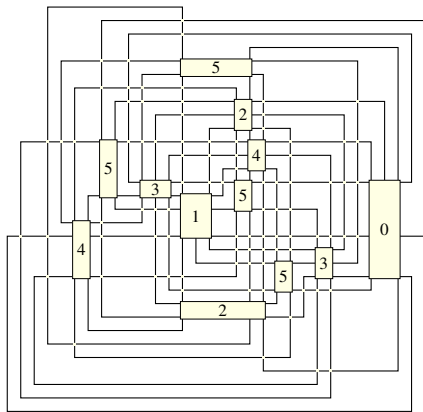


(29)  $\nu(\mathcal{K}_{1,1,1,9}) = 16.$

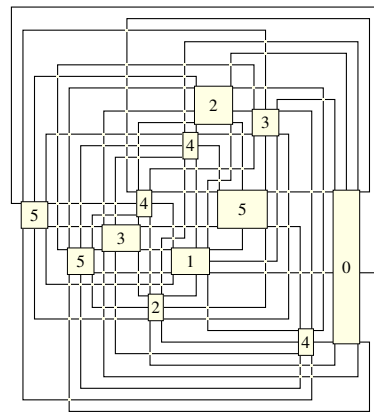


(30)  $43 \leq \nu(\mathcal{K}_{1,1,5 \times 2}) \leq 103.$

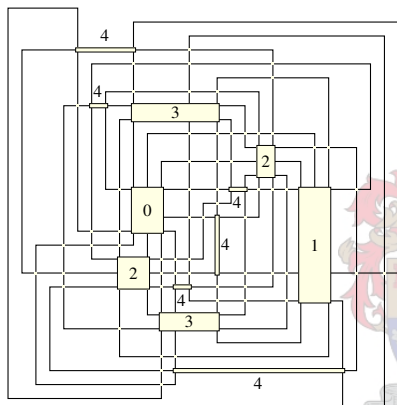
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



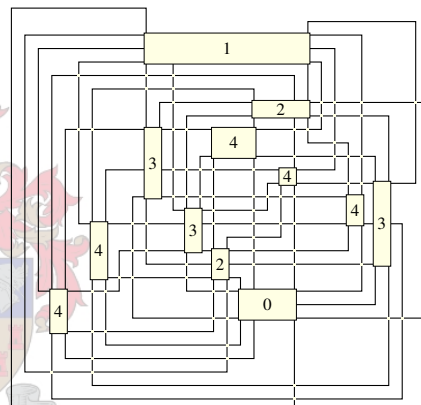
(31)  $43 \leq \nu(\mathcal{K}_{1,1,2,2,2,4}) \leq 87.$



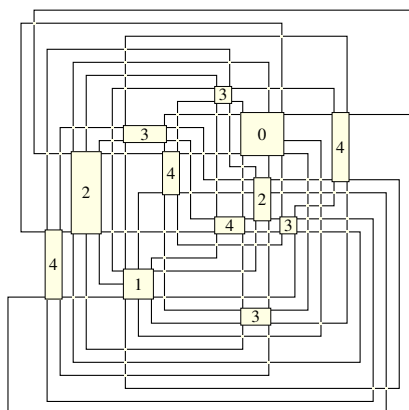
(32)  $43 \leq \nu(\mathcal{K}_{1,1,2,2,3,3}) \leq 96.$



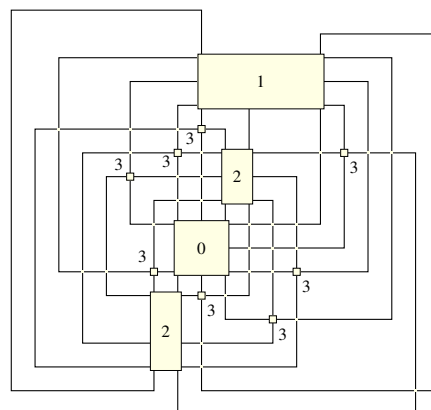
(33)  $43 \leq \nu(\mathcal{K}_{1,1,2,2,6}) \leq 61.$



(34)  $43 \leq \nu(\mathcal{K}_{1,1,2,3,5}) \leq 76.$

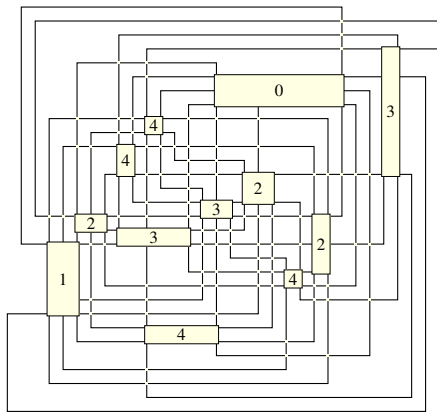


(35)  $36 \leq \nu(\mathcal{K}_{1,1,2,4,4}) \leq 76.$

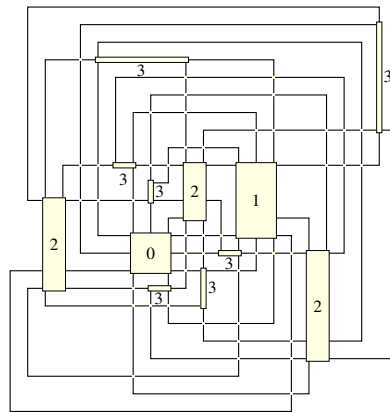


(36)  $\nu(\mathcal{K}_{1,1,2,8}) = 28.$

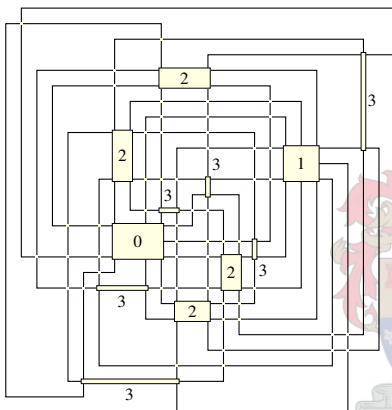
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



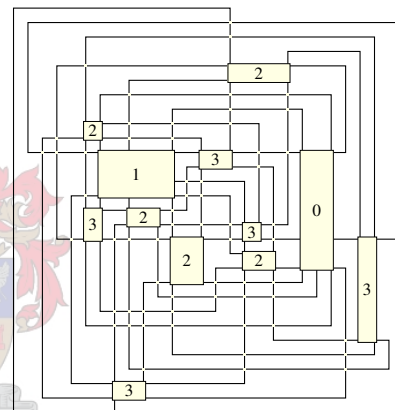
(37)  $43 \leq \nu(\mathcal{K}_{1,1,3,3,4}) \leq 84.$



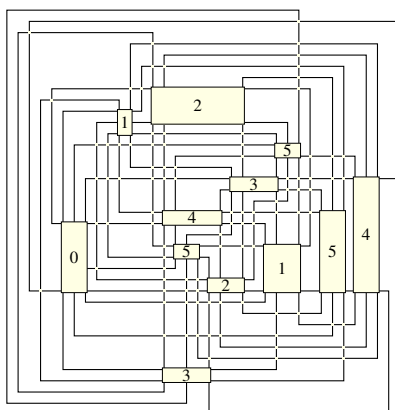
(38)  $43 \leq \nu(\mathcal{K}_{1,1,3,7}) \leq 46.$



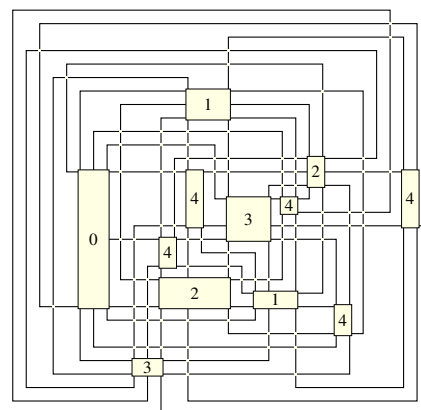
(39)  $36 \leq \nu(\mathcal{K}_{1,1,4,6}) \leq 54.$



(40)  $36 \leq \nu(\mathcal{K}_{1,1,5,5}) \leq 60.$

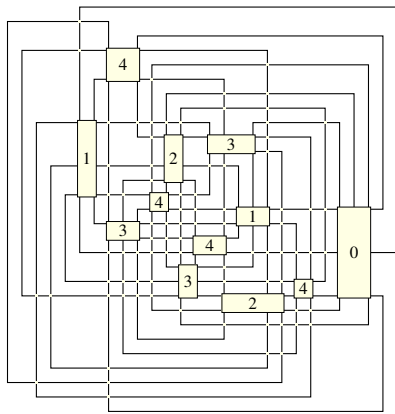


(41)  $43 \leq \nu(\mathcal{K}_{1,4 \times 2,3}) \leq 94.$

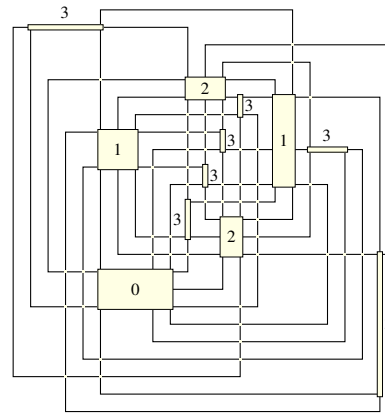


(42)  $43 \leq \nu(\mathcal{K}_{1,2,2,2,5}) \leq 75.$

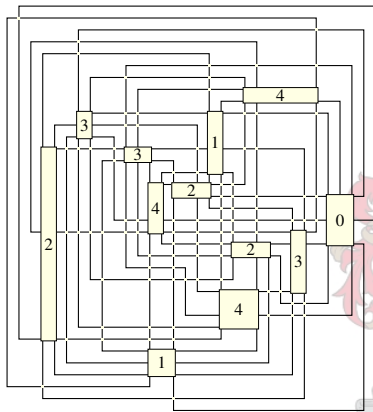
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



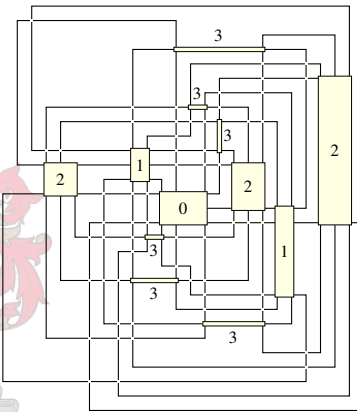
(43)  $43 \leq \nu(\mathcal{K}_{1,2,2,2,3,4}) \leq 82.$



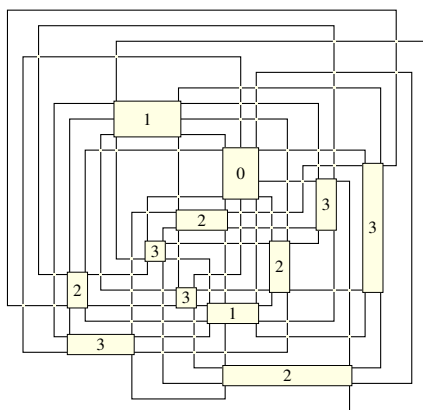
(44)  $43 \leq \nu(\mathcal{K}_{1,2,2,2,7}) \leq 46.$



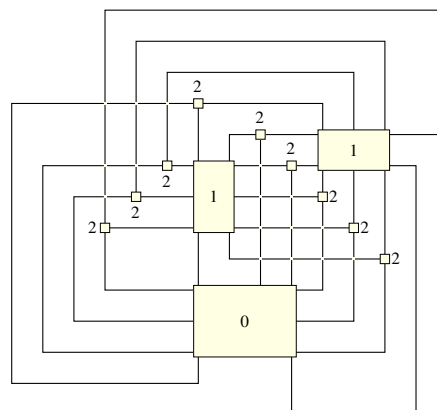
(45)  $43 \leq \nu(\mathcal{K}_{1,2,3,3,3,3}) \leq 89.$



(46)  $43 \leq \nu(\mathcal{K}_{1,2,3,3,6}) \leq 58.$



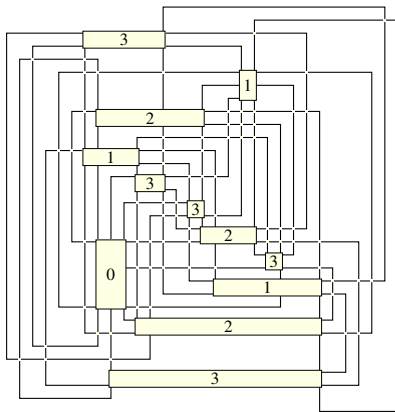
(47)  $36 \leq \nu(\mathcal{K}_{1,2,4,5}) \leq 66.$



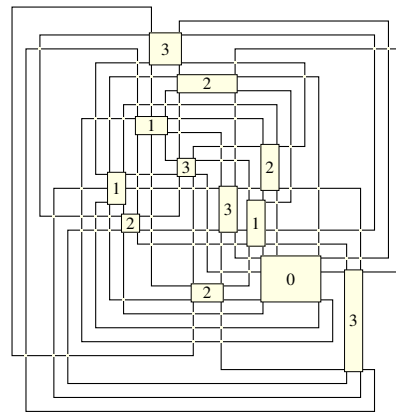
(48)  $\nu(\mathcal{K}_{1,2,9}) = 16.$

Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.

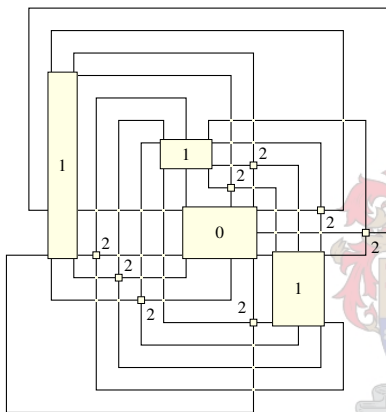




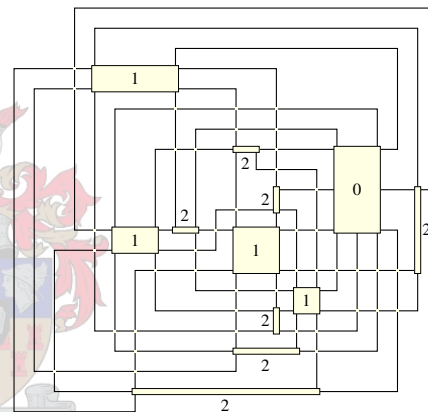
(49)  $36 \leq \nu(\mathcal{K}_{1,3,3,5}) \leq 70.$



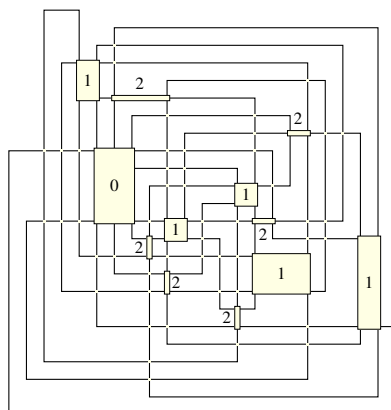
(50)  $36 \leq \nu(\mathcal{K}_{1,3,4,4}) \leq 72.$



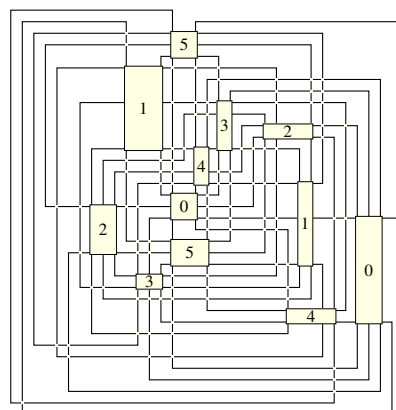
(51)  $\nu(\mathcal{K}_{1,3,8}) = 28.$



(52)  $36 \leq \nu(\mathcal{K}_{1,4,7}) \leq 42.$

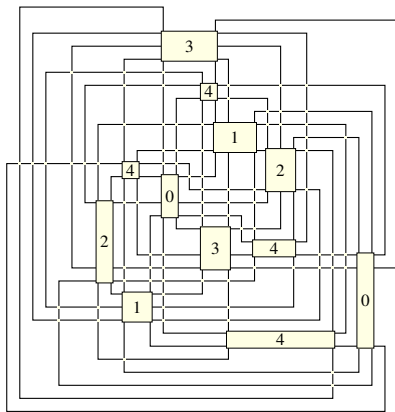


(53)  $36 \leq \nu(\mathcal{K}_{1,5,6}) \leq 48.$

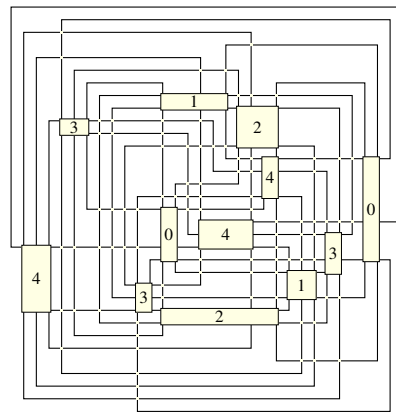


(54)  $36 \leq \nu(\mathcal{K}_{6 \times 2}) \leq 94.$

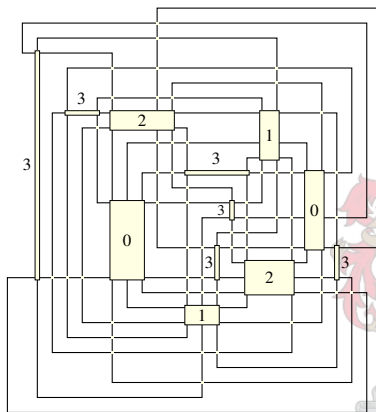
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



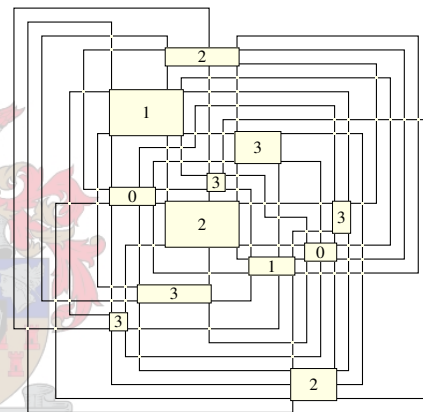
(55)  $36 \leq \nu(\mathcal{K}_{4 \times 2, 4}) \leq 78.$



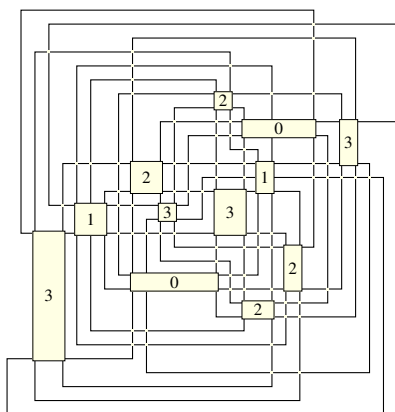
(56)  $43 \leq \nu(\mathcal{K}_{2, 2, 2, 3, 3}) \leq 88.$



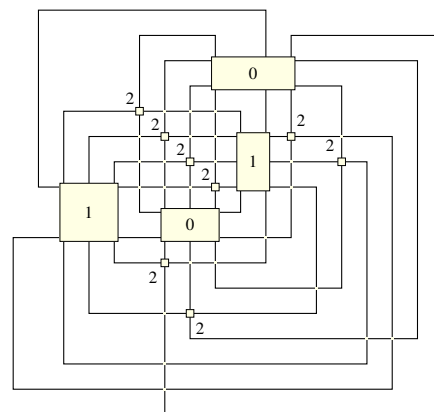
(57)  $36 \leq \nu(\mathcal{K}_{2, 2, 2, 2, 6}) \leq 54.$



(58)  $43 \leq \nu(\mathcal{K}_{2, 2, 3, 3, 5}) \leq 70.$

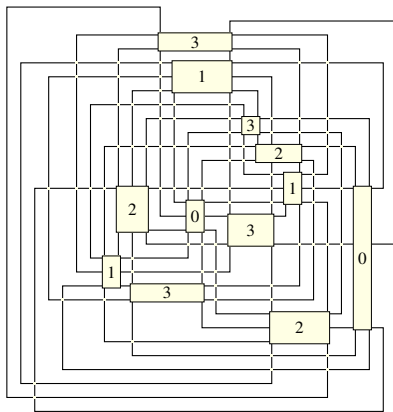


(59)  $36 \leq \nu(\mathcal{K}_{2, 2, 4, 4, 4}) \leq 68.$

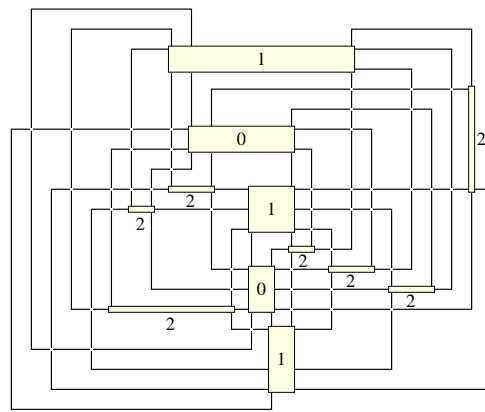


(60)  $\nu(\mathcal{K}_{2, 2, 8}) = 24.$

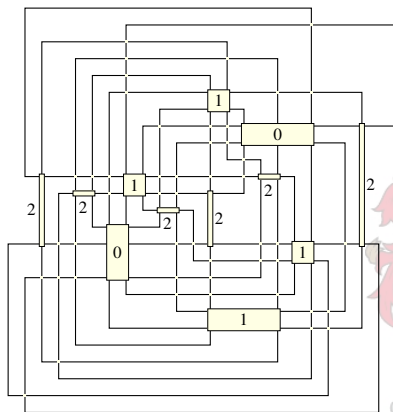
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



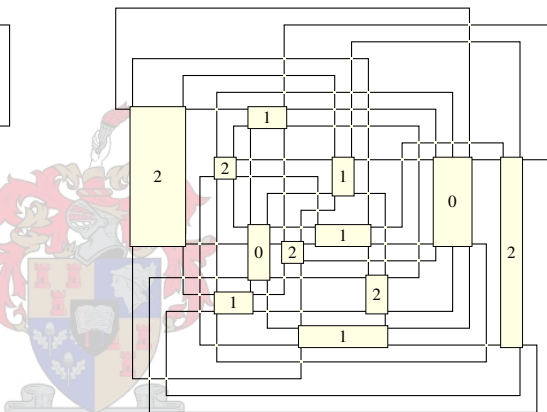
(61)  $43 \leq \nu(\mathcal{K}_{2,3,3,4}) \leq 77.$



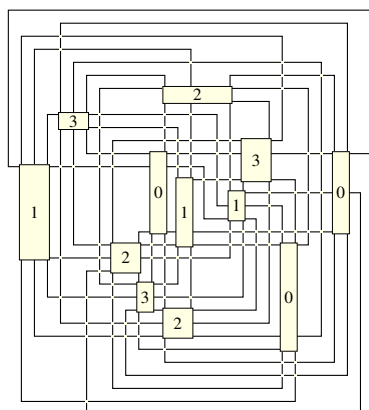
(62)  $\nu(\mathcal{K}_{2,3,7}) = 43.$



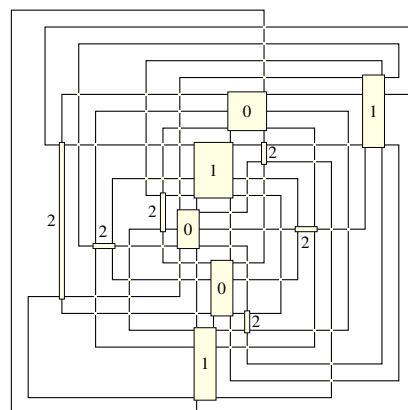
(63)  $36 \leq \nu(\mathcal{K}_{2,4,6}) \leq 48.$



(64)  $36 \leq \nu(\mathcal{K}_{2,5,5}) \leq 56.$

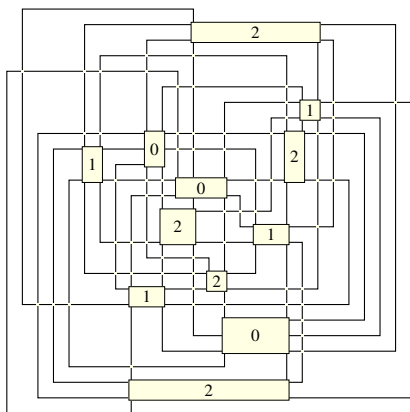


(65)  $36 \leq \nu(\mathcal{K}_{4 \times 3}) \leq 82.$

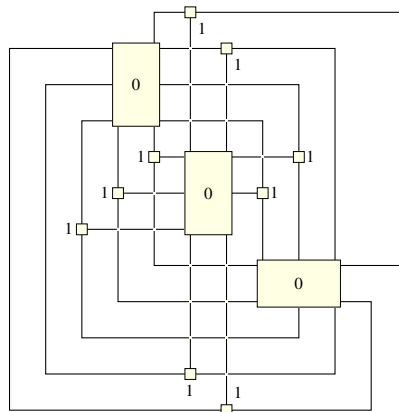


(66)  $36 \leq \nu(\mathcal{K}_{3,3,6}) \leq 55.$

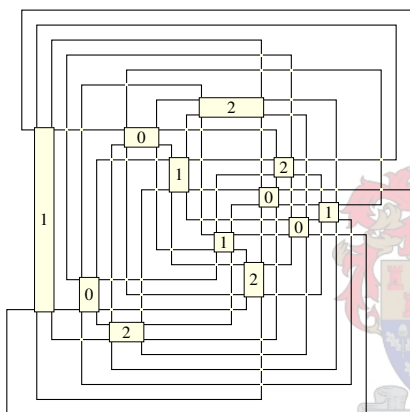
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



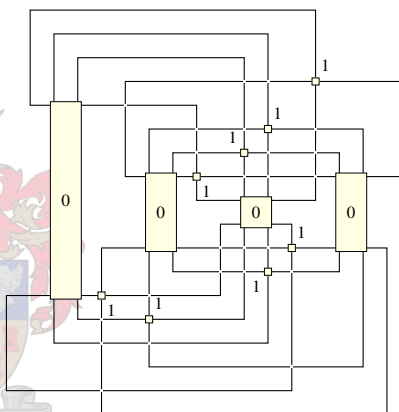
(67)  $36 \leq \nu(\mathcal{K}_{3,4,5}) \leq 62.$



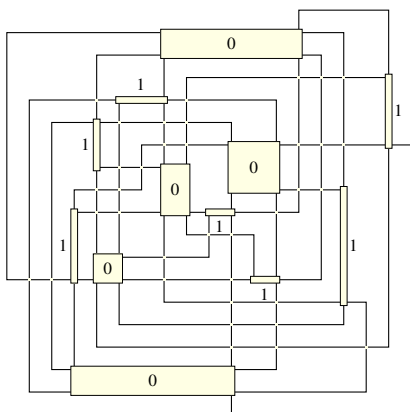
(68)  $\nu(\mathcal{K}_{3,9}) = 16.$



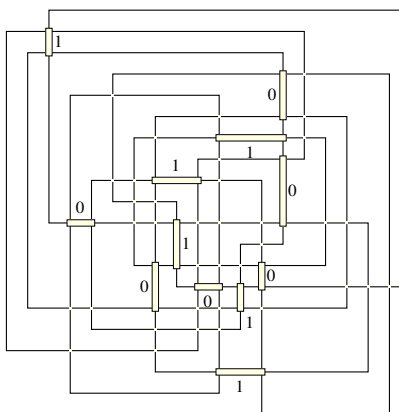
(69)  $24 \leq \nu(\mathcal{K}_{4,4,4}) \leq 60.$



(70)  $\nu(\mathcal{K}_{4,8}) = 24.$

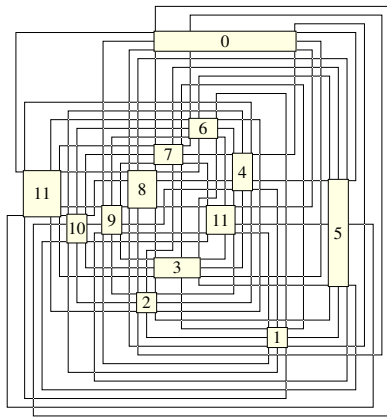


(71)  $\nu(\mathcal{K}_{5,7}) = 36.$

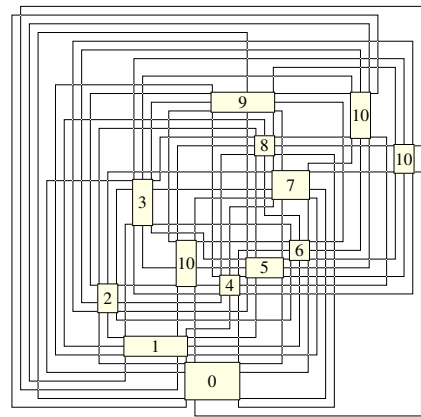


(72)  $\nu(\mathcal{K}_{6,6}) = 36.$

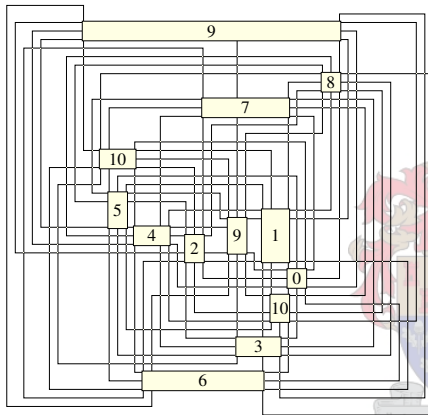
Figure 7.15 (continued): Non-planar complete multipartite graphs of order 12.



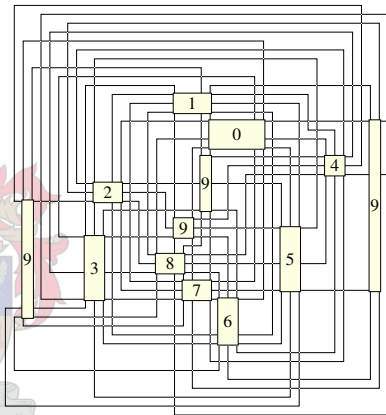
$$(1) 118 \leq \nu(\mathcal{K}_{11 \times 1, 2}) \leq 210.$$



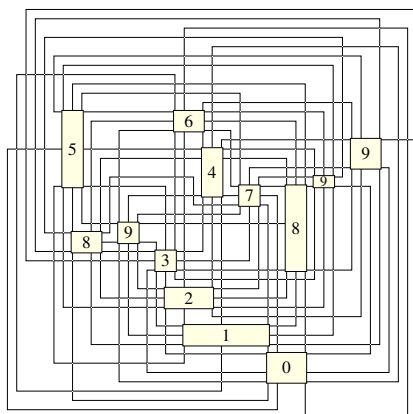
$$(2) 79 \leq \nu(\mathcal{K}_{10 \times 1, 3}) \leq 200.$$



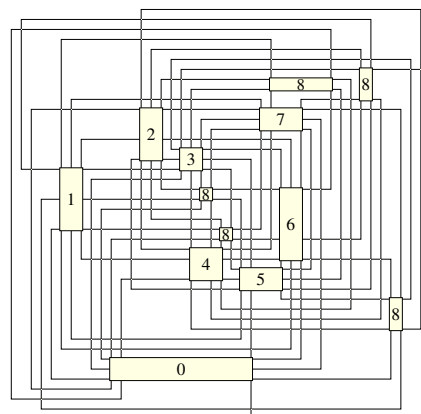
$$(3) 79 \leq \nu(\mathcal{K}_{9 \times 1, 2, 2}) \leq 196.$$



$$(4) 79 \leq \nu(\mathcal{K}_{9 \times 1, 4}) \leq 176.$$

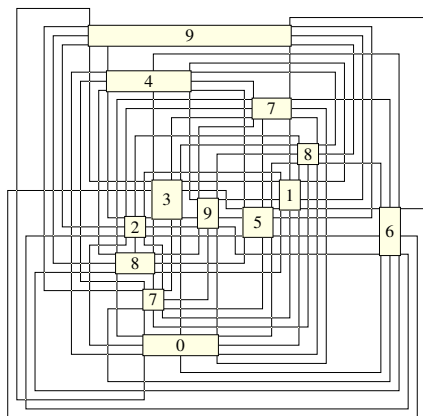


$$(5) 79 \leq \nu(\mathcal{K}_{8 \times 1, 2, 3}) \leq 186.$$

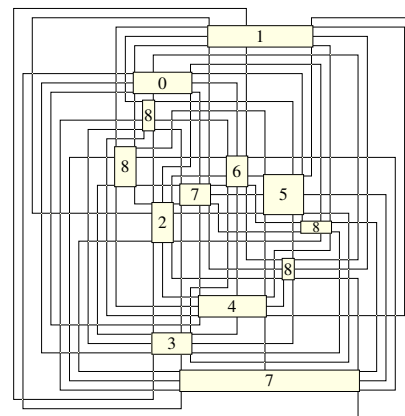


$$(6) 56 \leq \nu(\mathcal{K}_{8 \times 1, 5}) \leq 156.$$

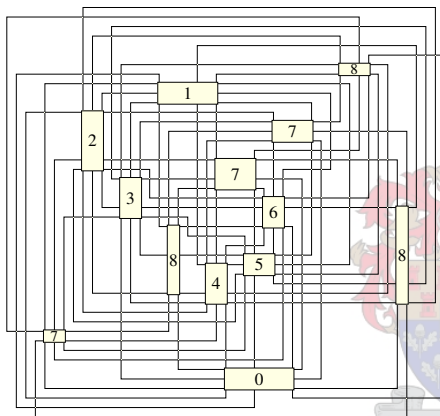
Figure 7.16: Non-planar complete multipartite graphs of order 13.



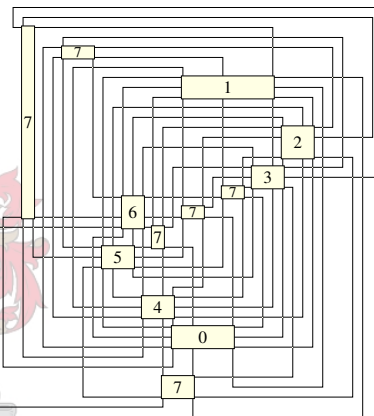
(7)  $79 \leq \nu(\mathcal{K}_{7 \times 1, 2, 2, 2}) \leq 183.$



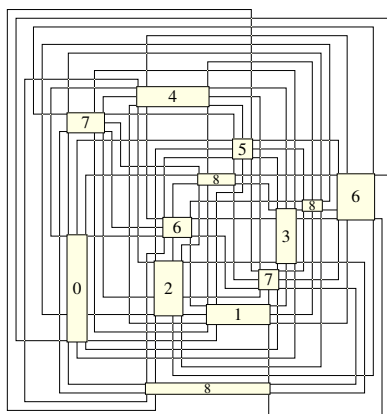
(8)  $56 \leq \nu(\mathcal{K}_{7 \times 1, 2, 4}) \leq 164.$



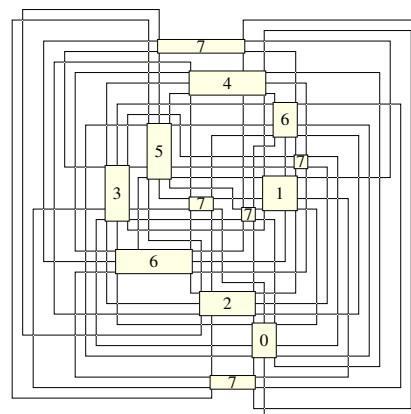
(9)  $56 \leq \nu(\mathcal{K}_{7 \times 1, 3, 3}) \leq 177.$



(10)  $56 \leq \nu(\mathcal{K}_{7 \times 1, 6}) \leq 126.$

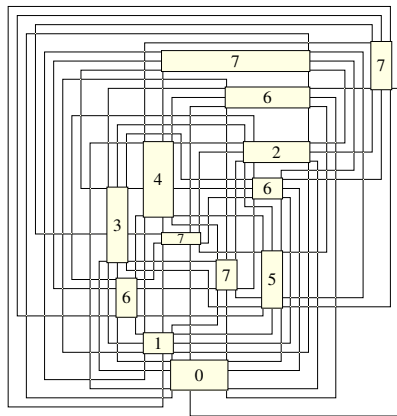


(11)  $56 \leq \nu(\mathcal{K}_{6 \times 1, 2, 2, 3}) \leq 173.$

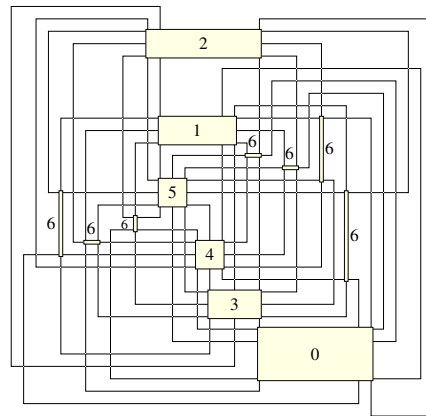


(12)  $56 \leq \nu(\mathcal{K}_{6 \times 1, 2, 5}) \leq 144.$

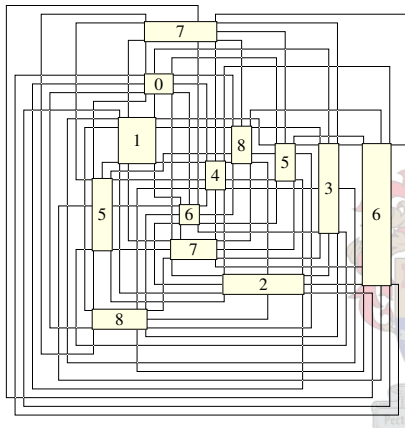
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



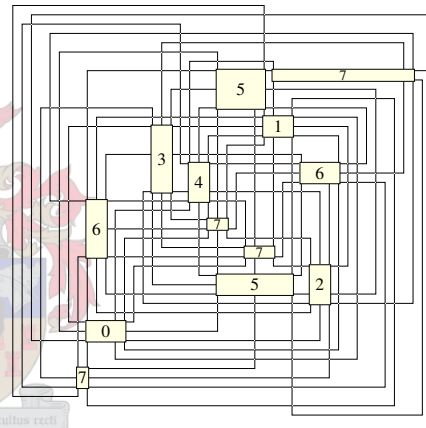
$$(13) \quad 56 \leq \nu(\mathcal{K}_{6 \times 1, 3, 4}) \leq 155.$$



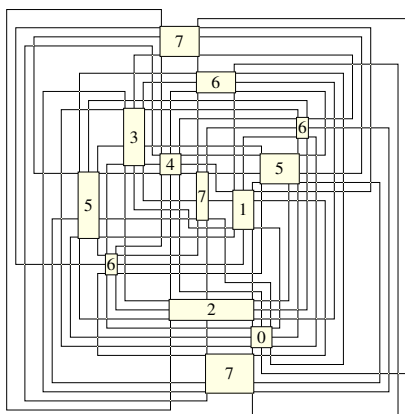
$$(14) \quad 56 \leq \nu(\mathcal{K}_{6 \times 1, 7}) \leq 99.$$



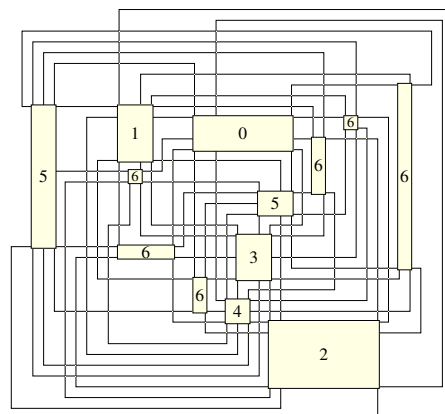
$$(15) \quad 56 \leq \nu(\mathcal{K}_{5 \times 1, 4 \times 2}) \leq 171.$$



$$(16) \quad 56 \leq \nu(\mathcal{K}_{5 \times 1, 2, 2, 4}) \leq 153.$$

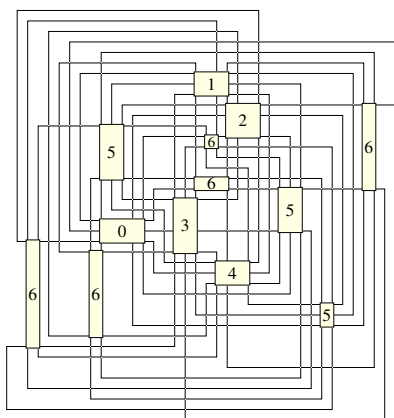


$$(17) \quad 56 \leq \nu(\mathcal{K}_{5 \times 1, 2, 3, 3}) \leq 164.$$

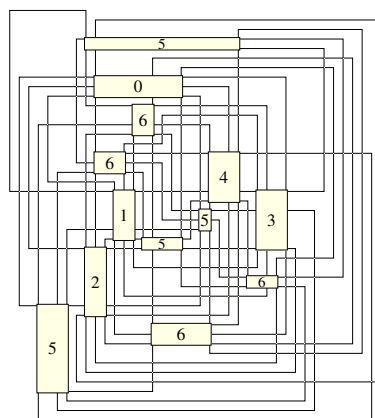


$$(18) \quad 56 \leq \nu(\mathcal{K}_{5 \times 1, 2, 6}) \leq 117.$$

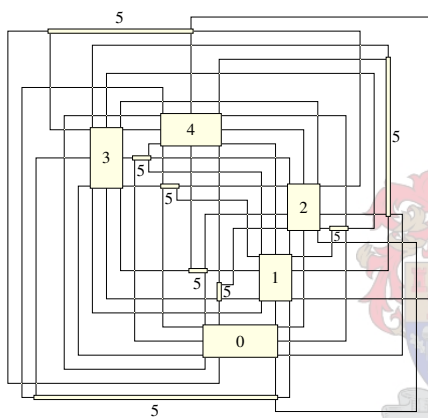
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



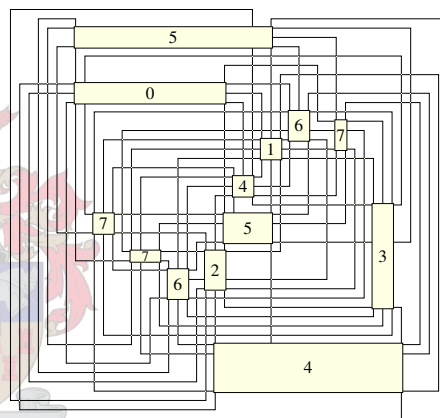
(19)  $56 \leq \nu(\mathcal{K}_{5 \times 1, 3, 5}) \leq 137$ .



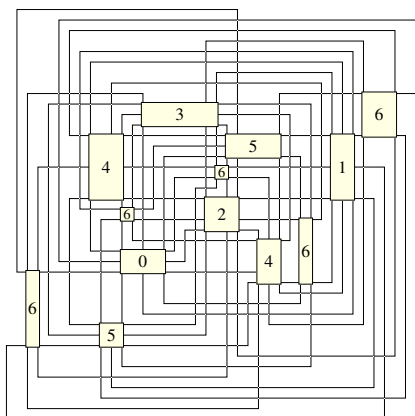
(20)  $56 \leq \nu(\mathcal{K}_{5 \times 1, 4, 4}) \leq 137$ .



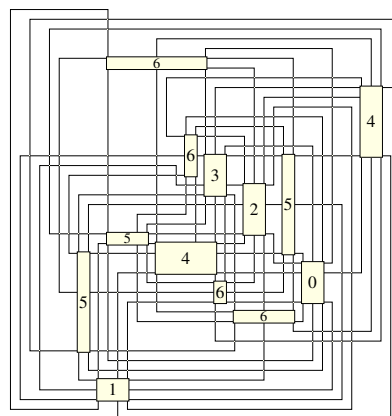
(21)  $56 \leq \nu(\mathcal{K}_{5 \times 1, 8}) \leq 69$ .



(22)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 2, 2, 2, 3}) \leq 161$ .



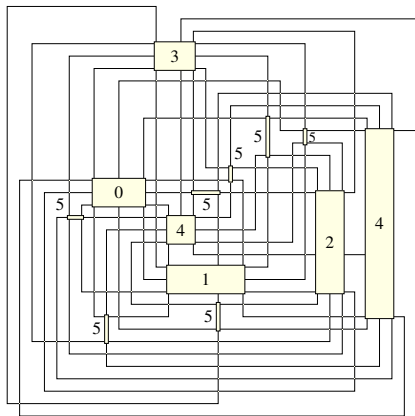
(23)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 2, 2, 5}) \leq 133$ .



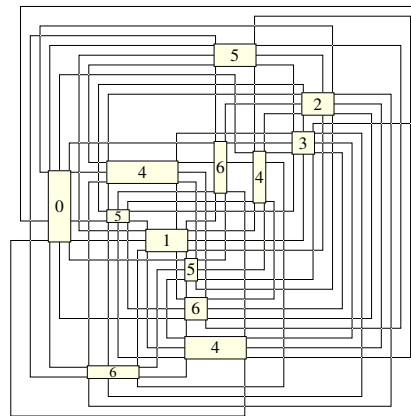
(24)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 2, 3, 4}) \leq 144$ .

Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.

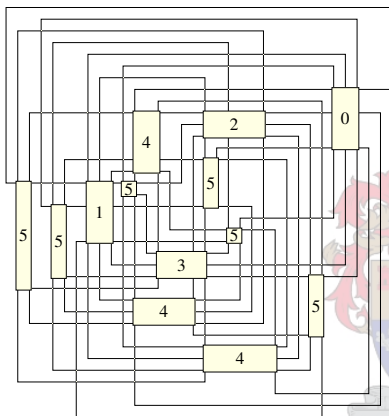




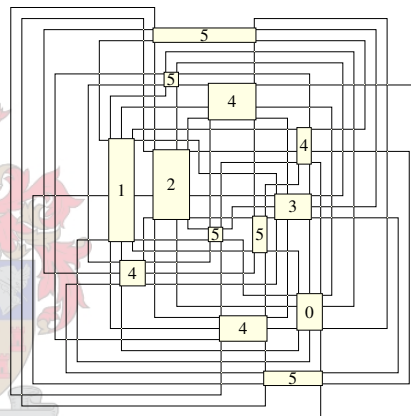
(25)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 2, 7}) \leq 90.$



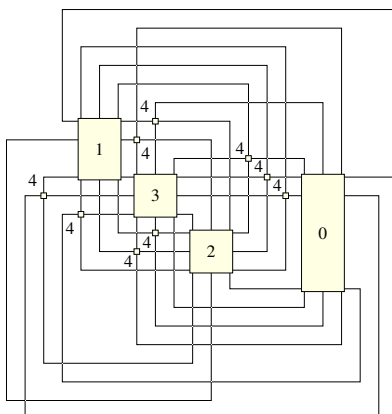
(26)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 3, 3, 3}) \leq 156.$



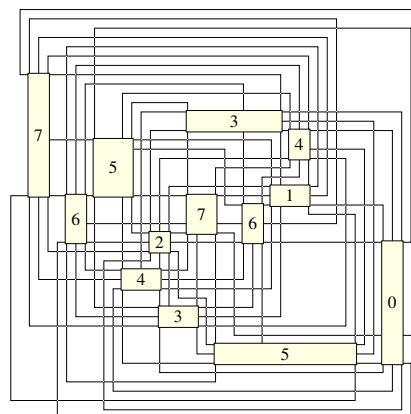
(27)  $56 \leq \nu(\mathcal{K}_{4 \times 1, 3, 6}) \leq 110.$



(28)  $54 \leq \nu(\mathcal{K}_{4 \times 1, 4, 5}) \leq 119.$

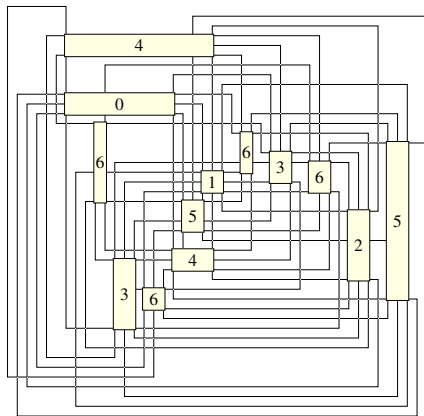


(29)  $\nu(\mathcal{K}_{4 \times 1, 9}) = 41.$

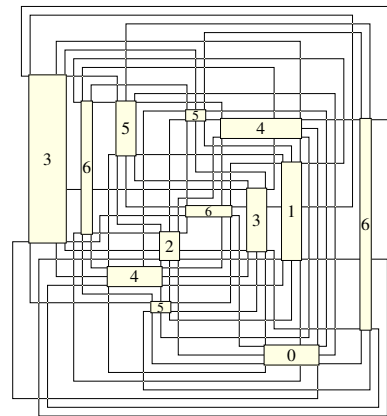


(30)  $56 \leq \nu(\mathcal{K}_{1, 1, 1, 5 \times 2}) \leq 161.$

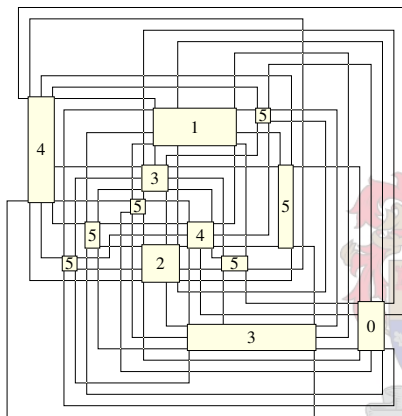
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



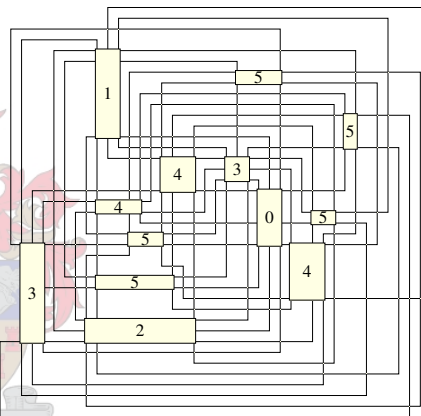
(31)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,2,2,4}) \leq 143.$



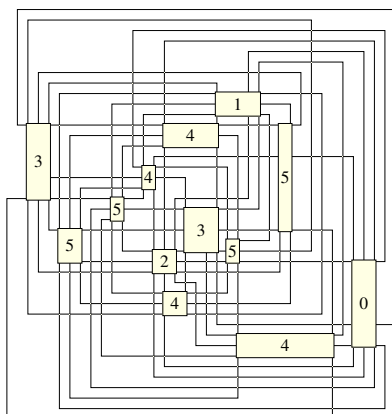
(32)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,2,3,3}) \leq 152.$



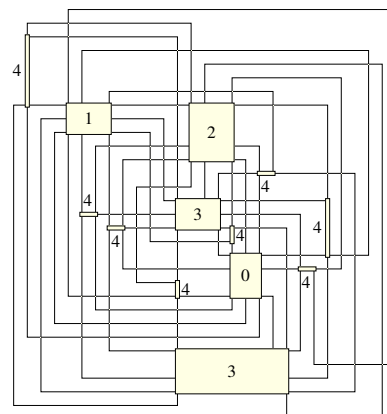
(33)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,2,6}) \leq 109.$



(34)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,3,5}) \leq 126.$

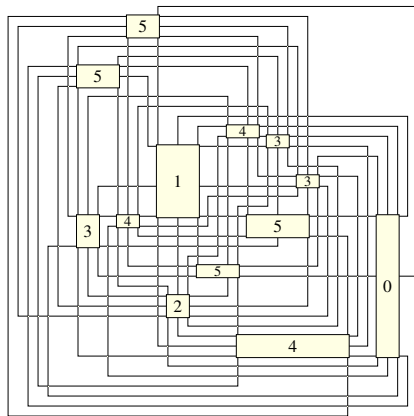


(35)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,4,4}) \leq 128.$

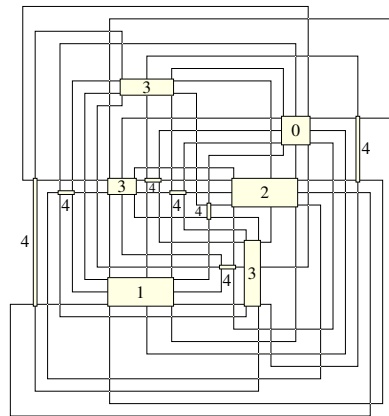


(36)  $56 \leq \nu(\mathcal{K}_{1,1,1,2,8}) \leq 64.$

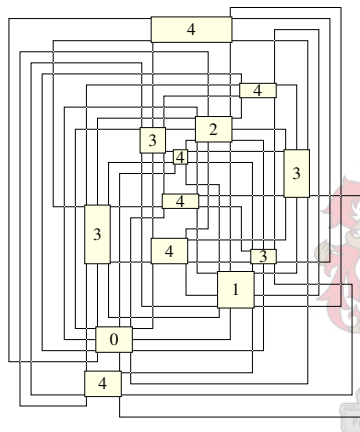
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



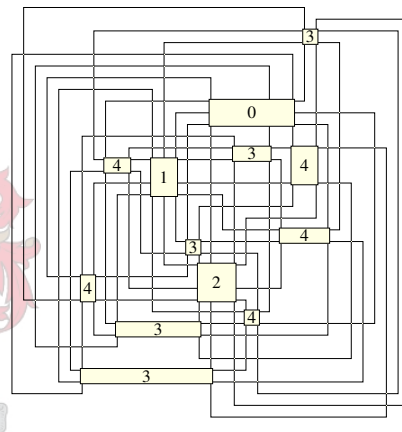
(37)  $56 \leq \nu(\mathcal{K}_{1,1,1,3,3,4}) \leq 136$ .



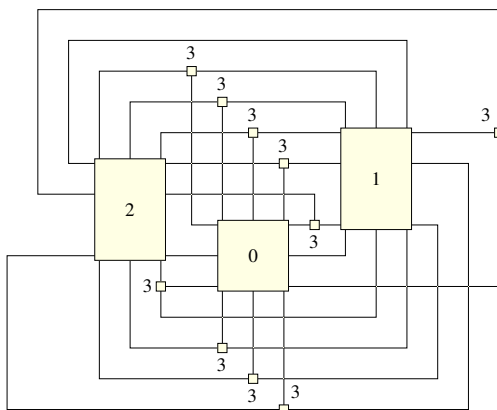
(38)  $56 \leq \nu(\mathcal{K}_{1,1,1,3,7}) \leq 86$ .



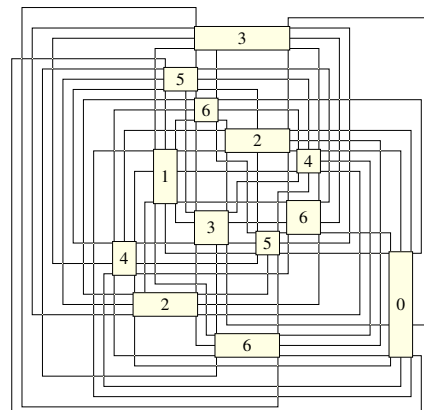
(39)  $54 \leq \nu(\mathcal{K}_{1,1,1,4,6}) \leq 98$ .



(40)  $54 \leq \nu(\mathcal{K}_{1,1,1,5,5}) \leq 105$ .

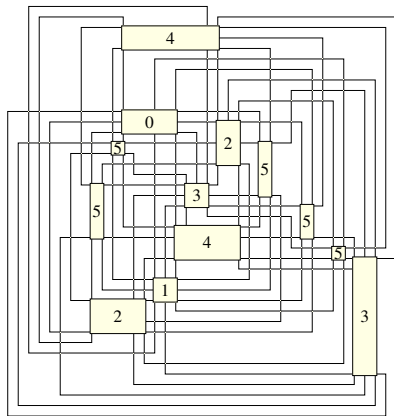


(41)  $\nu(\mathcal{K}_{1,1,1,10}) = 20$ .

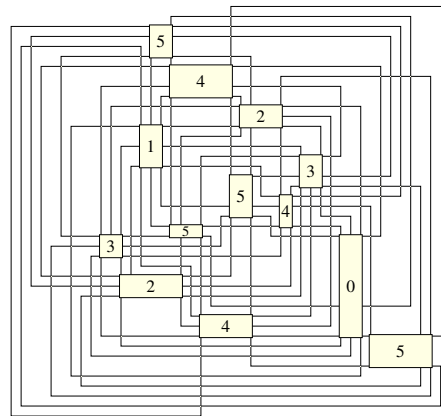


(42)  $56 \leq \nu(\mathcal{K}_{1,1,4 \times 2,3}) \leq 150$ .

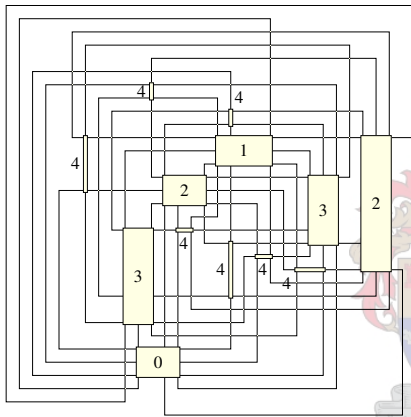
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



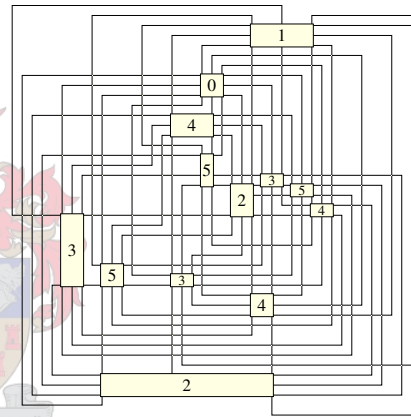
(43)  $56 \leq \nu(\mathcal{K}_{1,1,2,2,2,5}) \leq 123.$



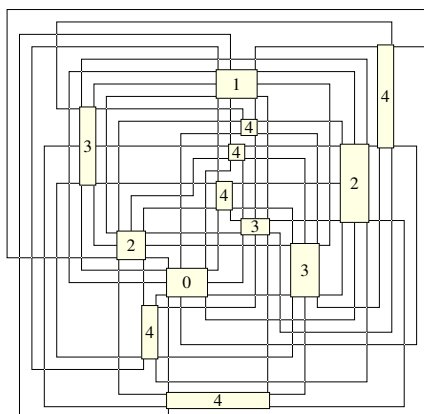
(44)  $56 \leq \nu(\mathcal{K}_{1,1,2,2,3,4}) \leq 134.$



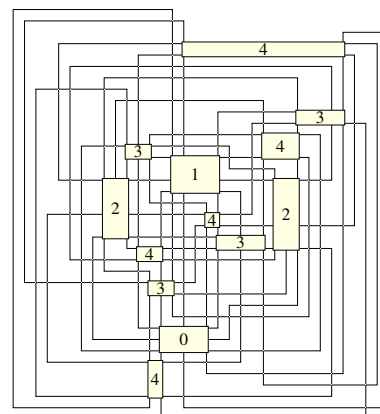
(45)  $56 \leq \nu(\mathcal{K}_{1,1,2,2,7}) \leq 82.$



(46)  $56 \leq \nu(\mathcal{K}_{1,1,2,3,3,3}) \leq 144.$

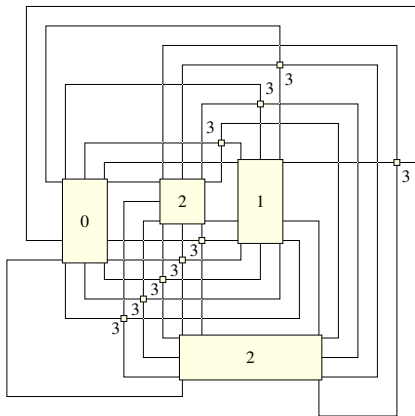


(47)  $56 \leq \nu(\mathcal{K}_{1,1,2,3,6}) \leq 102.$

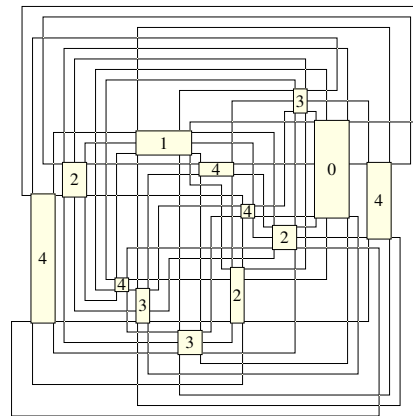


(48)  $54 \leq \nu(\mathcal{K}_{1,1,2,4,5}) \leq 110.$

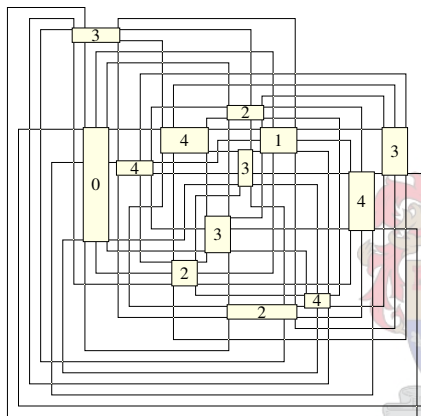
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



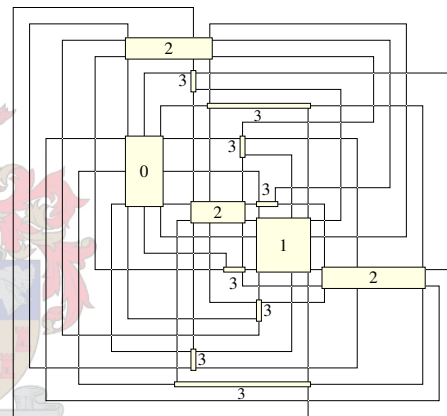
(49)  $\nu(\mathcal{K}_{1,1,2,9}) = 36.$



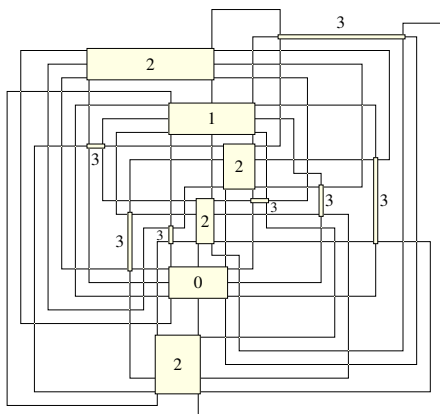
(50)  $56 \leq \nu(\mathcal{K}_{1,1,3,3,5}) \leq 120.$



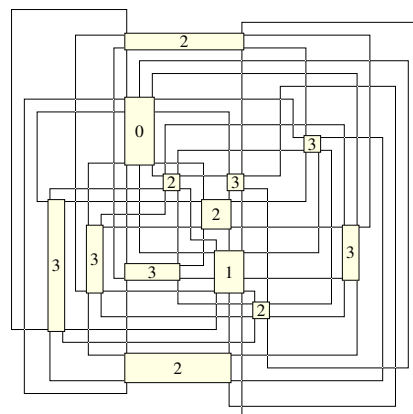
(51)  $56 \leq \nu(\mathcal{K}_{1,1,3,4,4}) \leq 120.$



(52)  $56 \leq \nu(\mathcal{K}_{1,1,3,8}) \leq 60.$

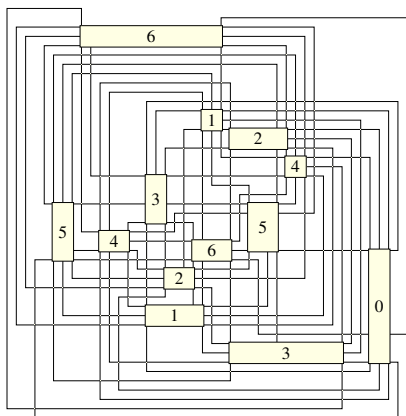


(53)  $54 \leq \nu(\mathcal{K}_{1,1,4,7}) \leq 74.$

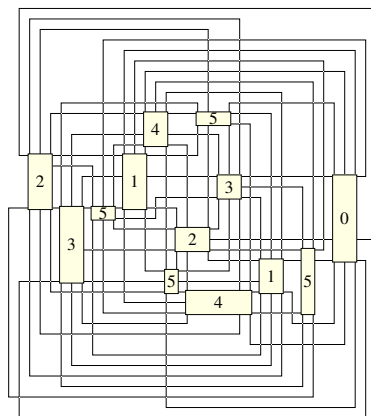


(54)  $54 \leq \nu(\mathcal{K}_{1,1,5,6}) \leq 84.$

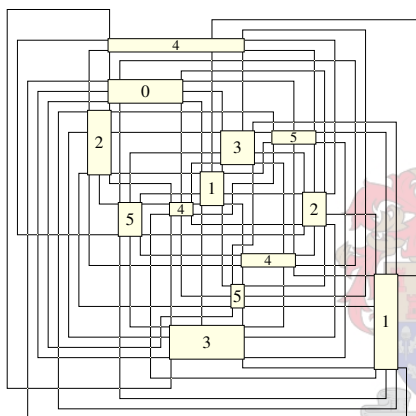
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



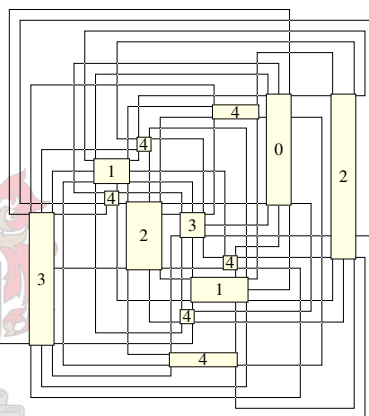
(55)  $56 \leq \nu(\mathcal{K}_{1,6 \times 2}) \leq 152$ .



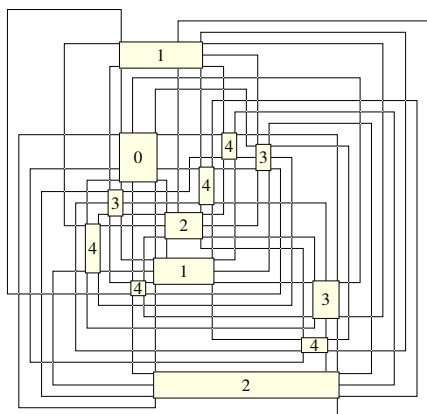
(56)  $56 \leq \nu(\mathcal{K}_{1,4 \times 2,4}) \leq 134$ .



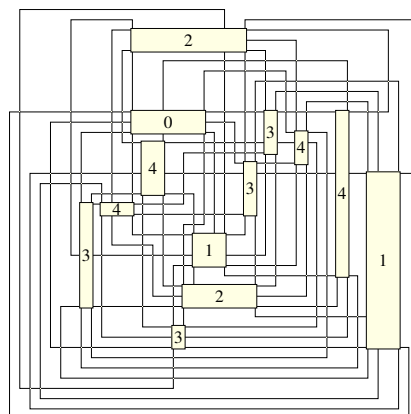
(57)  $56 \leq \nu(\mathcal{K}_{1,2,2,2,3,3}) \leq 141$ .



(58)  $56 \leq \nu(\mathcal{K}_{1,2,2,2,6}) \leq 102$ .

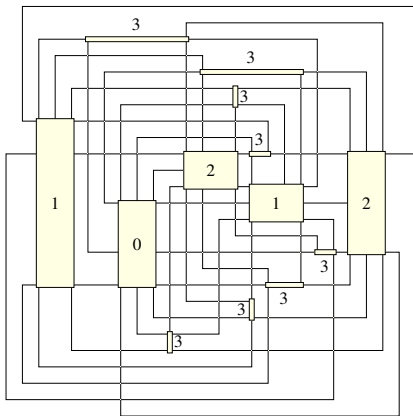


(59)  $56 \leq \nu(\mathcal{K}_{1,2,2,3,5}) \leq 116$ .

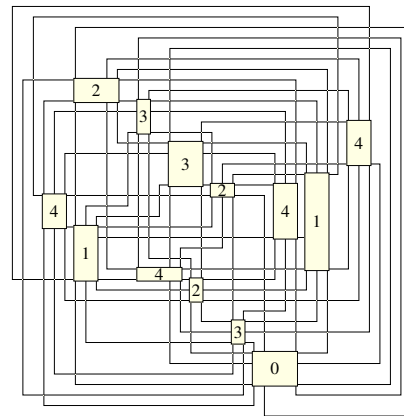


(60)  $56 \leq \nu(\mathcal{K}_{1,2,2,4,4}) \leq 120$ .

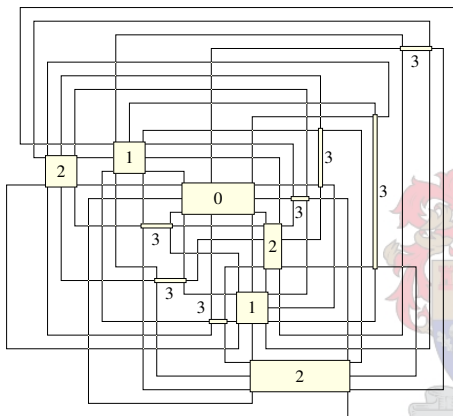
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



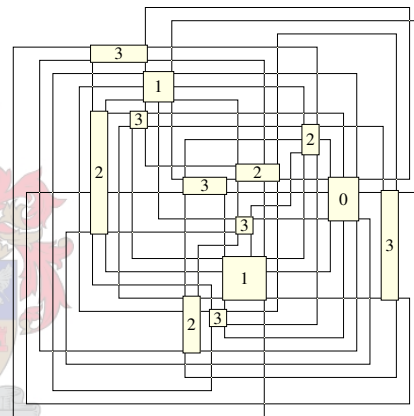
(61)  $56 \leq \nu(\mathcal{K}_{1,2,2,8}) \leq 60$ .



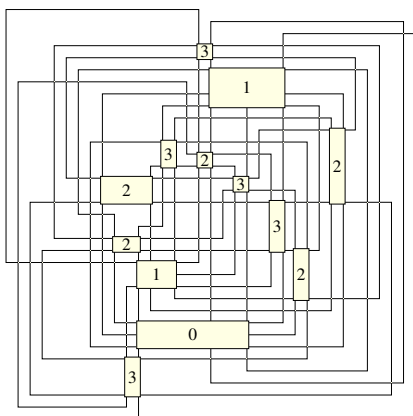
(62)  $56 \leq \nu(\mathcal{K}_{1,2,3,3,4}) \leq 126$ .



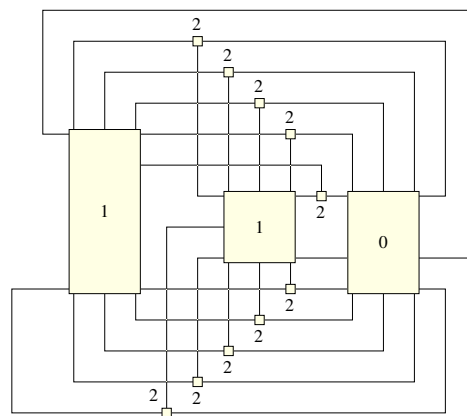
(63)  $56 \leq \nu(\mathcal{K}_{1,2,3,7}) \leq 78$ .



(64)  $54 \leq \nu(\mathcal{K}_{1,2,4,6}) \leq 92$ .

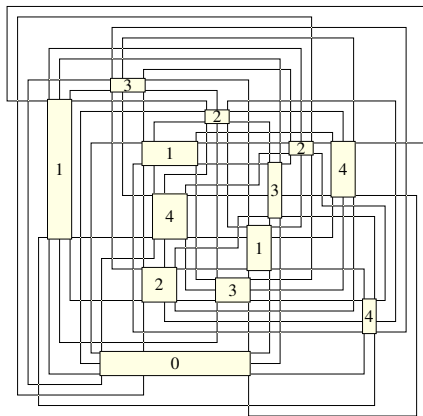


(65)  $54 \leq \nu(\mathcal{K}_{1,2,5,5}) \leq 96$ .

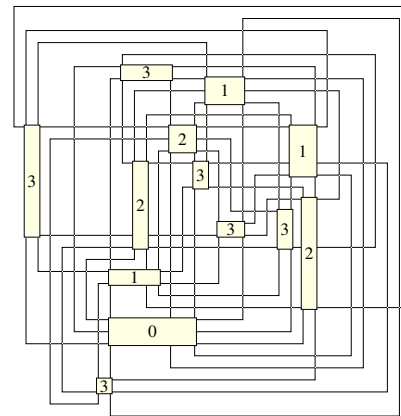


(66)  $\nu(\mathcal{K}_{1,2,10}) = 20$ .

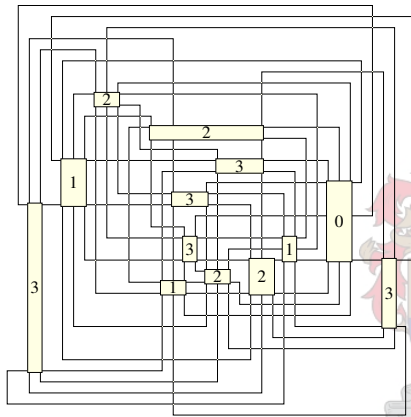
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



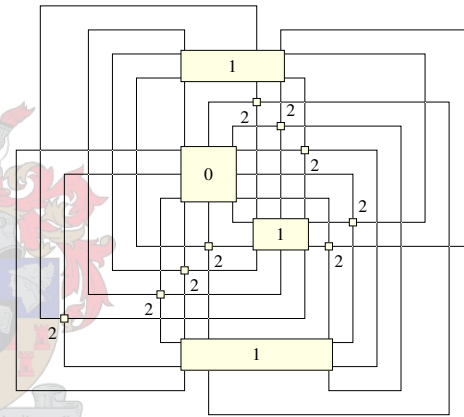
(67)  $54 \leq \nu(\mathcal{K}_{1,4 \times 3}) \leq 137$ .



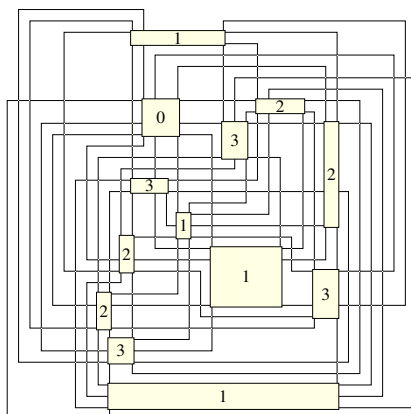
(68)  $54 \leq \nu(\mathcal{K}_{1,3,3,6}) \leq 96$ .



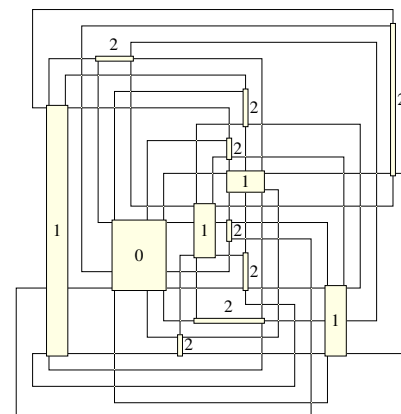
(69)  $54 \leq \nu(\mathcal{K}_{1,3,4,5}) \leq 104$ .



(70)  $\nu(\mathcal{K}_{1,3,9}) = 36$ .



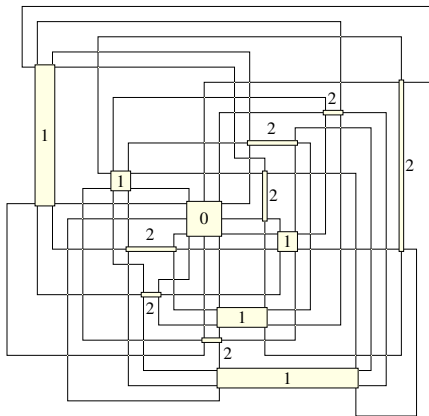
(71)  $48 \leq \nu(\mathcal{K}_{1,4,4,4}) \leq 108$ .



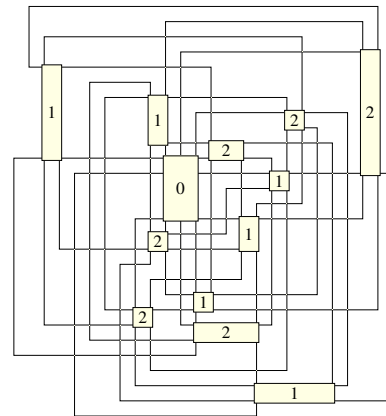
(72)  $48 \leq \nu(\mathcal{K}_{1,4,8}) \leq 56$ .

Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.

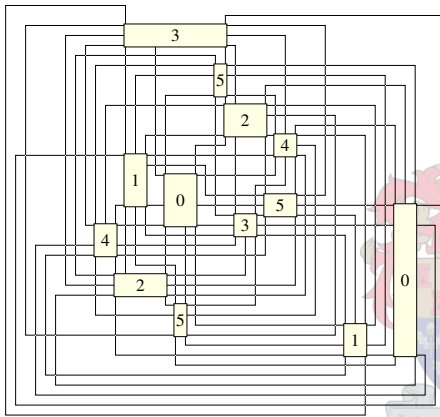




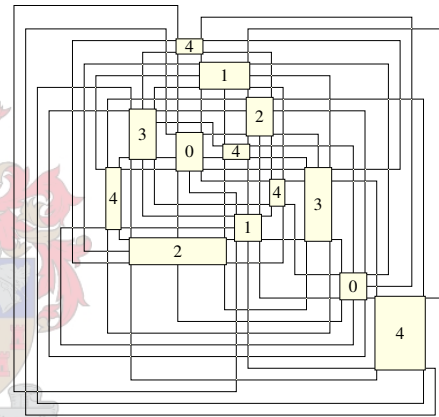
$$(73) \quad 54 \leq \nu(\mathcal{K}_{1,5,7}) \leq 66.$$



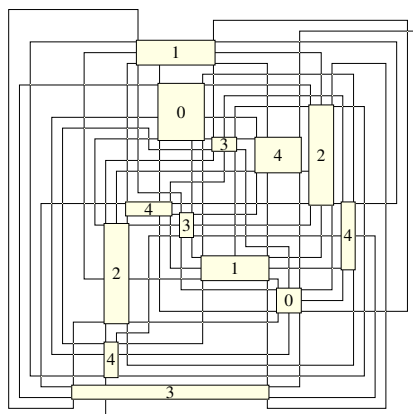
$$(74) \quad 54 \leq \nu(\mathcal{K}_{1,6,6}) \leq 72.$$



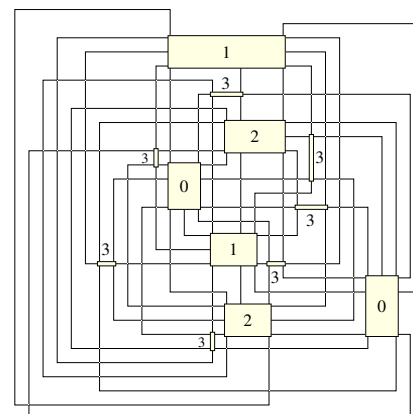
$$(75) \quad 56 \leq \nu(\mathcal{K}_{5 \times 2,3}) \leq 141.$$



$$(76) \quad 54 \leq \nu(\mathcal{K}_{4 \times 2,5}) \leq 114.$$

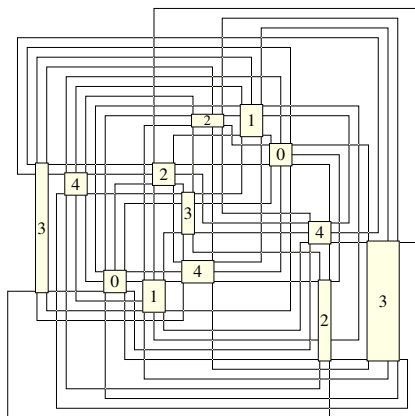


$$(77) \quad 56 \leq \nu(\mathcal{K}_{2,2,2,3,4}) \leq 125.$$

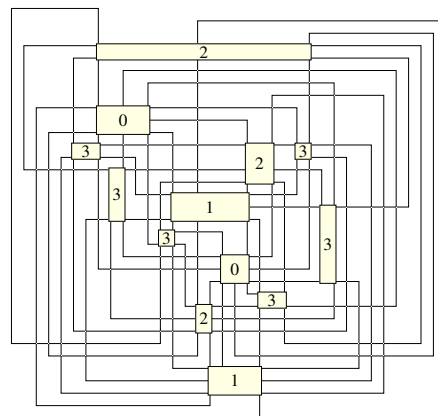


$$(78) \quad 54 \leq \nu(\mathcal{K}_{2,2,2,7}) \leq 75.$$

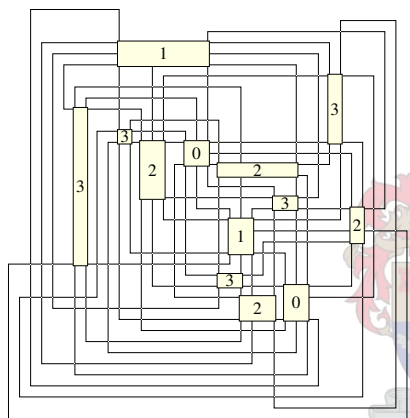
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



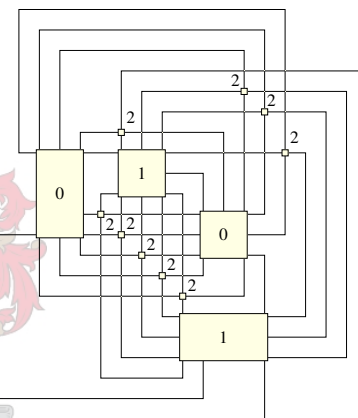
(79)  $56 \leq \nu(\mathcal{K}_{2,2,3,3,3}) \leq 133.$



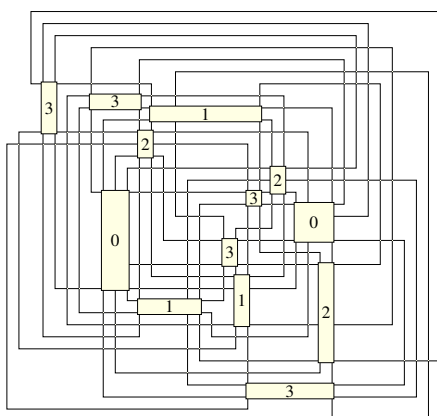
(80)  $56 \leq \nu(\mathcal{K}_{2,2,3,6}) \leq 95.$



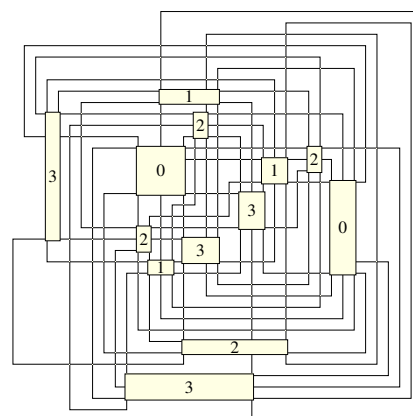
(81)  $54 \leq \nu(\mathcal{K}_{2,2,4,5}) \leq 102.$



(82)  $\nu(\mathcal{K}_{2,2,9}) = 32.$

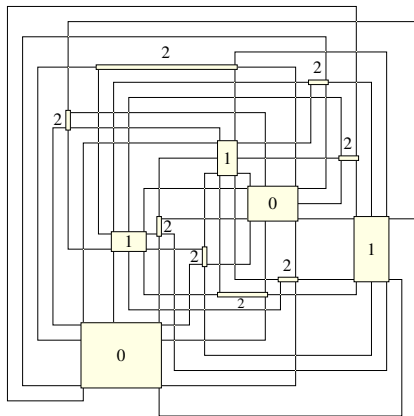


(83)  $56 \leq \nu(\mathcal{K}_{2,3,3,5}) \leq 110.$

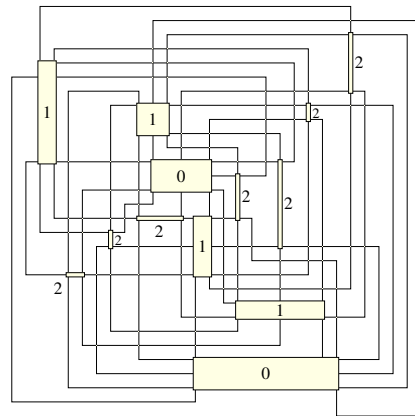


(84)  $56 \leq \nu(\mathcal{K}_{2,3,4,4}) \leq 112.$

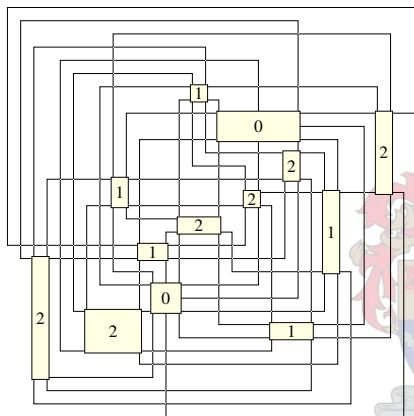
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



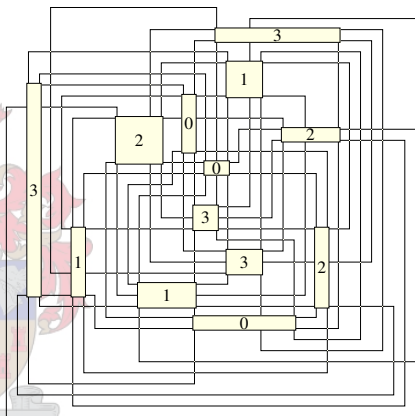
(85)  $\nu(\mathcal{K}_{2,3,8}) = 56$ .



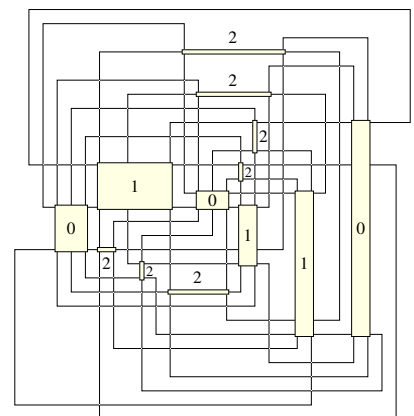
(86)  $54 \leq \nu(\mathcal{K}_{2,4,7}) \leq 68$ .



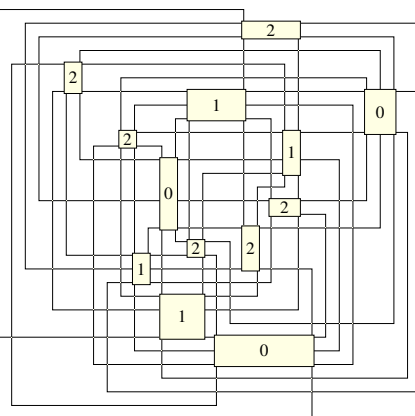
(87)  $54 \leq \nu(\mathcal{K}_{2,5,6}) \leq 78$ .



(88)  $54 \leq \nu(\mathcal{K}_{3,3,3,4}) \leq 119$ .

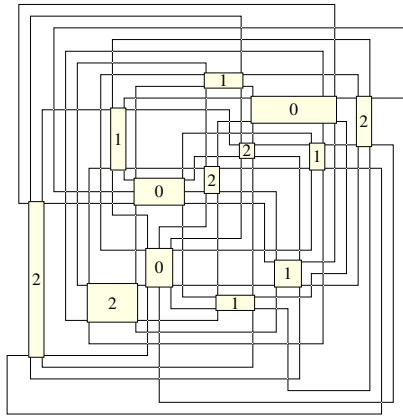


(89)  $54 \leq \nu(\mathcal{K}_{3,3,7}) \leq 75$ .

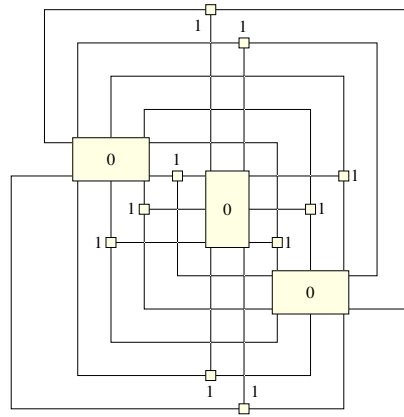


(90)  $54 \leq \nu(\mathcal{K}_{3,4,6}) \leq 86$ .

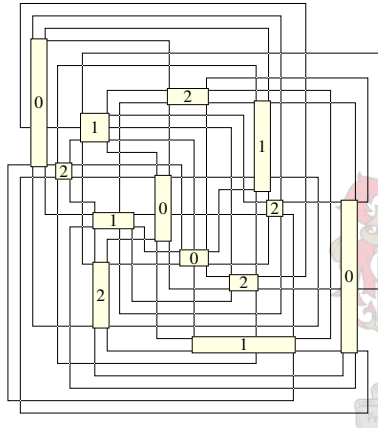
Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.



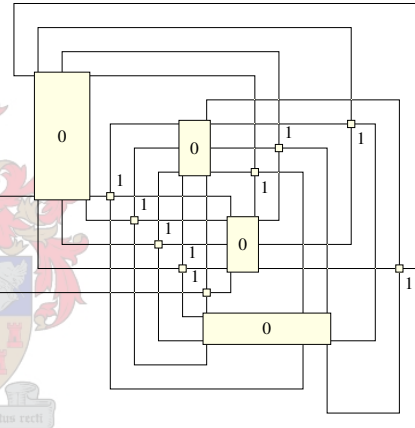
(91)  $48 \leq \nu(\mathcal{K}_{3,5,5}) \leq 92.$



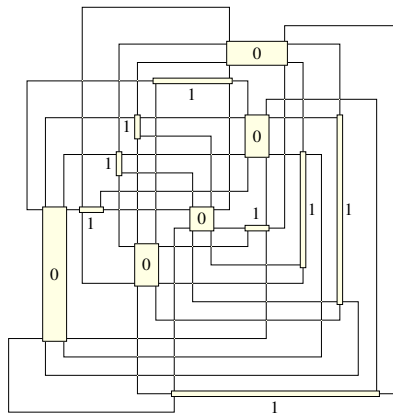
(92)  $\nu(\mathcal{K}_{3,10}) = 20.$



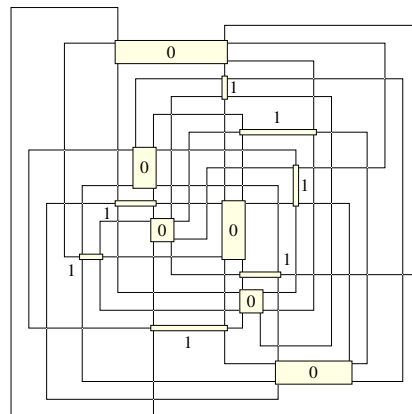
(93)  $48 \leq \nu(\mathcal{K}_{4,4,5}) \leq 92.$



(94)  $\nu(\mathcal{K}_{4,9}) = 32.$



(95)  $\nu(\mathcal{K}_{5,8}) = 48.$



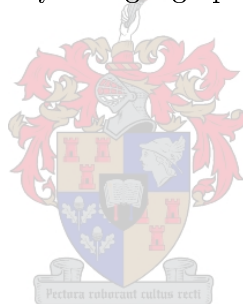
(96)  $\nu(\mathcal{K}_{6,7}) = 54.$

Figure 7.16 (continued): Non-planar complete multipartite graphs of order 13.

## 7.3 Chapter summary

In the first section of this chapter, § 7.1, the convergence properties of the upper bound algorithms implemented in this thesis were discussed. It was shown that the rate of convergence of the **GreedySide** algorithm is favourable in comparison to the Cimikowski–Shope [CS96] algorithm, and especially to the genetic edge layout algorithm in § 6.2.1.3.

The output of the upper bound algorithms was considered in § 7.1. It was first shown that the **GreedySide** algorithm performs poorly on subdivided graphs, whilst the Cimikowski–Shope neural network algorithm seems to fare no worse with such graphs. In the next section, § 7.2.2, the output of the lower bound algorithm, as applied to some hypercubes, was shown. It was pointed out that the algorithm is very sensitive to the graph that is chosen to be graph-to-graph embedded into the graph of which the crossing number is sought. It was found that certain bipartite graphs are useful in this context, whilst the complete graphs do not seem to yield good results when graph-to-graph embedded into hypercubes. The output of the Garey–Johnson implementation was considered in § 7.2.3. An example of a crossing configuration that did not lead to a planar graph (that contains artificial vertices modelling crossings) was demonstrated, as well as an example of a crossing configuration that did lead to a planar graph. Finally, a catalogue of drawings of all non-planar multipartite graphs of orders 6–13 was given realising crossing number upper bounds as determined by the tabu search algorithm. It was seen that the crossing number upper bound of only a single graph, namely  $\mathcal{K}_{1,2,2,2,2}$ , was improved after subdivision.





# Chapter 8

## Conclusions

Every human activity, good or bad, except mathematics,  
must come to an end.

— *Paul Erdős (1913–1996)*

A brief summary of the results obtained during research for this thesis is presented in § 8.1. In § 8.2, a number of open questions (and reasons for their importance) are posed. It is hoped that these questions will elicit further research.

### 8.1 Summary of work contained in this thesis

Chapter 3 was dedicated to finding precise and unambiguous definitions for the concepts of a graph drawing and the crossing number of a graph, as vagueness about these concepts has led to varying interpretations in the past.

The literature survey of Chapter 4 reveals that the graph crossing number problem has become a widely researched topic. Several analytical techniques were introduced, and from an algorithmic point of view, there are a number of proposed algorithms. Some heuristic algorithms have been implemented, although some others still have not (such as the probabilistic embedding algorithm of Shahrokhi, Székely, Sýkora and Vrřo [SSSV96c]).

The first main aim of Chapter 5 was to provide an implementation of the Garey–Johnson [GJ83] algorithm, and to consider methods by which the algorithm could exploit symmetries in graphs so as to reduce the number of computations that it has to perform. The second aim of the chapter was to make it clear that book drawings are not simply hobbled versions of plane drawings, but that any plane drawing is representable as a book drawing, if the graph of which the book drawing is to be made is subdivided sufficiently. It was also shown that a book drawing may be subdivided without increasing the number of crossings in the drawing, thereby making it possible to refine book drawings (*i.e.*, reduce the number of crossings), by subdividing its edges, and by shifting its subdivision edges around on the spine in order to decrease the number of crossings.

Chapter 6 dealt with heuristic methods for approximating the crossing number of a graph. A lower bound algorithm was developed, which is believed by the author to be the first general algorithm for finding lower bounds to graph crossing numbers in the plane. The theory behind this algorithm was extended from the concept of graph-to-graph embedding. Later in this chapter, a two-tier upper bound approximation framework was developed, where at the first level, a tabu search algorithm searches for vertex arrangements and, in turn, “drives” an edge

layout algorithm, which is responsible for finding good edge layouts. Two edge layout heuristics were also developed. One is a simple iterating greedy algorithm, and the other is a more elaborate genetic algorithm. Finally, it was shown how ideas from Székely’s independent–odd crossing number algorithm may be merged with the two–page book layout framework so as to produce an upper bound algorithm for the plane crossing number of the graph, in which the vertex arrangement may remain static.

Finally, Chapter 7 dealt with the issues that surround the convergence of the heuristic methods, as well as with the output produced by the algorithms that were programmed for this thesis. A catalogue of drawings and upper bounds to the crossing numbers of non–planar complete multipartite graphs of orders 6–13 was also given in this chapter. The question of the number of iterations required for the **GreedySide** algorithm was studied, and statistical results regarding its behaviour were presented, suggesting that it generally performs very well. Finally, lower bounds on the crossing number for some hypercube graphs were provided, thereby improving upon best known lower bounds for these graphs.

## 8.2 Possible future work and open questions

There is much potential in improving the two–page book layout algorithms. In this thesis, all of the edges of a graph were subdivided simultaneously, which slowed down the tabu search algorithm significantly, due to the usually large number of subdivision vertices that were added to the spine. A technique which adapts by subdividing only a few edges at a time, and by removing subdivisions for edges which do not require them, could achieve higher levels of efficiency, whilst ensuring that the number of crossings decrease as far as possible.

The upper bound algorithm, discussed in § 6.3, which is based on ideas from Székely’s algorithm and book layouts, only occurred to the author at a very late stage; hence the lack of an implementation. This algorithm holds promise, because it removes the problem of having to determine vertex arrangements. Furthermore, the author believes that algorithms such as the neural network algorithm of Cimikowski and Shope [CS96] should be adaptable to this problem without much trouble.

As for the lower bound algorithm, it has to be said that the mechanism for creating edge embeddings could be improved, since it is based on a simple idea. Perhaps a post–optimization technique could be developed to fulfil this role. Another problem that was barely touched upon in this thesis, is the determination of good vertex embeddings.

During the course of the author’s research for this thesis, a number of interesting problems presented themselves. Most of these problems are difficult, but certainly warrant further investigation. A total of five have been selected to serve as possible starting points for brave researchers.

**Question 8.2.1** *Given a graph  $\mathcal{G}$ , what is the smallest number of edge subdivisions of  $\mathcal{G}$ , which would ensure that the resulting subdivided graph  $\mathcal{H}$  has a two–page layout which could achieve a total of  $\nu(\mathcal{G})$  crossings? Denote this value by  $z(\mathcal{G})$ . Is it possible to bound  $z(\mathcal{G})$  in terms of an easily computable parameter of  $\mathcal{G}$ ? ■*

If it is known that no edge in a graph  $\mathcal{G}$  may be crossed more than  $t$  times in a drawing of  $\mathcal{G}$  which realises its crossing number  $\nu(\mathcal{G})$ , then according to Theorem 5.3.2,  $z(\mathcal{G}) \leq (t + 1)(|V(\mathcal{G})| - 2)$ . However, this bound is not of practical use, since most edges would be crossed



a large number of times. In the hybrid two–page/Székely algorithm described in § 6.3, each edge has to be subdivided at most  $|V(\mathcal{G})| - 3$  times to ensure that all independent–odd drawings would be enumerated. This is a much better bound, and it would possibly make the problem of computing the exact crossing numbers of small graphs tractable when employing the two–page layout paradigm.

From the results obtained by the algorithms, the author conjectures that  $z(\mathcal{K}_n) = 0$  and that  $z(\mathcal{K}_{m,n}) = 0$ . This would trivially follow if it could be shown that the crossing numbers of these graphs are equal to their upper bound constructed values (§ 4.2.3.3 and § 4.2.3.2), since these constructions directly permit two–page layouts, without subdivisions. Furthermore, the author conjectures that for any graph  $\mathcal{G}$ , it holds that  $z(\mathcal{G}) \leq |V(\mathcal{G})| - 3$ , in accordance with the number of subdivisions required for the two–page/Székely algorithm.

**Question 8.2.2** *For an  $r$ -regular graph in general, which symmetries may be exploited by the Garey–Johnson algorithm (§ 4.3.1.1 and § 5.2) to reduce the total number of computations? ■*

In § 5.2, it was discussed how some symmetry information may be used to reduce the total number of computations in complete multipartite graphs. For these graphs alone, there are invariably a number of other symmetry considerations that could be taken into account. However, there are many graphs with a degree of regularity, and the exploitation of this structure could render the Garey–Johnson algorithm useful in determining the crossing numbers of a number of small graphs.

If it is possible to identify a large number of symmetries in well-known classes of graphs, then perhaps the Garey–Johnson algorithm could become a useful tool.

**Question 8.2.3** *In the lower bound algorithm (§ 6.1), when a graph  $\mathcal{G}$  is graph–to–graph embedded into a graph  $\mathcal{H}$ , which other considerations could be taken into account to discount crossings that would normally be counted because paths (which are the images of edges in  $\mathcal{G}$ ) share vertices in  $\mathcal{H}$ ? ■*

It is shown in § 6.1, that when a pair of paths share a sub–path, it is not necessary to count more than one crossing for the entire sub–path. A consequence of this observation is that when a pair of paths start at the same vertex, they need not cross each other for the sub–path they share, starting at that vertex.

The vertex congestion terms play a large role in determining the quality of the obtained lower bounds. If it is possible to discount any other crossings, then the lower bound would improve.

**Question 8.2.4** *The standard counting method (§ 4.2.2.1) is generally used as an analytical tool. In such a setting, when a lower bound for a graph  $\mathcal{G}$  must be determined, copies of a single type of subgraph for which the lower bound is known (where the number of vertices are fixed) are counted in  $\mathcal{G}$  (for example, bipartite graphs have been used in this way — see § 4.2.3.3). There is no reason why a number of heterogenous subgraphs could not be found in this way. This would be necessitated by graphs which are not entirely symmetrical. However, it would only be sensible to proceed algorithmically with such a scheme. Is it possible to formulate efficient algorithms for this task, that would compete with, or outperform, the lower bound algorithm described in § 6.1? ■*

This is certainly a field that requires more attention. The main problem is that generally, the larger the subgraphs, the better the lower bound. However, the problem of matching of subgraphs is really the problem of determining whether there are subgraphs that are isomorphic to subgraphs for which lower bounds are known. This represents a major hurdle, since the general subgraph isomorphism problem is an **NP**-complete problem. However, many techniques have been developed to solve the isomorphism problem efficiently for certain classes of graphs. The software library *nauty*, authored by McKay [McK90], implements a sophisticated algorithm that is very time efficient in determining whether two arbitrary graphs are isomorphic. Thus, this library represents a good starting point for the implementation of such a lower bound algorithm, and may be downloaded from the Internet.

**Question 8.2.5** *It was noted in, § 3.2.3, that the  $n$ -planar crossing number of a graph is related to the book crossing number on  $2n$  pages, as  $\nu_n^{(B)}(\mathcal{G}) \leq \nu_{2n}(\mathcal{G})$ . Is there an integer  $n_1$ , so that  $\nu_{n_1}^{(B)}(\mathcal{G}) < \nu_{2n_1}(\mathcal{G})$  for any graph  $\mathcal{G}$ ? If this is true, then for which value  $n_2 < 2n_1$  would it hold that  $\nu_{n_1}^{(B)}(\mathcal{G}) < \nu_{n_2}(\mathcal{G})$  for an arbitrary graph  $\mathcal{G}$ ? ■*

This question is not directly related to the work contained in this thesis. However, this seems to be an interesting problem, since at least the greedy page layout algorithm (§ 6.2.1.1) and the neural-network layout algorithm (§ 6.2.1.2) may readily be adapted to approximate the book crossing number for an arbitrary number of pages.

First one has to note that the intersection graph (§ 6.2.1) for an  $n$ -page book layout, has  $n$  possible vertex partitions. The edge layout algorithms therefore have to be designed to take this into account. The greedy layout algorithm, **GreedySide** (§ 6.2.1.1) requires no modification. For an edge  $e$ , when it has to decide on which page  $e$  is to be placed, it simply has a larger number of pages to consider. The neural-network algorithm (§ 6.2.1.2) also requires very little modification. It has up and down functions  $U^{(\uparrow)}$  and  $U^{(\downarrow)}$  for each edge in the input graph  $\mathcal{G}$ , and the values of these functions depend on whether an edge is to be drawn on the upper or lower page (or both if the algorithm has not yet converged). In the generalized case of  $n$  pages, page  $i$  has a set of functions  $U^{(i)}$ , and convergence of the neural network algorithm is reached when for each edge  $e \in E(\mathcal{G})$ , there is only one function  $U_e^{(j)}$ ,  $1 \leq j \leq n$  for which  $U_e^{(j)} > 0$  (*i.e.*  $e$  may unambiguously be assigned to page  $i$ ).

An affirmative answer to this question would make it possible to implement generalizations of the upper bound techniques considered in § 6.2 to bound the  $n$ -planar crossing number of a graph.

# Appendix A

## Kuratowski's Theorem

Mathematical proofs, like diamonds, are hard as well as clear,  
and will be touched with nothing but strict reasoning.

— John Locke (1632–1704)

In 1930, Kazimierz Kuratowski gave a proof to a theorem characterizing all planar graphs in terms of subdivisions of only two forbidden subgraphs. It is one of the most significant and beautiful results in Topological Graph Theory.

**Theorem A.0.1** (Kuratowski [Kur30], 1930) *A graph  $\mathcal{G}$  is planar if and only if it contains no subgraph isomorphic to a subdivision of  $\mathcal{K}_5$  or  $\mathcal{K}_{3,3}$ .*

This particular proof closely follows the proof of C. Thomassen [Tho81]. Proving that the presence of a subdivision of either  $\mathcal{K}_5$  or  $\mathcal{K}_{3,3}$  in graph  $\mathcal{G}$  renders  $\mathcal{G}$  non-planar is straightforward. By Theorem 2.1.5,  $\mathcal{K}_5$  and  $\mathcal{K}_{3,3}$  are both non-planar; furthermore, according to Theorem 2.1.1, a subdivision of a non-planar graph is also non-planar. Combining these results yields the required result. The rest of the argument is dedicated to proving the converse of this statement.

A number of auxiliary concepts used for this part of the proof are introduced first, with a subsection dedicated to each concept. Firstly, a subgraph called a *C-component*, is defined. This structure simplifies the study of crossing interactions between a pair of C-components, if the C-components are assumed to be planar. Secondly, a graph derived from  $\mathcal{G}$ , of which the structure depends on the C-components of  $\mathcal{G}$  is introduced. This structure, known as the *overlap graph* of  $\mathcal{G}$ , is shown to be non-bipartite, which implies certain configurations of the associated C-components. Finally, it is shown that a configuration of C-components derived from the overlap graph of  $\mathcal{G}$  yields either a subdivision of  $\mathcal{K}_5$  or a subdivision of  $\mathcal{K}_{3,3}$  in  $\mathcal{G}$ .

### A.1 Connectedness

An important assumption of the proof is that each vertex be part of a cycle in  $\mathcal{G}$ . If  $\mathcal{G}$  violates this assumption, the concept of graph connectedness may be used to partition  $\mathcal{G}$  into subgraphs for which the assumption is true. Kuratowski's theorem may then be applied to the subgraphs.

**Definition A.1.1** *A graph  $\mathcal{G}$  is said to be  $k$ -connected, if the removal of any set of  $j$  vertices, where  $j < k$ , does not disconnect  $\mathcal{G}$ .*

The following proposition follows directly from the definition of  $k$ -connectedness.

**Proposition A.1.1** *If a graph is  $k$ -connected, it is also  $(k - 1)$ -connected.* ■

## A.2 Overview of the method of proof

This part of the proof is by the strong form of induction over the order of the graph. Let  $\mathcal{G}$  be a 2-connected *non-planar* graph of order  $n$ . As the induction hypothesis, assume that Kuratowski’s theorem is true for all 2-connected graphs of order less than  $n$ . It will be shown by contradiction that  $\mathcal{G}$  necessarily contains a subdivision of  $\mathcal{K}_5$  or  $\mathcal{K}_{3,3}$  as subgraph. Therefore assume that this is not the case — this implies that every subgraph of  $\mathcal{G}$  of order less than  $n$  may not contain a subdivision of  $\mathcal{K}_5$  or  $\mathcal{K}_{3,3}$  and must therefore be planar according to the induction hypothesis.

The contradiction assumption forces  $\mathcal{G}$  to be a vertex critical non-planar graph; that is,  $\mathcal{G}$  is such, that the removal of any vertex from  $V(\mathcal{G})$  renders  $\mathcal{G}$  planar.

## A.3 Handling 1-connected graphs and subdivided graphs

It is assumed that the graph  $\mathcal{G}$  of which the planarity is under question is 2-connected. In the case that  $\mathcal{G}$  is 1-connected, but not 2-connected, it must contain at least one *cut-vertex* (*i.e.*, a vertex whose removal disconnects the graph into at least two components) as shown in Figure A.1(a). Let  $v$  be a cut-vertex and let  $C$  be a component that results from the removal of  $v$  in  $\mathcal{G}$ . Define a new subgraph  $C' = \langle V(C) \cup \{v\} \rangle$  — if such subgraphs are created for each component that was created with the removal of  $v$ , then the new subgraphs will contain all edges in  $\mathcal{G}$  and they will have only the vertex  $v$  in common. An example of such subgraphs is shown in Figure A.1(b). If each of these subgraphs is planar, then it will be possible to ensure that  $v$  is drawn in the outer region for each of the plane drawings of the subgraphs. All of the subgraphs may then be “glued” together by coalescing the vertex  $v$  in each of the subgraphs. This leads to a planar configuration.

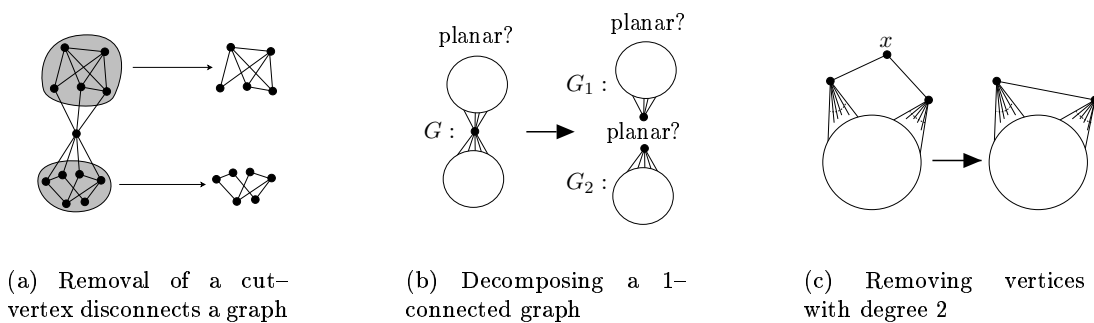


Figure A.1: Transformations to handle 1-connectivity and vertices of degree 2 in the proof of Kuratowski’s theorem.

Vertices of degree 2 may be regarded as subdivision vertices and as such, according to Proposition 2.1.1, the replacement of paths containing only vertices of degree 2, with edges, does not alter the planarity or non-planarity of a graph. This action is illustrated in Figure A.1(c) for a path with a single subdivision vertex  $x$ .

## A.4 C-components

A C-component of a cycle  $C$  in a graph  $\mathcal{G}$  is, loosely speaking, the maximally connected subgraphs of  $\mathcal{G}$  that are attached to  $C$ , but that are not part of  $C$ .

**Definition A.4.1** *Given a cycle  $C$  in a graph  $\mathcal{G}$ , the union of a maximally connected component of the graph that results from the removal of  $C$  from  $\mathcal{G}$  (the removal of the cycle might leave several components) and the edges joining to the component to  $C$ , is called a C-component of  $\mathcal{G}$  with respect to  $C$  (or simply a C-component if the graph and cycle are clear from the context). The vertices of a C-component  $c$  that are in the vertex set of the cycle  $C$ , are called the clasp vertices (Thomassen [Tho81] called them vertices of attachment) of  $c$ , and the edges of  $c$  incident to the clasp vertices are called the clasp edges of  $c$  (they are called “feet” by some authors). ■*

To make this definition clear by means of a concrete example, consider Figure A.2. The various C-components with respect to the cycle  $C$ , labelled  $B_1, B_2, \dots, B_6$ , are shown in the grey regions. The white vertices indicate clasp vertices and the edges adjacent to the clasp vertices located in the grey regions are the clasp edges.

Note that each C-component is guaranteed to have at least two clasp vertices, due to the fact that  $\mathcal{G}$  is required to be 2-connected.

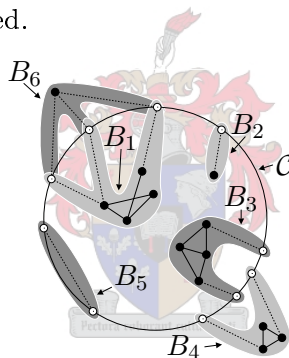


Figure A.2: An illustration of C-components.

C-components have also been called “bridges” by researchers such as Tutte [Tut77] — they should not be confused here with the conventional graph theoretic meaning of the word “bridge,” which is an edge whose removal separates a graph into two or more components.

**Definition A.4.2** *For a cycle  $C$  in a graph  $\mathcal{G}$ , a pair of C-components  $B_1$  and  $B_2$  of  $C$  are said to overlap if either*

1. *two clasp vertices from  $B_1$  alternate with at least two clasp vertices from  $B_2$  on the cycle  $C$ , in which case the C-components are said to be skew, or to overlap in a skew fashion — for example let  $x_1, x_2$  be clasp vertices of  $B_1$  and let  $y_1, y_2$  be clasp vertices of  $B_2$ ; then, in a walk around the cycle  $C$ , if the clasp vertices are encountered in the order  $x_1, y_1, x_2, y_2$ ,  $B_1$  and  $B_2$  are skew, or*
2. *if  $B_1$  and  $B_2$  have three of their clasp vertices in common, in which case they are said to be C-equivalent, or to overlap in a C-equivalent fashion.*

*The relation of overlapping is symmetric — if  $A$  overlaps  $B$ , then  $B$  overlaps  $A$ . ■*

From the definition of overlapping, it may be shown that the only type of overlappings between  $C$ -components that are  $C$ -equivalent, but not skew, are the cases in which the  $C$ -equivalent  $C$ -components have exactly three clasp vertices in common.

**Proposition A.4.1** *If two  $C$ -components  $B_1$  and  $B_2$  have more than three clasp vertices in common, they are skew.*

**Proof:** Four clasp vertices, say  $v_1, v_2, v_3$  and  $v_4$ , are required for the property of skewness in an overlapping of two  $C$ -components. Let the clasp vertices  $v_1$  and  $v_3$  be assigned to the  $C$ -component  $B_1$ , whilst the the clasp vertices  $v_2$  and  $v_4$  are assigned to the  $C$ -component  $B_2$ . According to the definition of overlapping, the respective  $C$ -components each contain two clasp vertices that are alternating, and the overlapping is therefore skew. ■

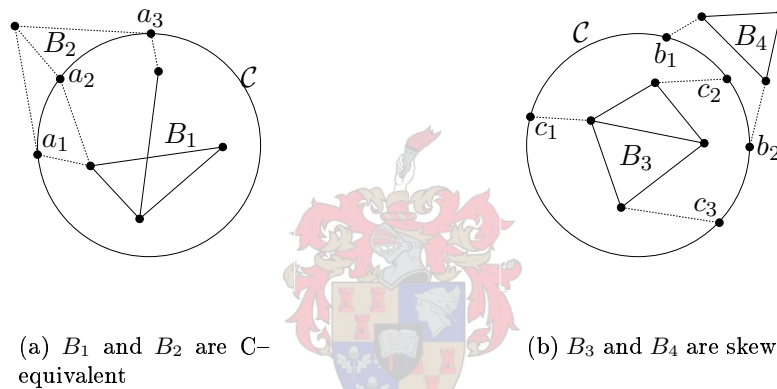


Figure A.3: The two types of  $C$ -component overlapping.

In Figure A.3(a) the  $C$ -components  $B_1$  and  $B_2$  are  $C$ -equivalent with the common clasp vertices  $a_1, a_2$  and  $a_3$  on  $C$ , whilst in Figure A.3(b) the  $C$ -components  $B_3$  and  $B_4$  are skew. The clasp vertices  $b_1$  and  $b_2$  of  $B_4$  are alternated by the clasp vertices  $c_2$  and  $c_3$  of  $B_3$  on  $C$ .

If two overlapping  $C$ -components are drawn on same side of the cycle, at least one line crossing results. This is illustrated in Figure A.4, where a pair of  $C$ -equivalent  $C$ -components are drawn in the exterior of the cycle in part (a) of the figure, and where a pair of skew  $C$ -components are drawn in the exterior of the cycle in part (b) of the figure. The  $C$ -components could have been drawn in the interiors of their respective cycles instead, without loss of generality. For both Figures A.4(a) and (b), it is assumed, without loss of generality, that the  $C$ -components labelled  $B_1$  were drawn first. In both drawings, the clasp edges of the  $C$ -components labelled  $B_2$  that could be drawn without the introduction of crossings were drawn as solid lines. The dashed edges emanating from the  $C$ -components labelled  $B_2$  in both figures indicate the possibilities for drawing the single remaining clasp edges in each of the two figures. Regardless of the choices, edge crossings are inevitable.

Two  $C$ -components  $B_1$  and  $B_2$  that only have two clasp vertices in common, and that are, in addition, not skew, may be drawn so that no crossings result. This is achieved by drawing (say)  $B_1$  in the interior of  $B_2$  (*i.e.*, such that  $B_1$  is drawn between the clasp edges of  $B_2$ ). For this reason, at least three common clasp vertices are required for overlapping.

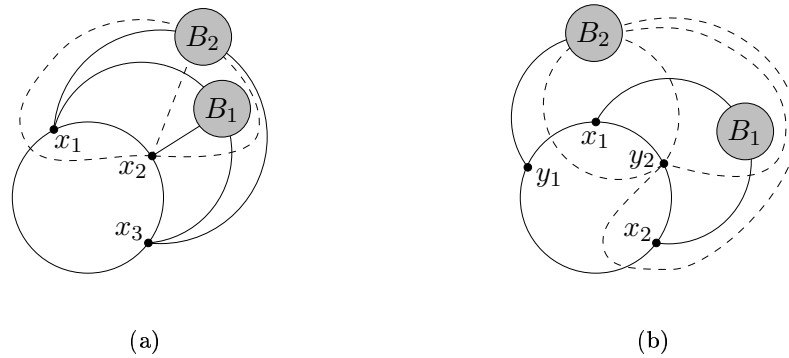


Figure A.4: Planar overlapping C-components that are drawn on the same side of a cycle cause at least one crossing.

### A.5 The overlap graph

The *overlap graph*<sup>1</sup>  $\mathcal{O}_C(\mathcal{G})$  of a graph  $\mathcal{G}$  with respect to a cycle  $\mathcal{C}$  in  $\mathcal{G}$  is an auxiliary graph of which the vertices represent C-components of  $\mathcal{C}$ , such that a pair of vertices is joined by an edge if the corresponding C-components overlap. An example of a C-component configuration of a graph  $\mathcal{G}$ , and its corresponding overlap graph may be seen in Figures A.5(a) and (b) respectively.

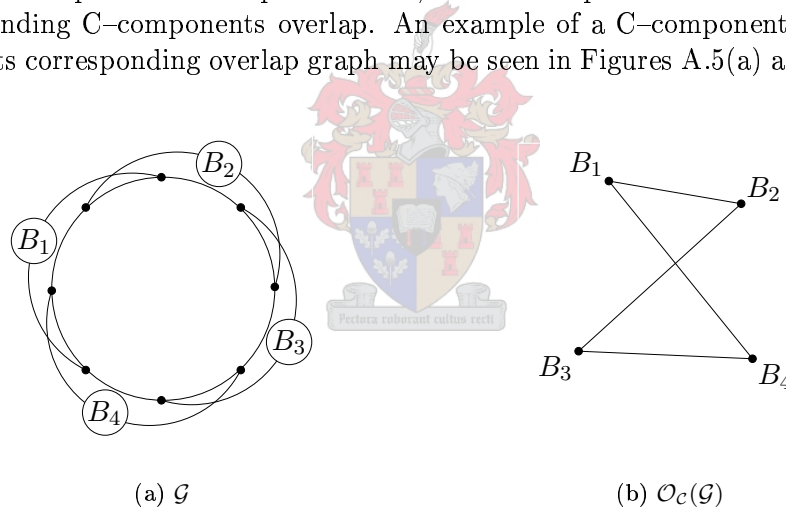


Figure A.5: A graph and its corresponding *overlap graph*.

An overlap graph corresponding to any cycle  $\mathcal{C}$  in  $\mathcal{G}$  will by necessity be non-bipartite, as shown by the following theorem. Note that the C-components corresponding to any cycle  $\mathcal{C}$  of  $\mathcal{G}$  are planar, due to the assumption that any subgraph of an order less than the order of  $\mathcal{G}$ , is planar.

**Theorem A.5.1** *If a graph  $\mathcal{G}$  is non-planar, then for any cycle  $\mathcal{C}$  of  $\mathcal{G}$  with the property that the C-components of  $\mathcal{C}$  are planar, the overlap graph  $\mathcal{O}_C(\mathcal{G})$  is non-bipartite.*

**Proof:** Consider the case where the overlap graph  $\mathcal{O}_C(\mathcal{G})$  for a non-planar graph  $\mathcal{G}$  with a cycle  $\mathcal{C}$  is bipartite, with a bipartition of the vertex set  $V(\mathcal{O}_C(\mathcal{G})) = A \cup B$ . Let  $\mathcal{C}$  be represented as a polygon in the plane. By drawing all of the C-components corresponding to vertices in  $A$ , say,

<sup>1</sup>As a point of interest, the definition of the overlap graph is quite similar to that of the *intersection graph* defined in § 6.2.1.

in the interior of the polygon, and all of the  $C$ -components of  $B$  in the exterior of the polygon, the  $C$ -components may be drawn such that no edges between any pair of  $C$ -components that are drawn on the same side of the polygon intersect, since the subgraphs (*i.e.*, the  $C$ -components) are themselves planar, and since  $C$ -components in the same partite set do not overlap. This constitutes a contradiction, since  $\mathcal{G}$  is assumed non-planar. Therefore  $\mathcal{O}_C(\mathcal{G})$  is not bipartite. ■

Because the overlap graph of a non-planar graph is not bipartite, it must contain a cycle of odd length (Theorem 2.1.1). The following theorem shows that an overlap graph of  $\mathcal{G}$  must, in fact, contain a cycle of length 3. This implies that  $\mathcal{G}$  contains three mutually overlapping  $C$ -components, and by the pigeonhole principle, two of the three overlapping  $C$ -components must be drawn on the same side of the cycle  $\mathcal{C}$ , which forces a crossing.

**Lemma A.5.1** *In a non-planar graph  $\mathcal{G}$ , and a cycle  $\mathcal{C}$  of  $\mathcal{G}$  with the property that all of the  $C$ -components of  $\mathcal{C}$  are planar, the minimal length of a cycle in  $\mathcal{O}_C(\mathcal{G})$  is 3.*

**Proof:** By Theorem A.5.1, it is known that  $\mathcal{O}_C(\mathcal{G})$  contains a cycle  $\mathcal{C}'$  of odd length. Let the vertices of  $\mathcal{C}'$  be labelled as  $B_0, B_1, \dots, B_{2n}$  (indices are expressed modulo  $2n + 1$ ), and let  $\mathcal{C}'$  be chosen to minimize  $n$ .

Suppose first that  $n \geq 2$ . Since  $n$  is minimal, a  $C$ -component  $B_i$  overlaps only the  $C$ -components  $B_{i-1}$  and  $B_{i+1}$  on  $\mathcal{C}'$ , and no other  $C$ -component  $B_j$ , where  $j \neq i - 1, i + 1$ . This is true, since otherwise a shorter cycle could be constructed by using the edge between the vertices corresponding to  $B_i$  and  $B_j$  in  $\mathcal{O}_C(\mathcal{G})$ , which would contradict the minimality of  $n$ . It is important to note that no pair  $B_i$  and  $B_{i+1}$  of overlapping  $C$ -components are  $C$ -equivalent, since then  $B_i$  would also overlap  $B_{i+2}$ , again allowing a shorter cycle to be constructed in  $\mathcal{O}_C(\mathcal{G})$ , thereby contradicting the minimality of  $n$ . Therefore, every pair of overlapping  $C$ -components  $B_i$  and  $B_{i+1}$  overlap in a skew fashion. This configuration is depicted in Figure A.6(a). Let  $x_1, x_2$  ( $y_1, y_2$ ) be clasp vertices of  $B_{i-1}$  ( $B_{i+1}$ ), such that the clasp vertices of  $B_i$  intersect the curve segments of the drawing of  $\mathcal{C}'$  between  $x_1$  and  $x_2$  (between  $y_1$  and  $y_2$ ). Since  $B_{i-1}$  and  $B_{i+1}$  do not overlap, let it be assumed that  $x_1, x_2, y_1, y_2$  occur in this order on  $\mathcal{C}$  (with the possibility that  $x_1 = y_2$  or  $x_2 = y_1$ ).

Let  $z_1$  ( $z_2$ ) be a clasp vertex of  $B_i$  in the curve segment of  $\mathcal{C}'$  between  $x_1$  and  $x_2$  ( $y_1$  and  $y_2$ ) such that  $z_1$  ( $z_2$ ) does not intersect  $x_1$  nor  $x_2$  ( $y_1$  nor  $y_2$ ). Let  $\mathcal{P}$  be a path in  $B_i$  with  $z_1$  and  $z_2$  as end-vertices. Since  $B_i$  does not overlap any  $C$ -component  $B_j$ , where  $j \neq i - 1, i + 1$ , it follows that the union of  $\mathcal{P}$  and the sub-path in  $\mathcal{C}'$  extending from  $z_2$  to  $z_1$  (in that cyclic order — the bold path in Figure A.6(b) which commences at the vertex  $z_2$  and is drawn clockwise around  $\mathcal{C}'$ , ending at the vertex  $z_1$ , represents  $\mathcal{P}$ ) is a cycle  $\mathcal{C}^*$  containing all the clasp vertices of every  $C$ -component  $B_j$ , with  $j \neq i - 1, i + 1$ . Let  $B'_i = \langle V(B_{i-1}) \cup \{x_2\} \cup \{y_1\} \cup V(B_{i+1}) \rangle$  such that  $x_1, z_1, z_2$  and  $y_2$  are its clasp vertices — in Figure A.6(c),  $B_i$  is placed on the interior of  $\mathcal{C}$ , and then in Figure A.6(d),  $B'_i$  is formed by coalescing  $B_{i-1}$  and  $B_{i+1}$ . Finally, it is shown in Figure A.6(e) that  $B_i$  may be transformed to a  $C$ -component  $B^*$  of the cycle  $\mathcal{C}^*$ , where the vertices of  $B_i$  that intersect  $\mathcal{C}^*$ , become the clasp vertices of  $B^*$ . Now  $B_0, \dots, B_{i-2}, B'_i, B_{i+2}, \dots, B_{2n}$  is an odd cycle of  $\mathcal{O}_C(\mathcal{G})$  of length  $2n - 1$ , contradicting the minimality of  $n$ .

Therefore  $n = 1$  and  $\mathcal{C}$  contains exactly 3 mutually overlapping  $C$ -components. ■

## A.6 The *coup-de-grâce*

Let  $\mathcal{C}$  be a cycle in  $\mathcal{G}$  — due to the fact that  $\mathcal{G}$  is 2-connected, it is guaranteed to contain a cycle.  $\mathcal{C}$  contains three mutually overlapping  $C$ -components, due to Lemma A.5.1. Let these



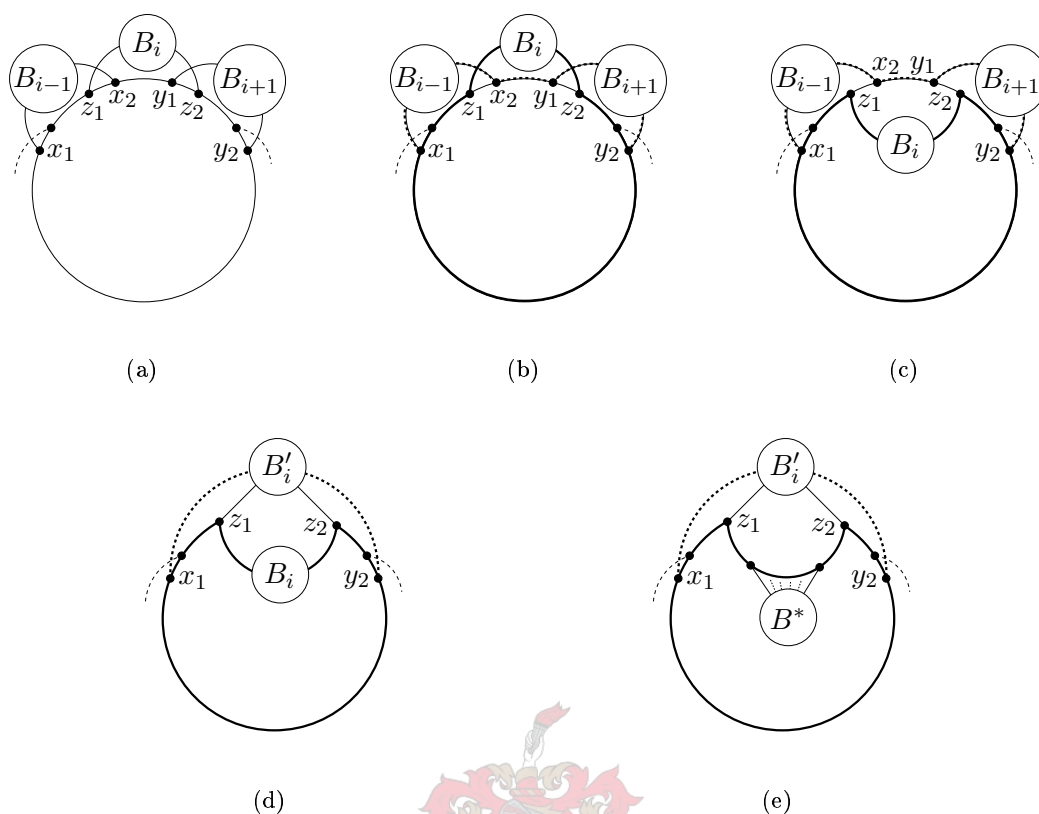


Figure A.6: Finding the minimal cycle length on which C-components may be attached.

C-components be labelled  $B_1, B_2$  and  $B_3$  respectively. The addition of any clasp vertices to one of these C-components cannot alter the fact that it overlaps another C-component (it may only transform the overlapping from a C-equivalent overlapping that is not skew, to an overlapping that is skew). Therefore, if all edge and vertex minimal cases of mutual overlapping between C-components are enumerated, all forbidden subgraphs which are present in non-planar graphs may be found. There are four cases, which are listed in Table A.1 for  $B_1, B_2$  and  $B_3$  (the order of the C-components is irrelevant, as a relabeling of the C-components will yield a desired order).

	$B_1B_2$	$B_2B_3$	$B_1B_3$
Case 1	C-equivalent, not skew	C-equivalent, not skew	C-equivalent, not skew
Case 2	C-equivalent, not skew	C-equivalent, not skew	Skew
Case 3	C-equivalent, not skew	Skew	Skew
Case 4	Skew	Skew	Skew

Table A.1: Overlap possibilities

### Case 1:

There is only one possibility, and this is illustrated in Figure A.7(a). Clearly, if any of the C-components were to have an extra clasp vertex on any position of the cycle, some overlappings would be rendered skew. A subdivision of  $K_{3,3}$  in may be found in Figure A.7(a) by selecting vertices  $\{a_1, a_2, a_3\}$  from the C-components  $B_1, B_2$  and  $B_3$  respectively as the first partite set and

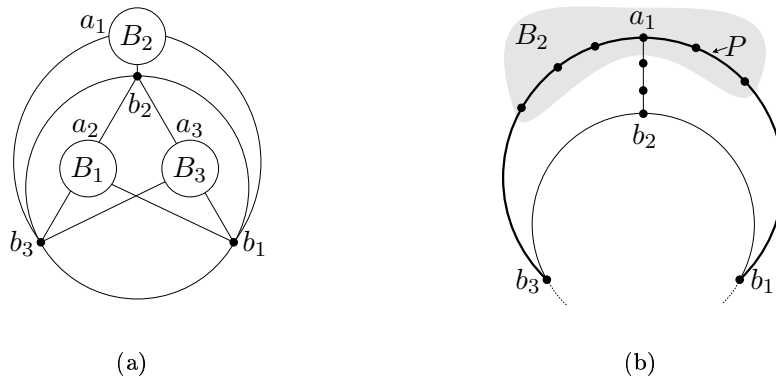


Figure A.7: Case 1.

by designating the vertices  $\{b_1, b_2, b_3\}$  as belonging to the second partite set. Some care should be taken in the selection of  $a_1, a_2$  and  $a_3$  from the C-components, since not every vertex in a C-component will be sufficient. Consider the selection of the vertex  $a_1$  from the C-component  $B_2$ . Firstly, there must be a path from the clasp vertex  $b_1$  to the clasp vertex  $b_3$  that passes through  $B_2$  — this is guaranteed, since the C-component is connected and since both  $b_1$  and  $b_3$  are adjacent to vertices in  $B_2$ . Such a path, labelled  $P$ , is shown in bold in Figure A.7(b). Finally, there must be a path from the clasp vertex  $b_2$  to a vertex in  $P$ , due to the fact that  $B_2$  is connected. Let this vertex in  $P$  be chosen as  $a_1$ , as shown in Figure A.7(b). The vertices  $a_2$  and  $a_3$  are chosen similarly.

**Case 2:**

It is not possible to construct the structure corresponding to this case, since the only way to do so, would be to commence with Case 1, and to add an extra clasp vertex for one of the C-components. This would, however, render the overlappings of two pairs of C-components skew, resulting in Case 3.

**Case 3:**

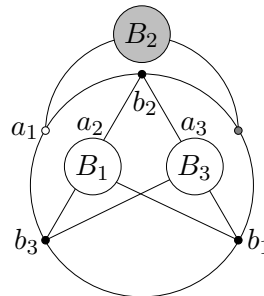


Figure A.8: Case 3.

The only minimal construction that exists for this case, is shown in Figure A.8. The addition of clasp vertices to  $B_2$  would not affect its overlapping status with respect to the other C-components, as it already overlaps both of the other C-components in a skew fashion. However,

the addition of a clasp vertex to either  $B_1$  or to  $B_3$  would render their overlapping skew. In this construction, a subdivision of  $K_{3,3}$  may be found, by selecting the vertices  $\{a_1, a_2, a_3\}$  for the first partite set and by selecting the vertices  $\{b_1, b_2, b_3\}$  for the second partite set. The vertices  $a_2$  and  $a_3$  are selected according to the method described in Case 1.

**Case 4:**

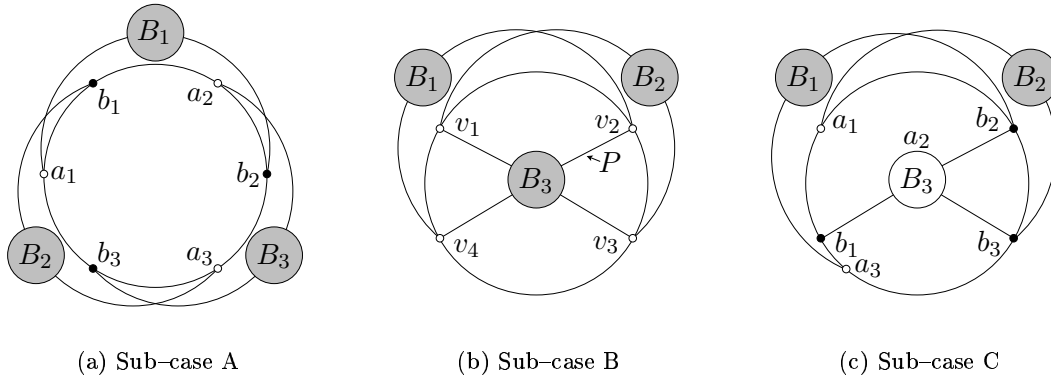


Figure A.9: Case 4.

The three sub-cases shown in Figure A.9 represent all the minimal constructions for purely skew overlappings. It is shown in each of the following sub-cases that these constructions are indeed minimal.

**Sub-case A:** The removal of a clasp edge from a C-component  $B$ , would leave  $B$  with only a single clasp edge. This violates the assumption of 2-connectivity of  $\mathcal{G}$ . The construction is therefore minimal with respect to the number of vertices and edges used. Since all overlappings are skew, the addition of other clasp vertices will have no effect on the overlapping configurations. A subdivision of  $\mathcal{K}_{3,3}$  may be obtained in this construction by letting the vertices  $a_1, a_2$  and  $a_3$  be members to the first partite set, whilst letting the vertices  $b_1, b_2$  and  $b_3$  be members to the second partite set.

**Sub-case B:** At least four clasp vertices are required to enable the possibility of a skew overlapping between a pair of C-components. Therefore, the configuration in Figure A.9(b) is minimal with respect to the number of vertices used. The C-components  $B_1$  and  $B_2$  have the minimum number of edges. It may be verified that the removal of a clasp edge from  $B_3$  will result in  $B_3$  not being overlapped by one of the other two C-components.

It may be seen from Figure A.9(b) that there must be path  $P$  from the clasp vertex  $v_2$  to the clasp vertex  $v_4$  that passes through the C-component  $B_3$  — this is guaranteed, due to the fact that  $B_3$  is connected. There is also a path  $p_1$  that joins the clasp vertex  $v_1$  to a vertex in  $P$  and a path  $p_2$  that joins the clasp vertex  $v_3$  to a vertex in  $P$  — note that all vertices of  $P$ , except for its end-vertices  $v_2$  and  $v_4$ , are part of the C-component  $B_3$ . Now there are two sub-sub-cases to consider:

- i. If both  $p_1$  and  $p_2$  join  $P$  at a common vertex, as shown in Figure A.10(a), a subdivision of  $\mathcal{K}_5$  may be found by selecting all of the black vertices, labelled  $a_1, a_2, a_3, a_4$  and  $a_5$ , where  $a_5$  is the common vertex described above.

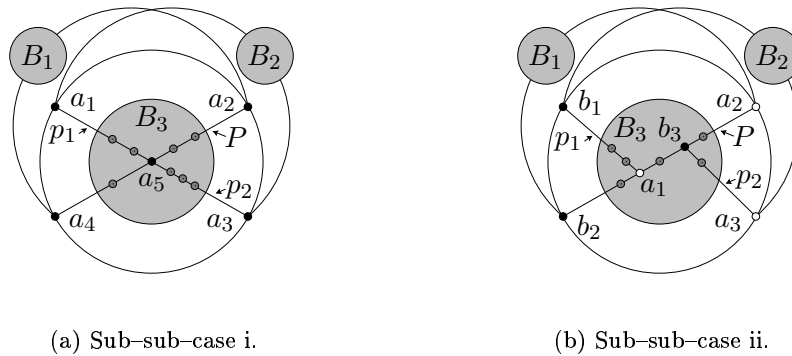


Figure A.10: Sub-case B.

- ii. If  $p_1$  and  $p_2$  join  $P$  at different vertices, as shown in Figure A.10(b), a subdivision of  $\mathcal{K}_{3,3}$  may be found by letting the white vertices,  $a_1, a_2$  and  $a_3$  belong to one partite set, and by letting the black vertices,  $b_1, b_2$  and  $b_3$  belong to the other partite set. The vertices  $a_1$  and  $b_3$  are the vertices at which  $p_1$  and  $p_2$  join  $P$ , respectively.

**Sub-case C:** The configuration in Figure A.9(c) results if a clasp edge of either  $B_1$  or of  $B_2$  is “split off” and moved away from a clasp vertex shared with  $B_3$ . In this case, without loss of generality, the clasp vertex labelled  $a_3$  of  $B_1$  was “split off” and moved away from the clasp vertex labelled  $b_1$ . The clasp edge of  $B_3$  which was attached to the vertex  $a_1$  in sub-case B has no impact on the overlapping structure of the three C-components, and has therefore been deleted from this configuration. A subdivision of  $\mathcal{K}_{3,3}$  may be discerned by letting the placing the vertices  $a_1, a_2$  and  $a_3$  in the first partite set, and by placing the vertices  $b_1, b_2$  and  $b_3$  in the second partite set.

It is noted that in sub-case C, it is irrelevant whether or not the clasp edge of  $B_2$ , which is attached to the clasp vertex  $b_3$ , “splits off” and is moved away from  $b_3$  (since the overlapping configurations will remain unchanged). In fact, the only reason that there is a clasp edge of  $B_3$  attached to  $b_3$ , is to ensure that  $B_3$  overlaps  $B_1$ . Should the clasp vertex of  $B_1$  be “split off” and moved away from  $b_2$  in an anti-clockwise direction, the clasp edge of  $B_3$  attached to  $b_2$  would cause  $B_3$  to overlap  $B_1$  in a skew fashion, in which case the clasp edge attached to  $b_3$  would be obviated. But if this happens, one arrives at sub-case A. Therefore, all minimal cases for Case 4 have been enumerated. ■

## Appendix B

# Computer Implementations

The purpose of computing is insight, not numbers!  
— *Richard Hamming (1915–1998)*

The purpose of this appendix is to provide the reader with access to the source code that was used in the implementations of the algorithms described in Chapters 5 and 6. A mixture of the programming languages Python, C and C++ was used in the implementations.

A number of the heuristic optimization techniques were first prototyped in Python and later reprogrammed in C. Algorithms for which execution speed was not a critical factor were all programmed only in Python. The overall framework for the software was programmed in Python.

For Python programming, version 2.3 of the *CPython* [Pyt] interpreter was used. Programs written in C and in C++ were compiled using version 3.3.2 of *Gnu Compiler Collection* [GCC]. The author found the software tool, *SWIG 1.3* [SWI], invaluable for constructing modules that allowed the CPython interpreter to access the compiled C and C++ code. Version 4.0 of the *LEDA* [LED] library was used for the GUI component of the software and its planarity testing procedures were used in the implementation of the Garey-Johnson algorithm.

A large number of the implementations in this chapter have been kept as faithful in form as possible to the pseudo-code of the algorithms they implement. Where possible, the source codes have been annotated with line numbers from the pseudo-code algorithms, to indicate how the pseudo-code constructs map to constructs in C, C++ or Python, so as to facilitate understanding of the implementations. The line number annotations were encoded as source comments for the various languages.

A line number annotation in both C and C++ has the form `/*n*/`, where `n` is the line number. These annotations were inserted into the left margins of C/C++ constructs that relate to pseudo-code constructs. Where margin space was insufficient, the line numbers were inserted into new lines directly above the constructs they referenced. A Python line number annotation has the form `# n` and all Python line number annotations were inserted as left justified lines, directly above the constructs they referenced.

### B.1 Edge layout heuristics

In all of the edge layout algorithms, `cr_G` is an intersection graph and the array `page` maps vertices of `cr_G` to page numbers. Every algorithm in this section modifies `page` to the layout solution, but the graph `cr_G` is never modified.

### B.1.1 The GreedySide algorithm

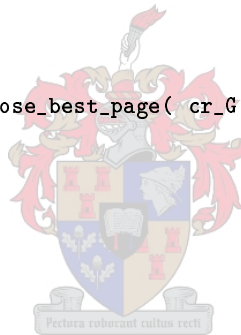
These algorithms are implementations of Algorithm 6.3 (**GreedySide**), that is described in § 6.2.1.1. Due to the almost one-to-one mapping of the pseudo-code of Algorithm 6.3 to the implementations, the reader is referred to § 6.2.1.1 for a description of the algorithm.

#### Python implementation

```
def greedy_side( cr_G, page ):
    iterations = 0
# 1
    stabalised = False

    while not stabalised:
# 3
        no_crossings = 0
# 4
        stabalised = True
        iterations += 1

# 5
    for cr_v in cr_G.vertices():
# 6
        current_page = page[ cr_v ]
# 7
        val, new_page = c_algorithms.choose_best_page( cr_G, cr_v )
# 8
        if new_page != current_page:
            stabalised = False
            no_crossings += val
# 13
    return no_crossings / 2
```



#### C implementation

```
int
greedy_side( graph *cr_G, GArray *page )
{
    int no_crossings = 0;
/*1*/
    bool stabalised = false;

/*2*/
    while ( !stabalised )
    {
        vertices_itr v_itr = graph_vertices( cr_G );
        vertex *cr_v;
/*3*/ no_crossings = 0;
/*4*/ stabalised = true;
        int new_page;

/*5*/
        while ( ( cr_v = vertices_itr_next( &v_itr ) ) != NULL )
        {
/*6*/     int current_page = g_array_index( page, int, idx( cr_v ) );
/*7*/     no_crossings += choose_best_page( &new_page, page, cr_G, cr_v );

/*8*/     if ( new_page != current_page )
```

```

/*9*/     stabilised = false;
        }
    }

/*13*/
    return no_crossings / 2;
}

```

### B.1.2 The Cimikowski-Shope neural network algorithm

Implementations for Algorithm 6.4 and Algorithm 6.5 (Cimikowski-Shope) are given in this section. The reader is referred to § 6.2.1.2 for a description of the algorithm. The algorithm `neural_crossing_ll` (Algorithm 6.4) was employed by both the Python and C versions of `neural_layout` (Algorithm 6.5), due to the fact that it is an often executed loop, and executes in  $O(|E(\mathcal{G})|)$  time. The reader is referred to § 6.2.1.2 for descriptions of the two algorithms.

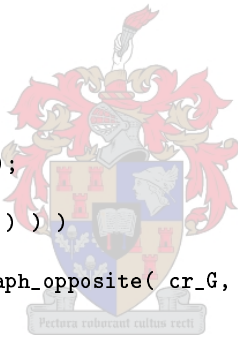
#### C implementation of `neural_crossing`

```

int
neural_crossing_ll( int *page, graph *cr_G, vertex *cr_v )
{
/*1*/
    int total_crossings = 0;
    edges_itr e_itr;
    edge *e;

    e_itr = graph_out_edges( cr_G, cr_v );
/*2*/
    while ( ( e = edges_itr_next( &e_itr ) ) )
    {
/*3*/ total_crossings += page[ idx( graph_opposite( cr_G, e, cr_v ) ) ];
    }
/*4*/
    return total_crossings;
}

```



#### Python implementation of `neural_layout`

```

def neural_layout( cr_G, page, A, B, C, dt, iterations, min_rand ):
    U_up = {}
    U_down = {}
# 1
    for v in cr_G.vertices():
# 2
        U_up[ v ] = random.uniform( min_rand, 0 )
# 3
        U_down[ v ] = random.uniform( min_rand, 0 )

    V_up = c_data_types.GVertexArrayInt( cr_G.num_vertices() )
    V_down = c_data_types.GVertexArrayInt( cr_G.num_vertices() )

    t = 0
# 5
    equilibrium = False
# 6
    while not equilibrium and t < iterations:
# 7

```

```

    equilibrium = True
# 8
    t += 1
# 9
    for cr_v in cr_G.vertices():
# 10
        if U_up[ cr_v ] > 0.0:
# 11
            V_up[ cr_v ] = 1
            else:
# 13
                V_up[ cr_v ] = 0

# 15
        if U_down[ cr_v ] > 0.0:
# 16
            V_down[ cr_v ] = 1
            else:
# 18
                V_down[ cr_v ] = 0

# 22
    for cr_v in cr_G.vertices():
# 23
        if V_up[ cr_v ] == V_down[ cr_v ]:
# 24
            equilibrium = False

    # compute the hill climbing constant C
    if V_up[ cr_v ] == 0 and V_down[ cr_v ] == 0:
        hill = C
    else:
        hill = 0

    a_factor = V_up[ cr_v ] + V_down[ cr_v ] - 1

    sum_V_up = c_algorithms.neural_crossing( V_up, cr_G, cr_v )
    sum_V_down = c_algorithms.neural_crossing( V_down, cr_G, cr_v )

    common = -A * a_factor + hill

# 27
    U_up[ cr_v ] += ( common + B * ( -sum_V_up + sum_V_down ) ) * dt
# 28
    U_down[ cr_v ] += ( common + B * ( sum_V_up - sum_V_down ) ) * dt

# 30
    if abs( U_up[ cr_v ] ) > 1:
        U_up[ cr_v ] /= abs( U_up[ cr_v ] )

# 31
    if abs( U_down[ cr_v ] ) > 1:
        U_down[ cr_v ] /= abs( U_down[ cr_v ] )

    return equilibrium

```

### C implementation of neural\_layout

```

bool
neural_layout( graph *cr_G

```



```

        , GVertexArrayInt *page
        , double A
        , double B
        , double C
        , double dt
        , int iterations
        , double min_rand )
{
    double U_up[ graph_num_vertices( cr_G ) ];
    double U_down[ graph_num_vertices( cr_G ) ];
    int V_up[ graph_num_vertices( cr_G ) ];
    int V_down[ graph_num_vertices( cr_G ) ];

    int t = 0;
    bool equilibrium = false;

    vertices_itr v_itr;
    vertex *cr_v;
    int cr_i;

    if ( !neural_preconditions( page, A, B, C, dt, iterations, min_rand ) ) return -2;

    RANDOMISE();
/*1*/
    for ( int i = 0; i < graph_num_vertices( cr_G ); i++ )
        {
/*2*/ U_up[ i ] = UNIFORM( min_rand, 0 );
/*3*/ U_down[ i ] = UNIFORM( min_rand, 0 );
        }

/*6*/
    while ( !equilibrium && t < iterations )
        {
/*7*/ equilibrium = true;
/*8*/ t += 1;

        v_itr = graph_vertices( cr_G );
/*9*/ while ( ( cr_v = vertices_itr_next( &v_itr ) ) != NULL )
            {
                cr_i = idx( cr_v );

/*10*/ if ( U_up[ cr_i ] > 0.0 )
/*11*/     V_up[ cr_i ] = 1;
/*12*/ else
/*13*/     V_up[ cr_i ] = 0;

/*15*/ if ( U_down[ cr_i ] > 0.0 )
/*16*/     V_down[ cr_i ] = 1;
/*17*/ else
/*18*/     V_down[ cr_i ] = 0;
            }

        v_itr = graph_vertices( cr_G );
/*22*/
        while ( ( cr_v = vertices_itr_next( &v_itr ) ) != NULL )
            {
                double hill;
                double sum_V_up, sum_V_down;
                double common;

                cr_i = idx( cr_v );

```

```

/*23*/   if ( V_up[ cr_i ] == V_down[ cr_i ] )
/*24*/       equilibrium = false;

        if ( ( V_up[ cr_i ] == 0 ) && ( V_down[ cr_i ] == 0 ) )
            hill = C;
        else
            hill = 0.0;

        common = -A * ( V_up[ cr_i ] + V_down[ cr_i ] - 1 ) + hill;

        sum_V_up   = neural_crossing_ll( V_up, cr_G, cr_v );
        sum_V_down = neural_crossing_ll( V_down, cr_G, cr_v );

/*27*/   U_up[ cr_i ]   += ( common + B * ( -sum_V_up + sum_V_down ) ) * dt;
/*28*/   U_down[ cr_i ] += ( common + B * ( sum_V_up - sum_V_down ) ) * dt;

/*30*/   if ( fabs( U_up[ cr_i ] ) > 1.0 )
/*30*/       U_up[ cr_i ] /= fabs( U_up[ cr_i ] );

/*31*/   if ( fabs( U_down[ cr_i ] ) > 1.0 )
/*31*/       U_down[ cr_i ] /= fabs( U_down[ cr_i ] );
    }
}

return equilibrium;
}

```

### B.1.3 The genetic algorithm

The genetic algorithm code uses a small library, which was programmed to ease the implementation of such algorithms. For a new problem instance, functions for mutation, crossover, selection and basic population control (*i.e.*, creation of new genes and enumeration of genes) have to be provided. The functions for the most important of these concepts, namely, mutation, crossover and selection are discussed in this section.

#### B.1.3.1 The selection mechanism

The selection code in this section implements the selection behaviour discussed in § 6.2.1.3. In a population  $P$  where  $n$  individual chromosomes compete per tournament, a total of  $n \times |P|$  individuals will be selected from  $|P|$  during the course of the selection process. Since the population is enumerated in a sequential order, the selection will “wrap around” a number of times, since there are only  $|P|$  individuals. When the “wrapping around” occurs, the order of the individuals in the population is shuffled.

```

template < typename Population, typename Compare >
struct tournament_select
{
    typedef typename population_traits< Population >::chromosome_iterator chromosome_iterator;

    tournament_select ( Compare const& _cmp, int _no_contestants )
        : cmp( _cmp )
        , no_contestants( _no_contestants ) {}

    template < typename Time > inline chromosome_iterator
    operator() ( Population& p, Time iteration )

```

```

{
    typedef typename population_traits<Population>::chromosome_iterator    chromosome_iterator;
    typedef typename iterator_traits<chromosome_iterator>::iterator_category iterator_category;

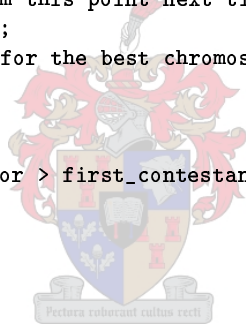
    // Create circular iterators that will wrap around upon
    // completion of enumeration of p
    circular_iterator< chromosome_iterator > last_contestant;
    circular_iterator< chromosome_iterator > ret_value;

    // Set the end iterator equal to the beginning iterator
    last_contestant = first_contestant;
    // For the number of competitors
    for ( int i = 0; i < no_contestants; i++ )
    {
        ++last_contestant; // advance the last contestant
        // When the population has been enumerated entirely, perform a
        // random shuffle on it
        if ( last_contestant.base() == begin_iterator( p ) )
            detail::shuffle_population( p, iterator_category() );
    }

    // Determine the fittest element in the set of contestants
    ret_value = std::min_element( first_contestant, last_contestant, cmp );
    // Shift first_contestant to the position of last_contestant so
    // that selection will proceed from this point next time around
    first_contestant = last_contestant;
    // Return the population iterator for the best chromosome
    return ret_value.base();
}

circular_iterator< chromosome_iterator > first_contestant;
Compare const& cmp;
int no_contestants;
};

```



### B.1.3.2 The mutation mechanism

The algorithm object `stl_bit_mutate_t`, operates on any C++ STL-like container, and assumes that the contents of the container are zeros or ones. It flips the bit of every element with a probability of `mutation_chance`.

```

struct stl_bit_mutate_t
    : public default_mutator
{
    stl_bit_mutate_t( double _mutation_chance )
        : mutation_chance( _mutation_chance ) {}

    template < typename Population, typename Time >
    inline bool
    operator () ( Population& p
                , typename population_traits< Population >::chromosome_iterator child
                , const Time iteration )
    {
        typedef typename population_traits<Population>::chromosome_descriptor chromosome_descriptor;
        typedef typename chromosome_traits< chromosome_descriptor >::gene_iterator c_itr;
        bool has_mutated = false;

        // For each bit in the child chromosome...
    }
};

```

```

for (c_itr = begin_iterator(genes(*child)); c_itr != end_iterator(genes(*child)); ++c_itr)
{
    // If the selected random number is smaller than the probability value
    if ( ( (double) random() / (double) RAND_MAX ) < mutation_chance )
    {
        // flip the corresponding bit
        *c_itr = ( *c_itr > 0 ) ? 0 : 1;
        // and set the mutation flag, which will be returned
        has_mutated = true;
    }
}

return has_mutated;
}

double mutation_chance;
};

```

### B.1.3.3 The crossover mechanism

The crossover code in this section employs the circular encoding described in § 6.2.1.3. There is one important difference — in § 6.2.1.3, the edge layout data for edges that are grouped together into a cell are stored within the cell, but in this implementation, the cell entries corresponding to the edges contain indices into an array that stores the actual edge layout data for the edges, or more specifically, the layout data for the crossing-vertices corresponding to the edges.

The rationale behind this scheme is that the edge layout data for the crossing-vertices of the intersection graph is normally stored in the array `page`. Thus, if the edge layout data is stored into `page`, the existing data structures may be used transparently with the genetic algorithm code. For example, the mutation code from the previous section also expects to receive a sequence, and it is very easy to provide the additional programming infrastructure to facilitate this.

The type that implements the circular encoding is called `mate_template`. The function `build_templates` constructs an instance of this type that corresponds to a vertex arrangement of the input graph  $G$ . The type `e_map` is a mapping from the edges of  $G$  to the crossing-vertices of the corresponding intersection graph `cr_G` of  $G$ .

In this scheme, `mt[ i ]` provides a list of indices corresponding to cell  $i$  in the circular encoding. For some list index  $j$ , `mt[ i ][ j ]` is an index into the array `page`, corresponding to one of the edges in cell  $i$  (or more correctly, one of the crossing vertices in cell  $i$ ). The edge to which it corresponds is irrelevant, since it only matters that all edges within the same cell index are treated as a unit in the crossover operation.

Note that in the code, not only the index of an edge is stored in the cell, but also the distance between its incident vertices on the circle (of the circular encoding). This was only done for debugging purposes and has no influence on the algorithm.

The crossover is performed by the algorithm object `mate_template_cross`. The two crossover points on the circle are selected first. In the first `for` loop, the relevant genes from the first parent are copied into the child. This process is repeated for the genes of the second parent in the second `for` loop.

The list `straddling_edges` is constructed from all edges that straddle the crossover points. This list is used by the local optimizer which is the last of the loops in the function. It can easily be seen that this is an implementation of the GreedySide algorithm.

```

void
build_templates( mate_template& mt
, graph *G
, const GArray *spine_inv
, const GArray *e_map )
{
edge *e;
int *spine_inv_ll = (int*) spine_inv->data;
vertex **e_map_ll = (vertex**) e_map->data;

mt.resize( 2 * graph_num_vertices( G ) );

FORALL_EDGES( G, e )
{
// Is there a crossing-vertex for this edge?
// When e_map_ll[ idx( e ) ] == NULL, it means that the corresponding
// crossing-vertex was isolated in cr_G, and thus deleted.
if ( e_map_ll[ idx( e ) ] != NULL )
{
int spine_idx = 0;
int edge_dist = 0;
// src_idx = index of source( e ) on the spine (circle)
int src_idx = spine_inv_ll[ idx( graph_source( G, e ) ) ];
// tgt_idx = index of target( e ) on the spine (circle)
int tgt_idx = spine_inv_ll[ idx( graph_target( G, e ) ) ];
// Ensure that src_idx < tgt_idx by swapping if necessary
if ( src_idx > tgt_idx )
{
std::swap( src_idx, tgt_idx );
}

// Get the distances between the two indices around the circle
int dist_1 = mod( tgt_idx - src_idx, graph_num_vertices( G ) );
int dist_2 = mod( src_idx - tgt_idx, graph_num_vertices( G ) );

// Let spine_idx be the index in the middle of the shortest space
// between the two indices
if ( dist_1 < dist_2 )
{
spine_idx = mod( 2 * src_idx + dist_1, 2 * graph_num_vertices( G ) );
edge_dist = dist_1;
}

else
{
spine_idx = mod( 2 * tgt_idx + dist_2, 2 * graph_num_vertices( G ) );
edge_dist = dist_2;
}

// Add the index of the crossing-vertex corresponding to e to
// the cell. The distance between e's incident vertices is
// also stored in the cell, but this is only used for debugging purposes.
mt[ spine_idx ].push_back( vertex_int_pair( e_map_ll[ idx( e ) ], edge_dist ) );
}
} ENDFOR;
}

struct mate_template_cross
{
mate_template_cross ( mate_template& _mt, graph *cr_G, bool local_optimize )
: mt( _mt )

```

```

{
    this->cr_G = cr_G;
    this->local_optimize = local_optimize;
}

template < typename Time >
inline bool
operator () ( EdgePopulation& p
            , typename EdgePopulation::iterator parent1
            , typename EdgePopulation::iterator parent2
            , typename EdgePopulation::iterator child
            , Time iteration )
{
    int pos1 = UNIFORM_INT( 0, mt.size() );
    int pos2 = mod( pos1 + mt.size() / 2, mt.size() );
    std::vector< vertex* > straddling_edges;

    GArray* p1g = genes( *parent1 );
    GArray* p2g = genes( *parent2 );
    GArray* cg = genes( *child );
    vertex_int_vec::const_iterator itr;

    for ( int i = pos1; i != pos2; i = ( i + 1 ) % mt.size() )
    {
        for ( itr = mt[ i ].begin(); itr != mt[ i ].end(); ++itr )
        {
            ((int*) (cg->data))[ idx( itr->v ) ] = ((int*) (p1g->data))[ idx( itr->v ) ];
            if ( i - pos1 + itr->i > mod( pos2 - pos1, mt.size() )
                || i - pos1 - itr->i < 0 )
            {
                straddling_edges.push_back( itr->v );
            }
        }
    }

    for ( int i = pos2; i != pos1; i = ( i + 1 ) % mt.size() )
    {
        for ( itr = mt[ i ].begin(); itr != mt[ i ].end(); ++itr )
        {
            ((int*) (cg->data))[ idx( itr->v ) ] = ((int*) (p2g->data))[ idx( itr->v ) ];
            if ( i - pos2 + itr->i > mod( pos1 - pos2, mt.size() )
                || i - pos2 - itr->i < 0 )
            {
                straddling_edges.push_back( itr->v );
            }
        }
    }

    if ( local_optimize )
    {
        bool stabilised = false;
        /* This algorithm is adapted from the GreedySide algorithm */
        while ( !stabilised )
        {
            stabilised = true;
            int new_page;

            std::vector< vertex* >::const_iterator itr;
            for ( itr = straddling_edges.begin(); itr != straddling_edges.end(); ++itr )
            {
                int current_page = ((int*) (cg->data))[ idx( *itr ) ];

```

```

        choose_best_page( &new_page, cg, cr_G, *itr );

        if ( new_page != current_page )
            stabilised = false;
    }
}

return true;
}

mate_template& mt;
graph *cr_G;
bool local_optimize;
};

```

#### B.1.3.4 The genetic algorithm driver

This algorithm is a “driver” in the sense that it steps through the various generations, applying the functions for crossovers, mutation and selection at the right positions in the code. The algorithm is modelled on the visitor design pattern. That is, it invokes particular methods of a “visitor” (which is an algorithm object) at well defined points in the code. In the genetic context, these points correspond to the selection of parents, crossovers, mutation and the beginnings and ends of the generations. Each of the components that are provided as parameters may provide optional “callbacks,” that are executed by a visitor. These callbacks provide information regarding the points at which they have to be invoked.

The types generated by `sga_callback< T >::type`, where `T` is a type, are the callback types. If a callback is not defined for a type, `sga_callback< T >::type` simply generates a basic type that performs no actions. The callbacks from the various components are obtained by the call `get_callback`. This call returns the same basic type as described above, if no callback is defined for the type of object to which `get_callback` is applied. The callbacks are finally combined into a list, which is passed to the a visitor.

The algorithm allows the user to provide a visitor in the parameter list. This visitor is combined with the visitor constructed from the callbacks of the other components. The final combined visitor is called “`vis`” in the code.

To aid efficiency, the algorithm was designed to alternate between two populations of equal size as it proceeds through generations. For two populations *A* and *B*, if it selects parents from *A* in a generation, then the offspring are copied into *B*. For the next generation, parents are selected from *B*, and offspring are copied into *A*, and so forth.

```

template < typename Population
        , typename Select
        , typename Crosser
        , typename Mutator
        , typename Continuator
        , typename Visitor >
void
simple_en_bloc_genetic_algorithm
( Population& population
  , Select select
  , Crosser cross
  , Mutator mutate
  , Continuator continue_

```

```

, Visitor user_vis )
{
// Some typedefs to make the code easier to read.
typedef typename population_traits< Population >::chromosome_iterator  chromosome_iterator;
typedef typename population_traits< Population >::chromosome_descriptor chromosome;

// Types may have callback associated with them, which need to be invoked at certain
// points in the algorithm to ensure that they work properly
typedef typename sga_callback< Population >::type  PopulationVisitor;
typedef typename sga_callback< Select >::type      SelectVisitor;
typedef typename sga_callback< Crosser >::type     CrosserVisitor;
typedef typename sga_callback< Mutator >::type     MutatorVisitor;
typedef typename sga_callback< Continuator >::type ContinuatorVisitor;

// This is a list of visitors that are invoked at various stages.
// They're all obtained as callbacks from
typedef typename list5< PopulationVisitor
, SelectVisitor
, CrosserVisitor
, MutatorVisitor
, ContinuatorVisitor >::type VisitorList;

// Var definitions
sga_variables< Population, Select, Select, Crosser, Mutator, Continuator >
variables( population, select, select, cross, mutate, continue_ );
int iteration = 0;

PopulationVisitor  population_visitor( get_callback( population ) );
SelectVisitor      select_visitor( get_callback( select ) );
CrosserVisitor     crosser_visitor( get_callback( cross ) );
MutatorVisitor     mutator_visitor( get_callback( mutate ) );
ContinuatorVisitor continuator_visitor( get_callback( continue_ ) );

VisitorList algo_visitor_list
= make_list( population_visitor, select_visitor, crosser_visitor
, mutator_visitor, continuator_visitor );

// Combine the callbacks, along with the user-supplied visitor
// into a single visitor
combine_visitors< Visitor, spga_visitor< VisitorList > >
vis( user_vis, make_spga_visitor( algo_visitor_list ) );
Population auxiliary_population;
copy_population( population, auxiliary_population );

while ( continue_( variables, iteration ) )
{
// At each new era, it might be necessary to update the
// components of the algorithm. For example, the mutation
// rate may be pushed up if the population doesn't change
// much etc etc.
vis.new_era( variables, iteration );

// Select two chromosome from the population to mate
chromosome_iterator offspring = begin_iterator( auxiliary_population );

while ( offspring != end_iterator( auxiliary_population ) )
{
// Select two parent chromosomes
variables.parent1 = select( population, iteration );
variables.parent2 = select( population, iteration );
// Notify any visitors that we have selected our parents

```



```

vis.choose_parent( variables, iteration );
vis.choose_parent( variables, iteration );

for ( int i = 0; i < 2; ++i )
{
    variables.child = offspring;
    // Cross chromosomes for new offspring
    if ( cross( population, variables.parent1, variables.parent2
               , variables.child, iteration ) )
    {
        invalidate( *variables.child );
        vis.cross( variables, iteration );
    }

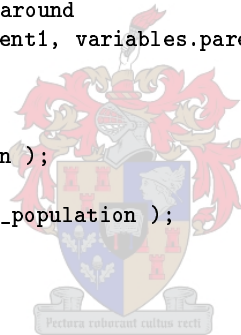
    // Mutate the offspring
    if ( mutate( population, variables.child, iteration ) )
    {
        invalidate( *variables.child );
        vis.mutate( variables, iteration );
    }

    vis.new_child( variables, iteration );
    ++offspring;
    if ( offspring == end_iterator( auxiliary_population ) ) break;
    // Swap the two parents around
    std::swap( variables.parent1, variables.parent2 );
}
}

vis.end_era( variables, iteration );

std::swap( population, auxiliary_population );
++iteration;
}
}

```



## B.2 Vertex arrangement heuristics

The algorithms in this section are concerned with the placement of vertices of a graph in the spine of a book, so as to minimize the number of crossings in the resultant graph drawing.

### B.2.1 Preconditioning algorithms

These algorithms are employed by the tabu search algorithm to precondition solutions, so as to accelerate convergence.

#### B.2.1.1 Nicholson's heuristic

The algorithm provided in this section is an implementation of the first part of Nicholson's heuristic (§ 4.3.2.2). It constructs a vertex arrangement by placing the vertices of the graph  $G$  onto the spine sequentially. The first vertex to be placed, is the vertex with the highest degree in  $G$ . For each new placement, the algorithm selects an unplaced vertex with the highest number of adjacent vertices already on the spine. This vertex is then placed into a position that minimizes the number of edge crossings caused by its addition.

```

int
choose_best_pages_for_adjacent_edges( graph *G
                                     , graph *cr_G
                                     , GEdgeArrayVertex *e_map
                                     , vertex *v
                                     , GVertexArrayInt *page )
{
    int total_crossings = 0;
    edges_itr e_itr = graph_out_edges( G, v );
    edge *e;
    int page_no;

    while ( ( e = edges_itr_next( &e_itr ) ) )
    {
        vertex *cr_v = g_array_index( e_map, vertex*, idx( e ) );

        if ( vertex_exists( cr_v ) )
            total_crossings += choose_best_page( &page_no, page, cr_G, cr_v );
    }
    return total_crossings;
}

vertex*
remove_vertex_with_highest_spine_connectivity( graph *G, int *v_selected, GArray *v_list )
{
    int max_i = 0;
    int max_degree = 0;
    int i;
    vertex *v;

    /* For all vertex list indices */
    for ( i = 0; i < v_list->len; i++ )
    {
        /* Let v be the vertex at index i */
        v = ((vertex**) v_list->data)[ i ];
        vertex *u;
        /* Get an iterator for the vertices adjacent to v */
        adjacency_itr a_itr = graph_adjacent_vertices( G, v );
        int connectivity_degree = 0;

        /* Compute the number of adjacent vertices of v that
           are selected
        */
        while ( ( u = adjacency_itr_next( &a_itr ) ) != NULL )
        {
            connectivity_degree += v_selected[ idx( u ) ];
        }

        /* If v has a higher connectivity than the highest,
           store the index of v in the list, and the connectivity
        */
        if ( connectivity_degree > max_degree )
        {
            max_degree = connectivity_degree;
            max_i = i;
        }
    }

    /* Let v be the vertex with the highest spine connectivity */
    v = ((vertex**) v_list->data)[ max_i ];
    /* Remove v from the vertex list */

```

```

g_array_remove_index_fast( v_list, max_i );
/* return v */
return v;
}

void
nicholson( graph *G, graph *cr_G
           , GArray *spine, GArray *spine_inv
           , GArray *e_map, GArray *page )
{
    int no_mapped_vertices = 0;
    /* Boolean array that indicates whether a vertex has already been
       mapped to the spine
       */
    int v_selected[ graph_num_vertices( G ) ];
    /* construct a list of vertices from G, from which vertices will be drawn
       by the algorithm as it sequentially embeds vertices.
       */
    GArray *v_list = get_v_list( G );
    vertex *v;

    /* First set all vertices up as having been placed onto the spine, so
       that the procedure remove_vertex_with_highest_spine_connectivity
       will find a vertex attaining the maximum degree in G
       */
    FILL_ARRAY( v_selected, graph_num_vertices( G ), 1 );
    v = remove_vertex_with_highest_spine_connectivity( G, v_selected, v_list );

    /* Then modify v_selected so that only the first vertex, v, is selected */
    FILL_ARRAY( v_selected, graph_num_vertices( G ), 0 );
    v_selected[ idx( v ) ] = 1;

    /* Add v to the spine */
    extend_intersection_graph( G, cr_G, spine, spine_inv, e_map, v, 0 );
    no_mapped_vertices += 1;

    /* while vertices remain in the vertex list */
    while ( v_list->len > 0 )
    {
        int best_cr = INT_MAX;
        int best_idx = 0;
        int i;
        v = remove_vertex_with_highest_spine_connectivity( G, v_selected, v_list );

        /* For each position on the spine */
        for ( i = 1; i < no_mapped_vertices + 1; i++ )
        {
            int cur_cr;

            /* Add v to the spine at position i */
            extend_intersection_graph( G, cr_G, spine, spine_inv, e_map, v, i );
            /* Determine placements for the edges incident to v that
               minimizes the number of crossings caused
               */
            cur_cr = choose_best_pages_for_adjacent_edges( G, cr_G, e_map, v, page );

            /* If the number of crossings for v at position i improves
               upon the best known minimum, record the position i and
               */
            if ( cur_cr < best_cr )
            {

```

```

        best_cr = cur_cr;
        best_idx = i;
    }

    /* remove v from the spine at position i */
    prune_intersection_graph( G, cr_G, spine, spine_inv, e_map, v );
}

/* Insert v at the spine position that minimizes the number of
   crossings caused by the insertion
*/
extend_intersection_graph( G, cr_G, spine, spine_inv, e_map, v, best_idx );
/* Place the edges incident to v such that best_cr crossings are caused
choose_best_pages_for_adjacent_edges( G, cr_G, e_map, v, page );
/* Note that the number of vertices was increased */
no_mapped_vertices += 1;
/* And that v has been mapped */
v_selected[ idx( v ) ] = 1;
}
}

```

### B.2.1.2 Pósa’s heuristic probabilistic algorithm for finding Hamiltonian cycles

Pósa [P76] proved that a random graph  $\mathcal{G}$ , with  $|E(\mathcal{G})| = O(\log(|V(\mathcal{G})|)|V(\mathcal{G})|)$  contains a Hamiltonian cycle with a high probability. A pivotal idea used by his proof is the concept of a rotational transformation. It is this idea that allows the construction of a heuristic algorithm for finding Hamiltonian cycles.

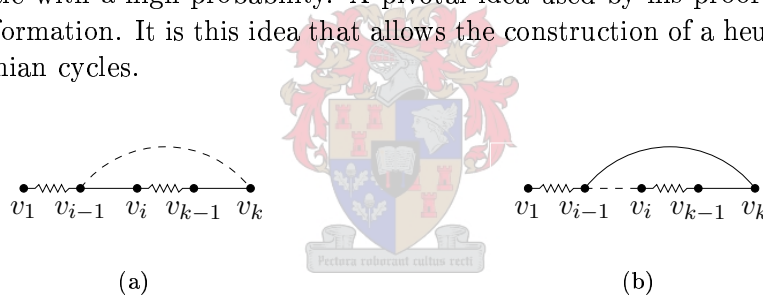


Figure B.1: An illustration of a rotational transformation with the vertex  $v_{i-1}$  as a pivot.

In a graph  $\mathcal{G}$ , a rotational transformation is simply a method by which a path  $\mathcal{P} = v_1, v_2, \dots, v_{i-1}, v_i, \dots, v_{k-1}, v_k$  in  $\mathcal{G}$  may be transformed into a path  $\mathcal{P}' = v_1, v_2, \dots, v_{i-1}, v_k, v_{k-1}, \dots, v_i$ , if there is an edge  $\{v_{i-1}, v_k\}$  in  $\mathcal{G}$ . In other words,  $\mathcal{P}'$  is obtained by deleting the edge  $\{v_{i-1}, v_i\}$  from  $\mathcal{P}$ , and by adding the edge  $\{v_{i-1}, v_k\}$  to  $\mathcal{P}$ . This transformation causes the order in which the vertices  $v_i, \dots, v_{k-1}, v_k$  appear in  $\mathcal{P}$  to be reversed as they appear in  $\mathcal{P}'$ , hence the name of transformation. For the purposes of this section, the vertex  $v_{i-1}$ , which is the last vertex in  $\mathcal{P}$  to occur before the set of rotated vertices, is known as a *pivot*. The transformation is illustrated in Figure B.1.

The idea is for the algorithm to construct a path, one vertex at a time. Ultimately, this path should contain all vertices of the graph, and the end-vertices of this path should be joined by an edge in the input graph, rendering a Hamiltonian cycle.

Let  $\mathcal{P}_i$  denote the vertex at position  $i$  in the path  $\mathcal{P}$ , where  $\mathcal{P}_0$  and  $\mathcal{P}_{n-1}$  are end-vertices, and where  $\mathcal{P}_0$  is the initial vertex with which the algorithm commenced, and which is never moved (since there is no potential pivot vertex placed before  $\mathcal{P}_0$ ). Given a path of length  $n$ ,  $n < |V(\mathcal{G})|$ , for an input graph  $\mathcal{G}$ , a random neighbour  $v$  of the last vertex in the path,  $\mathcal{P}_{n-1}$ , is selected. If  $v \notin \mathcal{P}$ , then the path is extended by placing  $v$  at the index  $n$  in  $\mathcal{P}$ . If, however,  $v \in \mathcal{P}$ ,  $\mathcal{P}$  is searched until an index  $i$  is found such that  $\mathcal{P}_i$  is a pivot vertex. A rotational transformation is

applied, which places the vertex  $\mathcal{P}_{i+1}$  at position  $n - 1$ , which is the end-vertex of  $\mathcal{P}$ . A random neighbour  $v'$  of  $\mathcal{P}_{n-1}$  is again selected and if  $v' \in \mathcal{P}$ , the process is repeated. Every vertex in  $\mathcal{P}$  may only be placed at the end index,  $n - 1$ , once, and if all vertices of  $\mathcal{P}$  have been placed at this position, and if  $\mathcal{P}$  does not contain all the vertices of  $\mathcal{G}$ , the algorithm terminates, and reports that no Hamiltonian cycle could be found. Otherwise, if all vertices have been placed into  $\mathcal{P}$ , the algorithm attempts to join the end-vertices of  $\mathcal{P}$  by an edge. If no such edge exists, the algorithm searches the list  $\mathcal{P}$  for a vertex  $\mathcal{P}_i$  that is joined to  $\mathcal{P}_0$  and for which the vertex  $\mathcal{P}_{i-1}$  can serve as a pivot (allowing  $\mathcal{P}_i$  to be placed at the end of  $\mathcal{P}$ ). If this is not possible, the algorithm reports failure.

The Python implementation of the algorithm is fully annotated with source code comments, as opposed to the C implementation which contains virtually no comments. It is therefore suggested that the Python algorithm be studied first if the reader wishes to understand the C implementation.

The array `P` represents the path that is constructed by the algorithm. The array `P_inv` maps vertices to their corresponding indices in `P`. For a vertex `v`, the array `P_processed` records the highest index at which a `v` has been placed in `P`, so that it may be determined whether `v` may be shifted to the end of `P` via a rotational transformation.

Both implementations contain optimizations that may be enabled by setting the appropriate flags. For the first optimization, when a neighbour of the last vertex in  $\mathcal{P}$  must be selected, a vertex is selected from the neighbours that are not in  $\mathcal{P}$ . This avoids unnecessary rotational transformations by extending a path as far as possible without performing any such transformations.

For the second optimization, neighbours that are not in  $\mathcal{P}$  and that have vertex degrees equal to two are selected in preference to any other neighbours. The justification for this optimization is that such vertices may be viewed as subdivided vertices and the algorithm should therefore attempt to traverse an entire subdivided edge before performing a rotational transformation.

## Python implementation

```
def hamiltonian_posa(G, follow_unmapped_vertices=False, follow_unmapped_2_deg_vertices=False):
    def random_element( seq ):
        """return a randomly selected element from the sequence seq
        """
        if len( seq ) > 0:
            return seq[ random.randint( 0, len( seq ) - 1 ) ]
        else:
            return None

    start_v = random_element( list( G.vertices() ) )

    # P is an array of vertices of G, representing the path that the algorithm
    # constructs as it progresses.
    P = [ start_v ]

    # P_inv maps vertices in G to their indices in P
    P_inv = dict( zip( G.vertices(), [ -1 for v in G.vertices() ] ) )
    P_inv[ start_v ] = 0

    # P_processed[ v ] for a vertex is smaller than len( P ) if v has not
    # yet been placed at the end of the path via a rotational transformation
    # for a given path length, and P_processed[ v ] == len( P ) if it has
    # already been placed there. Thus, if P_processed[ v ] == len( P ), it
```

```

# is known that v must not again be placed at the end of P.
P_processed = dict( zip( G.vertices(), [ 0 for v in G.vertices() ] ) )

def rotate( P, P_inv, new_end ):
    """Perform a rotational transformation with new_end as the pivot
    """
    start_pos = P_inv[ new_end ] + 1

    for i in xrange( 0, ( len( P ) - start_pos ) / 2 ):
        P[ start_pos+i ], P[ -1-i ] = P[ -1-i ], P[ start_pos+i ]
        P_inv[ P[start_pos+i] ], P_inv[ P[-1-i] ] = P_inv[ P[-1-i] ], P_inv[ P[start_pos+i] ]

def make_cycle( G, P, P_inv ):
    """Given a path P, try to find a rotational transformation that will
    yield a path with end-vertices that may be joined by an edge
    """
    # if there is no edge between the first and last vertices in P...
    if G.spanning_edge( P[ 0 ], P[ -1 ] ) == None:
        # for each adjacent vertex v to the last vertex in the path P
        for v in G.adjacent_vertices( P[ -1 ] ):
            # if the vertex following v in P (i.e. P[ P_inv[ v ] + 1 ])
            # is joined by an edge to the first vertex in P then...
            if G.spanning_edge( P[ 0 ], P[ P_inv[ v ] + 1 ] ) != None:
                # perform a rotation which places this vertex at the end
                # of the path, so that the sequence of the vertices corresponds
                # to the sequence of vertices on the cycle.
                rotate( P, P_inv, v )
                return True
    else:
        return True

    return False

while True:
    next_vertex = None

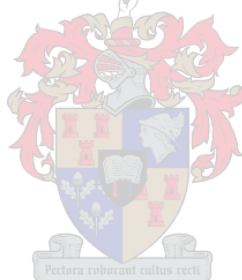
    # if vertices which have not been mapped to the path should be
    # visited in preference to vertices already in the path...
    if follow_unmapped_vertices:
        # compute the list of unmapped vertices adjacent to end-vertex of P
        # an unmapped vertex v has the property that P_inv[ v ] = -1
        open_candidates = [ v for v in G.adjacent_vertices( P[ -1 ] ) if P_inv[ v ] == -1 ]

        # if unmapped vertices with degree two should be visited in preference
        # to other unmapped vertices...
        if follow_unmapped_2_deg_vertices:
            # compute the list of such vertices
            open_2deg_candidates = [ v for v in open_candidates if G.degree( v ) == 2 ]
            # and select a random element from the set
            next_vertex = random_element( open_2deg_candidates )

        # if either unmapped vertices with degree two were not given preference,
        # or if they were given preference, but there were no such vertices adjacent
        # to the end-vertex, then...
        if next_vertex == None:
            next_vertex = random_element( open_candidates )

    # if unmapped vertices were given selection preference, but the end-vertex had no
    # such vertex, or if no such preference was given, then...
    if next_vertex == None:
        # compute a list of vertices which have not yet been placed at the end of the path

```



```

# via rotational transformations. Such a vertex v has the property
# that P_processed[ v ] < len( P )
candidates = [ v for v in G.adjacent_vertices( P[ -1 ] ) if P_processed[ v ] < len( P ) ]
next_vertex = random_element( candidates )

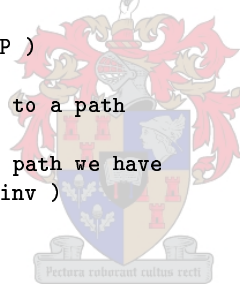
# if no vertex could be found to extend the path any further
if next_vertex == None:
    # try to create a cycle with the path we have
    has_cycle = make_cycle( G, P, P_inv )
    return P, len( P ), has_cycle

# if the selected vertex has not yet been mapped...
if P_inv[ next_vertex ] < 0:
    # it will be placed at the end of the path, since it must
    # have been found to have been adjacent to the vertex at
    # the end of the path.
    P_inv[ next_vertex ] = len( P )
    P.append( next_vertex )
# else if the selected vertex has been mapped but not yet
# placed at the end of the path via a rotational transformation...
elif P_processed[ next_vertex ] < len( P ):
    # then place it there via such a rotation
    rotate( P, P_inv, next_vertex )

# The selected vertex is marked as being at the end of the
# current path, so that it will not be placed there again via
# a rotational transformation.
P_processed[ next_vertex ] = len( P )

# if all vertices have been mapped to a path
if len( P ) == G.num_vertices():
    # try to create a cycle with the path we have
    has_cycle = make_cycle( G, P, P_inv )
    return P, len( P ), has_cycle

```



## C implementation

The C implementation contains one additional feature not present in the Python implementation. In the Python implementation, the path  $P$  is constructed by always extending it only in one direction. That is, no vertex is ever placed before  $P[0]$ . However, the algorithm may reach a point where  $P$  can only be extended from  $P[0]$ . The C implementation takes this into consideration, and reverses the array  $P$  when this occurs (so that path extension happens from the opposite side), by calling the function `reverse_path` on  $P$  and by re-attempting path extension.

```

PRIVATE INLINE void
rotate( vertex **path, int *path_inv, int vertices_mapped, vertex *new_end )
{
    int start_pos = path_inv[ idx( new_end ) ] + 1;
    int end_pos = vertices_mapped - 1;

    for ( int i = 0; i < ( end_pos - start_pos ) / 2; i++ )
    {
        SWAP( vertex*, path[ start_pos + i ], path[ end_pos - i ] );
        SWAP( int, path_inv[ idx(path[start_pos + i]) ], path_inv[ idx(path[end_pos - i]) ] );
    }
}

#define IDX2( width, row, col ) ( (width) * (row) + (col) )

```

```

#define NEXT_IN_PATH( path, path_inv, v ) path[ path_inv[ idx( v ) ] + 1 ]

PRIVATE void
reverse_path( vertex **path, int *path_inv, int vertices_mapped )
{
    for ( int i = 0; i < vertices_mapped / 2; i++ )
        {
            SWAP( vertex*, path[ i ], path[ vertices_mapped - 1 - i ] );
            SWAP( vertex*, path_inv[idx(path[i])], path_inv[idx(path[vertices_mapped - 1 - i])] );
        }
}

bool
make_cycle( graph *G, vertex **path, int *path_inv, int vertices_mapped )
{
    vertex *last_in_path = path[ vertices_mapped - 1 ];

    if ( graph_spanning_edge( G, path[ 0 ], last_in_path ) == NULL )
        {
            for ( int i = 0; i < 2; i++ )
                {
                    vertex *v;
                    adjacency_itr a_itr = graph_adjacent_vertices( G, last_in_path );
                    while ( ( v = adjacency_itr_next( &a_itr ) ) )
                        {
                            if ( graph_spanning_edge( G, path[ 0 ], NEXT_IN_PATH( path, path_inv, v ) ) )
                                {
                                    rotate( path, path_inv, vertices_mapped, v );
                                    return true;
                                }
                        }

                    reverse_path( path, path_inv, vertices_mapped );
                }

            return false;
        }

    else return true;
}

bool
hamiltonian_posa( int *vertices_mapped, graph *G, GArray *g_path, int flags )
{
    int path_inv[ graph_num_vertices( G ) ];
    int processed[ graph_num_vertices( G ) ];
    vertex **path = (vertex**) (g_path->data);

    memset( processed, '\0', sizeof( int ) * graph_num_vertices( G ) );
    for ( int i = 0; i < graph_num_vertices( G ); i++ ) { path_inv[ i ] = -1; }

    path[ 0 ] = graph_random_vertex( G );
    path_inv[ idx( path[ 0 ] ) ] = 0;
    *vertices_mapped = 1;

    while ( 1 )
        {
            vertex *next_vertex = NULL;
            vertex *last_in_path = path[ *vertices_mapped - 1 ];
            vertex *v;

```



```

for ( int i = 0; i < 2; i++ )
{
    if ( flags & FOLLOW_UNMAPPED_VERTICES )
    {
        vertex *open_candidates[ graph_num_vertices( G ) ];
        int open_candidates_idx = 0;

        adjacency_itr a_itr = graph_adjacent_vertices( G, last_in_path );
        while ( ( v = adjacency_itr_next( &a_itr ) ) )
        {
            if ( path_inv[ idx( v ) ] < 0 )
                open_candidates[ open_candidates_idx++ ] = v;
        }

        if ( flags & FOLLOW_UNMAPPED_2_DEG_VERTICES )
        {
            vertex *open_2deg_candidates[ graph_num_vertices( G ) ];
            int open_2deg_candidates_idx = 0;

            for ( int i = 0; i < open_candidates_idx; i++ )
            {
                if ( graph_degree( G, open_candidates[ i ] ) == 2 )
                    open_2deg_candidates[ open_2deg_candidates_idx++ ] = open_candidates[ i ];
            }

            if ( open_2deg_candidates_idx > 0 )
                next_vertex = open_2deg_candidates[ UNIFORM_INT( 0, open_2deg_candidates_idx ) ];
        }

        if ( next_vertex == NULL && open_candidates_idx > 0 )
            next_vertex = open_candidates[ UNIFORM_INT( 0, open_candidates_idx ) ];
    }

    if ( ( flags & SWAP_PATH ) && next_vertex == NULL )
        reverse_path( path, path_inv, *vertices_mapped );
    else
        break;
}

for ( int i = 0; i < 2; i++ )
{
    if ( next_vertex == NULL )
    {
        vertex *candidates[ graph_num_vertices( G ) ];
        int candidates_idx = 0;
        adjacency_itr a_itr = graph_adjacent_vertices( G, last_in_path );
        while ( ( v = adjacency_itr_next( &a_itr ) ) )
        {
            if ( processed[ idx( v ) ] < *vertices_mapped )
                candidates[ candidates_idx++ ] = v;
        }

        if ( candidates_idx > 0 )
            next_vertex = candidates[ UNIFORM_INT( 0, candidates_idx ) ];
    }

    if ( ( flags & SWAP_PATH ) && next_vertex == NULL )
        reverse_path( path, path_inv, *vertices_mapped );
    else
        break;
}

```

```

    if ( next_vertex == NULL )
        return make_cycle( G, path, path_inv, *vertices_mapped );

    if ( path_inv[ idx( next_vertex ) ] < 0 )
    {
        path_inv[ idx( next_vertex ) ] = *vertices_mapped;
        path[ *vertices_mapped ] = next_vertex;
        *vertices_mapped += 1;
    }

    else if ( processed[ idx( next_vertex ) ] < *vertices_mapped )
        rotate( path, path_inv, *vertices_mapped, next_vertex );

    processed[ idx( next_vertex ) ] = *vertices_mapped;

    if ( *vertices_mapped == graph_num_vertices( G ) )
        return make_cycle( G, path, path_inv, *vertices_mapped );
}
}

```

## B.2.2 Tabu algorithm

The tabu search algorithm in this section implements Algorithm 6.8. The implementation is quite faithful to the pseudo-code of Algorithm 6.8, and clarification of the code should therefore be sought in § 6.2.2, where Algorithm 6.8 is described in detail.

The tabu memory is simply a vector indexed by spine position. When a vertex is moved to a particular position on the spine, that position becomes poisoned for a number of steps. The tabu active list (`poison_q`) simply maintains a list of indices that are poisoned.

```

PRIVATE void
tabu_memory_insert( tabu_memory *self, int from_pos, int to_pos )
{
    /* It is inconvenient to work directly with GArray structures,
       so obtain handles to the arrays */
    int *poison = (int*) self->poison->data;
    int *poison_freq = (int*) self->poison_freq->data;

    /* If the position to be poisoned is not currently poisoned... */
    if ( poison[ to_pos ] == 0 )
        /* insert it into the tabu list of poisoned items */
        g_array_append_val( self->poison_q, to_pos );

    /* Update the poison value for the position to be poisoned
       poison[ to_pos ] += self->poison_length
       + ((double) poison_freq[to_pos] / self->max_poison_freq) * self->freq_penalizer;

    /* update the frequency with which the position to_pos has been poisoned */
    poison_freq[ to_pos ] += 1;
    /* update the maximum poisoning frequency */
    self->max_poison_freq = MAX( poison_freq[ to_pos ], self->max_poison_freq );
}

PRIVATE INLINE int
tabu_memory_tabu_value( tabu_memory *self, int from_pos, int to_pos )
{
    /* The tabu value of move is simply the poison value of position to
       which it will move the vertex */
}

```

```

    return g_array_index( self->poison, int, to_pos );
}

/**
 * The result structure holds the best found configuration
 */

PUBLIC int
tabu( graph *G
      , graph *cr_G
      , GArray *spine
      , GArray *spine_inv
      , GArray *e_map
      , GArray *page
      , crossing_no_alg *cross_alg
      , int iterations
      , int poison_length
      , int freq_penalizer
      , int _max_steps_since_improvement )
{
/*1,2*/
    int best_crossing_no = crossing_number( page, cr_G, 0 );
/*3*/
    tabu_memory *memory = tabu_memory_new( G, spine, poison_length, freq_penalizer );
    result *best_result = result_new( G, cr_G, spine, spine_inv, e_map, page );
    int current_iteration;
    prio_q *elite_solutions = prio_q_new();
    int max_steps_since_improvement = _max_steps_since_improvement;
    int steps_since_improvement = 0;

    graph_maintain_vertex_array( cr_G, "page", page->data, sizeof( int ) );

/*4,5*/
    for ( current_iteration = 0; current_iteration < iterations; current_iteration++ )
    {
        int i, j;
        int candidate_from = -1;
        int candidate_to = -1;

        bool revisit = false;

        int candidate_tabu_value = INT_MAX;
        int candidate_crossing_no = INT_MAX;

/*6*/
        steps_since_improvement += 1;

/*7*/
        for ( i = 1; i < graph_num_vertices( G ); i++ )
        {
/*8*/
            for ( j = 1; j < graph_num_vertices( G ); j++ )
            {
/*9*/
                if ( i == j ) continue;
/*10*/
                move_vertex_on_spine( G, cr_G, spine, spine_inv, e_map, i, j );

                /* Using the input page layout algorithm, cross_alg, compute a layout and
                 return the number of crossings in the layout */
/*11,12*/
                int tst_crossing_no = crossing_no_alg_compute( cross_alg, cr_G, page );
                /* Compute the tabu value of the move */
/*13*/
                int tst_tabu_value = tabu_memory_tabu_value( memory, i, j );

```

```

/*14*/     if ( tst_crossing_no < best_crossing_no )
            {
/*15*/         result_store( best_result, cr_G, spine, spine_inv, e_map, page );
/*16*/         best_crossing_no = tst_crossing_no;
/*16*/         candidate_tabu_value = 0;
/*16*/         steps_since_improvement = 0;
/*16*/         revisit = false;
            }

/*18*/     if ( ( tst_tabu_value == 0 && tst_crossing_no < candidate_crossing_no )
            || ( tst_tabu_value > 0 &&
                ( tst_tabu_value == candidate_tabu_value
                  && tst_crossing_no < candidate_crossing_no
                  || tst_tabu_value < candidate_tabu_value ) ) )
        {
/*19*/         candidate_tabu_value = tst_tabu_value;
/*19*/         candidate_from = i;
/*19*/         candidate_to = j;
/*19*/         candidate_crossing_no = tst_crossing_no;
        }

/*21*/     move_vertex_on_spine( G, cr_G, spine, spine_inv, e_map, j, i );
    }
}

/*25*/
if ( steps_since_improvement == 1 && revisit == false )
{
/*26*/     prio_q_insert( elite_solutions, (void*) result_new_copy( best_result ), 0 );
}

/*28*/     tabu_memory_decrease_tenure( memory );

/*29*/
if ( steps_since_improvement >= max_steps_since_improvement
    && elite_solutions->elements_in_heap > 0 )
{
    int old_prio;
    int elite_crossing_no;
    int found_good_elite;
    result *new_elite;
/*30*/     do
        {
/*31*/         old_prio = prio_q_min_prio( elite_solutions );
/*31*/         new_elite = (result*) prio_q_extract_min( elite_solutions );
/*31*/         elite_crossing_no
            = crossing_no_alg_compute( cross_alg, new_elite->cr_G, new_elite->page );
        found_good_elite = true;
        if ( elite_crossing_no > (int) (1.05 * (float)best_crossing_no)
            && elite_solutions->elements_in_heap > 2 )
            {
/*33*/                 result_free( new_elite );
                found_good_elite = false;
            }
        }
    }

/*35*/     while ( ! found_good_elite );
    result_extract( new_elite, cr_G, spine, spine_inv, e_map, page );
/*36*/     prio_q_insert( elite_solutions, (void*) new_elite, old_prio + 1 );
/*37*/     revisit = true;
/*37*/     steps_since_improvement = 0;

```

```

    }

    else
    {
/*39*/ tabu_memory_insert( memory, candidate_from, candidate_to );
/*40*/ move_vertex_on_spine(G, cr_G, spine, spine_inv, e_map, candidate_from, candidate_to);
    }
}

result *elite_result;
while ( ( elite_result = (result*) prio_q_extract_min( elite_solutions ) ) )
{
    result_free( elite_result );
}
prio_q_free( elite_solutions );

graph_forget_vertex_array( cr_G, "page" );
result_extract( best_result, cr_G, spine, spine_inv, e_map, page );
result_free( best_result );
tabu_memory_free( memory );
/*43*/
return best_crossing_no;
}

```

## B.3 The lower bound algorithm

Because the lower bound algorithm is really only effective for large graphs, the crossing number of the graph  $\mathcal{G}$  that must be embedded into the graph  $\mathcal{H}$  might be much larger than the values that can be stored in normal machine integers. For this reason, a variable precision number library, called GMP [GMP] was used in the calculation of the lower bound. The integer type in the GMP library was used, and all arithmetic functions operating on numbers of this type start with the text “mpz\_.”

### B.3.1 Compute weights

The algorithm in this section is an implementation of Algorithm 6.1. It matches Algorithm 6.1 closely, and it is recommended that the reader refer to the description of Algorithm 6.1 in § 6.1 to understand this implementation.

```

PRIVATE INLINE void
compute_weights( graph *H
                , double *weight
                , int num_edges_G
                , int max_edge_congestion
                , int *vertex_congestion
                , int *edge_congestion
                , bool crossing_pruning )
{
    edges_itr e_itr = graph_edges( H );
    edge *e;

/*1*/
    while ( ( e = edges_itr_next( &e_itr ) ) != NULL )
    {
/*2*/ weight[ idx( e ) ] = vertex_congestion[ idx( graph_target( H, e ) ) ];
}

```

```

/*3*/ if ( edge_congestion[ idx( e ) ] == max_edge_congestion )
/*4*/   weight[ idx( e ) ]
      += num_edges_G * num_edges_G * graph_num_vertices( H );
/*5*/ else
/*6*/   weight[ idx( e ) ]
      += edge_congestion[ idx( e ) ]
      / ( max_edge_congestion * graph_num_vertices( H ) );

/*8*/ if ( vertex_congestion[ idx( graph_target( H, e ) ) ]
      == 0 && edge_congestion[ idx( e ) ] == 0 )
    {
/*9*/   weight[ idx( e ) ] = 0.0001;
    }

/*12*/ weight is not returned, because it was passed in as a pointer
}

```

### B.3.2 Lowerbound

This section provides an implementation of the lower bound algorithm, Algorithm 6.2. The implementation is close enough in form to the pseudo-code of Algorithm 6.2, so that the best way to understand it would be to refer to pseudo-code and description of the algorithm in § 6.1.

In this implementation, a list of edges of  $G$ , `edge_array`, is constructed. The edges of  $G$  are selected from this list to be embedded into  $H$ . This list is permuted in a random fashion so that two consecutive embeddings of  $G$  into  $H$  will not yield the same edge mappings.

```

PRIVATE INLINE int
compute_graph_to_graph_mapping( mpz_ptr C
                               , graph * G
                               , graph * H
                               , ptr_non_null_vertex * psi
                               , ptr_owned_GList * vertex_mapped_edges
                               , int * vertex_congestion
                               , int * edge_congestion
                               , bool crossing_pruning
                               , int do_min_embed )
{
    int max_edge_congestion = 0;
/*1*/
    mpz_set_ui( C, 0 );

    int no_V_H = graph_num_vertices( H );
    int no_E_H = graph_num_edges( H );
    int no_E_G = graph_num_edges( G );

    double *weight;
    edge* *idx_to_edge;
    edge* *edge_array;
    int *path_len;
    fast_set *tau = fast_set_new( no_E_G );

    edge* pred[ no_V_H ];
    double dist[ no_V_H ];
    GList* edge_at_vertex[ no_V_H ];

    weight = NEW( double, no_E_H );

```

```

edge_array = NEW( edge*, no_E_G );
idx_to_edge = NEW( edge*, no_E_G );

FILL_ARRAY( edge_at_vertex, no_V_H, NULL );
FILL_ARRAY( weight, no_E_H, 1.0 );

/* Construct two arrays that map edge indices of G to the edges of G.
   The mapping is irrelevant for the first array, edge_array, since
   this array is randomly permuted. However, the mapping of
   idx_to_edge is significant, as may be seen later in the algorithm
*/
edge *e;
edges_itr e_itr = graph_edges( G );
while ( ( e = edges_itr_next( &e_itr ) ) )
{
    edge_array[ idx( e ) ] = e;
    idx_to_edge[ idx( e ) ] = e;
}

/* Permute edge_array randomly */
ptr_array_random_shuffle( (ptr_void*) edge_array
                        , (ptr_void*) ( edge_array + graph_num_edges( G ) ) );

/*2*/
for ( int i = 0; i < no_E_G; i++ )
{
    edge *e = EDGE( edge_array[ i ] );
    vertex *psi_source = VERTEX( psi[ idx( graph_source( G, e ) ) ] );
    vertex *psi_target = VERTEX( psi[ idx( graph_target( G, e ) ) ] );
    vertex *v;

/*3*/ compute_weights( H, weight, no_E_G, max_edge_congestion
                    , vertex_congestion, edge_congestion
                    , crossing_pruning );
/*4*/ dijkstra_ll( H, psi_source, weight, pred, dist );

    v = psi_target;

    /* set 'on_same_path' for all edges of G that map through v \in V(H),
       * which excludes any edges of G which start their path at v itself. */
/*6*/ for ( GList *list_itr = g_list_first( edge_at_vertex[ idx( v ) ] );
          list_itr != NULL; list_itr = g_list_next( list_itr ) )
    {
        edge *f = EDGE( list_itr->data );

        fast_set_insert( tau, idx( f ) );
        /* A crossing between e and f is possible if f doesn't start at the vertex v */
/*7*/ if ( psi[ idx( graph_source( G, f ) ) ] != psi_target
          && psi[ idx( graph_target( G, f ) ) ] != psi_target )
/*8*/     mpz_add_ui( C, C, 1 );
    }

/*11*/
    edge_at_vertex[idx(psi_target)] = g_list_prepend( edge_at_vertex[ idx(psi_target) ], e );
/*12*/
    vertex_congestion[ idx( psi_target ) ] += 1;

/*13*/
    while ( v != psi_source )
    {
        /* Climb the parent tree to get the edge pointing to v */

```

```

    edge *f = EDGE( pred[ idx( v ) ] );

    /* Update the vertex and edge congestion arrays */
/*14*/   edge_congestion[ idx( f ) ] += 1;
/*15*/   edge_congestion[ idx(graph_reverse( H, f )) ] = edge_congestion[ idx( f ) ];
/*16*/   v = VERTEX( graph_source( H, f ) );
/*17*/   vertex_congestion[ idx( v ) ] += 1;
    max_edge_congestion = MAX( edge_congestion[ idx(f) ], max_edge_congestion );

/*18*/   for ( GList *list_itr = g_list_first( edge_at_vertex[ idx( v ) ] );
           list_itr != NULL; list_itr = g_list_next( list_itr ) )
    {
        edge *f = EDGE( list_itr->data );
/*19*/       if ( !fast_set_in( tau, idx( f ) )
                && ( v != psi_source
                    || ( psi[ idx( graph_source( G, f ) ) ] != psi_source
                        && psi[ idx( graph_target( G, f ) ) ] != psi_source ) ) )
        {
/*20*/           mpz_add_ui( C, C, 1 );
        }
    }

    /* The following code differs slightly from Algorithm 6.2
       Firstly, the code for line 30 entails that tau is filled
       with entries from edge_at_vertex. This is never necessary
       when v == psi_source, since this loop would terminate,
       and the contents of tau constructed anew for the next
       edge considered. Thus, tau is only filled when v != psi_source. */
/*23*/   if ( v != psi_source )
    {
/*30*/       fast_set_clear( tau );
/*30*/       for ( GList *list_itr = g_list_first( edge_at_vertex[ idx( v ) ] );
                list_itr != NULL; list_itr = g_list_next( list_itr ) )
        {
/*30*/           fast_set_insert( tau, idx( EDGE( list_itr->data ) ) );
        }
    }

/*23*/   else
    {
/*24*/       for ( int i = 0; i < tau->size; i++ )
        {
/*25*/           edge *f = idx_to_edge[ g_array_index( tau->items, int, i ) ]
                if ( psi[ idx( graph_source( G, f ) ) ] == v
                    || psi[ idx( graph_target( G, f ) ) ] == v )
/*26*/               mpz_sub_ui( C, C, 1 );
        }

        fast_set_clear( tau );
    }

    /* Add e to the list of edges mapped through v */
/*31*/   edge_at_vertex[ idx(v) ] = g_list_prepend( edge_at_vertex[ idx(v) ], EDGE(e) );
}

}

fast_set_free( tau );
FREE( weight );
FREE( edge_array );
FREE( idx_to_edge );

```



```

    return 0;
}

```

### B.3.3 LowerTabu

The algorithm in this section is an implementation of a tabu search algorithm for finding vertex mappings in graph-to-graph embeddings (§ 6.1). Its conceptual structure is similar to that of the tabu search algorithm implementation (§ B.2.2) and it is recommended that the reader first understand how the upper bound tabu algorithm works, before attempting to understand the lower bound tabu algorithm.

In order to understand the functioning of the algorithm, it is important to understand its move types. Suppose a graph  $\mathcal{G}$  is graph-to-graph embedded into a graph  $\mathcal{H}$ . If  $|E(\mathcal{G})| = |E(\mathcal{H})|$ , it is possible to define a move in which the mappings of a pair of vertices in  $\mathcal{G}$  to vertices in  $\mathcal{H}$ , are swapped. Let such a move be known as a “swap” move. If  $|E(\mathcal{G})| < |E(\mathcal{H})|$ , there are vertices in  $\mathcal{H}$  which are not the images of vertices in  $\mathcal{G}$ , and it is possible to define an additional move in which a mapping of a vertex in  $\mathcal{G}$  is changed so as to map to a such a vertex in  $\mathcal{H}$ . Such a move is referred to as a “shift” move.

In the implementation, when a swap move occurs, the swapped vertices should maintain their positions for a number of steps (to avoid erratic swapping behaviour by the algorithm, and to encourage exploration of solutions containing the pair of vertices in a given order). This is achieved by letting the pair of vertices in  $\mathcal{H}$ , that are swapped by the move, define an attribute. The attribute memory for this attribute is therefore a two-dimensional structure that is indexed by a pair of vertices from  $\mathcal{H}$ .

A simpler attribute scheme is used for shift moves — when a mapping of a vertex  $u$  in  $\mathcal{G}$  is shifted to a previously unmapped vertex  $v$  in  $\mathcal{H}$ , the mapping should remain fixed for a number of steps, unless it can be shifted to a better position. Other shift moves cannot change the fact that  $u$  in  $\mathcal{G}$  maps to  $v$  in  $\mathcal{H}$ , since they can only reassign mappings of vertices from  $\mathcal{G}$  to vertices of  $\mathcal{H}$  that are not images of vertices in  $\mathcal{G}$ . However, swap moves can cause the mapping to be reassigned. To guard against this, the vertex in  $\mathcal{H}$  that becomes an image under a shift move defines an attribute. This attribute is easily implemented as a vector.

At each iteration, the tabu algorithm enumerates all possible swap moves, and if possible, shift moves, of its current vertex mapping. Each such move is performed, and for each move, the function `evaluate_tabu` is called, which computes a lower bound of the crossing number of  $\mathcal{H}$  using the given vertex mapping. This function also records the kind of move that led to the best solution, so that it can be determined after the neighbourhood perusal how the vertex mapping should be updated.

```

void
evaluate_tabu( tabu_vars *vars, int idx1, int idx2, int tst_tabu_value, int enum_mode )
{
    bool improving = true;

    while ( improving )
    {
        improving = false;
        lower_bound( vars->tst_crossing_no, vars->G, vars->H, vars->psi
                    , vars->lower_G, vars->lower_H, vars->crossing_pruning, vars->do_min_embed );

        if ( mpz_cmp( vars->tst_crossing_no, vars->lower_H ) > 0 )

```

```

    {
        improving = true;
        result_store( vars->best_result, vars->psi );
        mpz_set( vars->lower_H, vars->tst_crossing_no );
        vars->candidate_tabu_value = 0;
    }
}

if ( (tst_tabu_value == 0 && mpz_cmp(vars->tst_crossing_no, vars->candidate_crossing_no) > 0)
    || ( tst_tabu_value > 0 && tst_tabu_value <= vars->candidate_tabu_value ) )
{
    vars->candidate_tabu_value = tst_tabu_value;

    if ( enum_mode == DO_MOVE )
    {
        vars->from_vertex_idx = idx1;
        vars->to_vertex_idx = idx2;
        vars->improving_enum_mode = DO_MOVE;
    }

    else /* enum_mode == DO_SWAP */
    {
        vars->vertex_a_idx = idx1;
        vars->vertex_b_idx = idx2;
        vars->improving_enum_mode = DO_SWAP;
    }

    if (tst_tabu_value == 0 && mpz_cmp(vars->tst_crossing_no, vars->candidate_crossing_no) > 0)
        mpz_set( vars->candidate_crossing_no, vars->tst_crossing_no );
}

}

void
lower_tabu( mpz_ptr bound
           , graph *G
           , graph *H
           , GArray *psi
           , mpz_ptr lower_G
           , mpz_ptr lower_H
           , int iterations
           , int min_avoid_pos
           , int max_avoid_pos
           , int min_xchge
           , int max_xchge
           , bool do_swap
           , bool crossing_pruning
           , int do_min_embed )
{
    tabu_memory *memory = tabu_memory_new(H, min_avoid_pos, max_avoid_pos, min_xchge, max_xchge);
    tabu_vars vars;

    int current_iteration;

    GPtrArray *H_open_list;

    vertices_itr v_itr;
    vertex *v;

    /* The following code maps V(G) to the first |V(G)| vertices in V(H)

```

```

    for each mapped vertex  $v$   $H\_open \rightarrow data[ idx( v ) ] == true$ . Then a
    list,  $H\_open\_list$ , is constructed of all vertices in  $H$  for which
     $H\_open \rightarrow data[ idx( v ) ] == false$ .  $H\_open\_list$  is used to find
    vertices for which shift moves may be performed.
*/
{
    int i;
    int no_H_open_vertices;
    /* The set of vertices in  $H$  which are not images of vertices in  $G$  */
    GByteArray *H_open = g_byte_array_sized_new( graph_num_vertices( H ) );
    g_byte_array_set_size( H_open, graph_num_vertices( H ) );

    /* Initially no vertex in  $H$  is an image of a vertex in  $G$  */
    for ( i = 0; i < graph_num_vertices( H ); i++ )
    {
        H_open->data[ i ] = false;
    }

    /* Note which vertices in  $H$  are images of vertices in  $G$  */
    no_H_open_vertices = graph_num_vertices( H );
    for ( i = 0; i < graph_num_vertices( G ); i++ )
    {
        H_open->data[ idx( g_array_index( psi, vertex*, i ) ) ] = true;
        no_H_open_vertices -= 1;
    }

    /* Create the list for vertices which are not images */
    H_open_list = g_ptr_array_sized_new( no_H_open_vertices );

    /* Populate the list with vertices in  $H$  for which
     $H\_open \rightarrow data[ idx( v ) ] == false$ 
    */
    v_itr = graph_vertices( H );
    while ( ( v = vertices_itr_next( &v_itr ) ) != NULL )
    {
        if ( H_open->data[ idx( v ) ] == false )
        {
            g_ptr_array_add( H_open_list, v );
        }
    }

    g_byte_array_free( H_open, 1 );
}

/* Make  $H$  bidirected as required by Algorithm 6.2
graph_make_bidirected( H );

vars.G = G;
vars.H = H;
vars.psi = psi;
mpz_init_set( vars.lower_G, lower_G );
mpz_init_set( vars.lower_H, lower_H );
mpz_init( vars.tst_crossing_no );
mpz_init( vars.candidate_crossing_no );
vars.best_result = result_new( psi );
vars.crossing_pruning = crossing_pruning;
vars.do_min_embed = do_min_embed;

lower_bound( vars.tst_crossing_no, G, H, psi, lower_G, lower_H
, crossing_pruning, do_min_embed );
if ( mpz_cmp( vars.tst_crossing_no, vars.lower_H ) > 0 )

```

```

mpz_set( vars.lower_H, vars.tst_crossing_no );

for ( current_iteration = 0; current_iteration < iterations; current_iteration++ )
{
    int tst_tabu_value;

    vars.candidate_tabu_value = INT_MAX;
    /* Set the candidate crossing number to a very low value */
    mpz_set_str( vars.candidate_crossing_no, "-99999999999999999999999999999999", 10 );
    vars.improving_enum_mode = -1;

    /* For each vertex v in G */
    v_itr = graph_vertices( G );
    while ( ( v = vertices_itr_next( &v_itr ) ) != NULL )
    {
        int i;
        /* Determine whether H has any vertices that are not images
           of vertices in G, and enumerate this list
        */
        for ( i = 0; i < H_open_list->len; i += 1 )
        {
            /* Shift the mapping of v to the vertex at index i in H_open */
            SWAP_PTR( g_array_index( psi, vertex*, idx( v ) ), H_open_list->pdata[ i ] );

            /* Compute the tabu value for this shift */
            tst_tabu_value
                = tabu_memory_tabu_value( memory, g_array_index( psi, vertex*, idx( v ) )
                                          , (vertex*) H_open_list->pdata[ i ] );

            /* Evaluate the quality of the vertex mapping, and
               record its details if it improves upon the best known
               lower bound
            */
            evaluate_tabu( &vars, idx( v ), i, tst_tabu_value, DO_MOVE );

            /* Reset the vertex mapping to its prior state */
            SWAP_PTR( g_array_index( psi, vertex*, idx( v ) ), H_open_list->pdata[ i ] );
        }
    }

    if ( do_swap )
    {
        int i, j;

        /* For all pairs of vertex indices in G */
        for ( i = 0; i < graph_num_vertices( G ) - 1; i += 1 )
        {
            for ( j = i + 1; j < graph_num_vertices( G ); j += 1 )
            {
                /* Avoid the case where a vertex is swapped with itself */
                if ( i == j ) continue;

                /* Swap the mappings of v_i and v_j */
                SWAP_PTR( g_array_index( psi, vertex*, i ), g_array_index( psi, vertex*, j ) );

                /* Compute the tabu value for this shift */
                tst_tabu_value =
                    tabu_memory_tabu_value( memory, g_array_index( psi, vertex*, i )
                                          , g_array_index( psi, vertex*, j ) )
                    + tabu_memory_tabu_value( memory, g_array_index( psi, vertex*, j )
                                          , g_array_index( psi, vertex*, i ) );

                /* Evaluate the quality of the vertex mapping, and
            */

```

```

        record its details if it improves upon the best known
        lower bound
        */
        evaluate_tabu( &vars, i, j, tst_tabu_value, DO_SWAP );

        /* Swap the mappings of v_i and v_j */
        SWAP_PTR( g_array_index(psi, vertex*, i), g_array_index(psi, vertex*, j) );
    }
}

tabu_memory_decrease_tenure( memory );

/* If the most improving move was a shift move... */
if ( vars.improving_enum_mode == DO_MOVE )
{
    tabu_memory_insert( memory, g_array_index( psi, vertex*, vars.from_vertex_idx )
        , (vertex*) H_open_list->pdata[ vars.to_vertex_idx ] );

    /* Update the vertex mapping by shifting */
    SWAP_PTR( g_array_index( psi, vertex*, vars.from_vertex_idx )
        , H_open_list->pdata[ vars.to_vertex_idx ] )
}

/* If the most improving move was a swap move... */
else if ( vars.improving_enum_mode == DO_SWAP ) /* vars.improving_enum_mode == DO_SWAP */
{
    tabu_memory_insert( memory, g_array_index( psi, vertex*, vars.vertex_a_idx )
        , g_array_index( psi, vertex*, vars.vertex_b_idx ) );
    tabu_memory_insert( memory, g_array_index( psi, vertex*, vars.vertex_b_idx )
        , g_array_index( psi, vertex*, vars.vertex_a_idx ) );

    /* Update the vertex mapping by swapping */
    SWAP_PTR( g_array_index( psi, vertex*, vars.vertex_a_idx )
        , g_array_index( psi, vertex*, vars.vertex_b_idx ) );
}
}

tabu_memory_free( memory );
g_ptr_array_free( H_open_list, 1 );

result_extract( vars.best_result, psi );
mpz_set( bound, vars.lower_H );

mpz_clear( vars.lower_G );
mpz_clear( vars.lower_H );
mpz_clear( vars.tst_crossing_no );
mpz_clear( vars.candidate_crossing_no );
result_free( vars.best_result );
}

```

## B.4 Garey-Johnson

A C++ implementation of the Garey-Johnson algorithm is provided in this section. The algorithm implements the concept of independent crossing sets, combined with partial verification, as described in § 5.2.4.3.

### B.4.1 GareyJohnson

This algorithm is an implementation of Algorithm 5.4. It contains a rather great deal of code before the line number annotation `/*1*/`. This code is concerned with the allocation of memory on the stack for the various structures used by the algorithm. This scheme was used because allocation of such memory occurs in constant time, as opposed to dynamic memory allocation, which may sometimes be quite slow. The reader may wish to ignore most of the code before the line number annotation `/*1*/`, since this will not hamper understanding of the code. The Garey-Johnson algorithm is described in § 5.2, and it is recommended that the reader refer to this section to clarify the code of this implementation.

```

bool
garey_johnson( leda_graph& G, int k, int cull_complete )
{
    edge_pair_vec          lst_edge_pairs;
    edge_pair_itr_vec      selected_crossings;
    L_vec                  L( G );
    leda_node_array< carray< int > > v_deg_list( G );
    bool                   result          = false;
    size_t                 no_edges        = G.number_of_edges();
    size_t                 no_vertices     = G.number_of_nodes();

    STACK_ALLOC( leda_edge,    L_storage,  SQR( no_edges ) );
    STACK_ALLOC( int,          v_deg_s,    SQR( no_vertices ) );
    STACK_ALLOC( leda_edge,    out_edge_s, no_vertices );
    STACK_ALLOC( crossing_set*, c_set_s,   no_edges );
    STACK_ALLOC( leda_edge,    edges_s,    no_edges );

    crossing_set_vec c_sets( c_set_s, no_edges );

    G.make_undirected();

    /* Construct a list of all edge pairs of G, from which crossing
       pairs will be selected */
    for ( leda_edge e = G.first_edge(); e != G.last_edge(); e = G.succ_edge( e ) )
    {
        for ( leda_edge f = G.succ_edge( e ); f != leda_nil; f = G.succ_edge( f ) )
        {
            lst_edge_pairs.push_back( make_pair( e, f ) );
        }
    }

    /* If the symmetry culling for complete graphs is enabled, memory
       needs to be allocated for the structures used in the symmetry
       testing */
    if ( cull_complete )
    {
        size_t offset = 0;
        leda_node v;
        forall_nodes( v, G )
        {
            v_deg_list[ v ].init( v_deg_s + offset, no_vertices );
            offset += no_vertices;
        }
    }

    /* Initialize the memory to be used by the edge vector L */
    size_t offset = 0;
    forall_edges( e, G )

```

```

    {
        // Assign a chunk of raw memory for each L[ e ]
        L[ e ].init( L_storage + offset, no_edges );
        // Adjust the offset of raw memory beyond the number
        // of edges in L[ e ]
        offset += no_edges;
    }

    selected_crossings.resize( k );
    make_choose( selected_crossings.begin(), selected_crossings.end(), lst_edge_pairs.begin() );
/*1*/
    do
    {
        leda_edge e;
/*2*/ forall_edges( e, G )
        {
/*3*/     L[ e ].clear();
        }

/*5*/ for ( edge_pair_itr_vec::iterator itr = selected_crossings.begin()
           ; itr != selected_crossings.end(); ++itr )
        {
/*6*/     L[ (*itr)->first ].push_back( (*itr)->second );
/*7*/     L[ (*itr)->second ].push_back( (*itr)->first );
        }

    /* The following code implements the symmetry considerations for
       multipartite graph discussed in 5.2.4.1, applied only to
       complete graphs. Essentially what it does, is to construct a
       list l[v] for each vertex v, containing the degrees of its
       outgoing vertices. All such lists are sorted in descending
       order. For the list of vertices v_1, v_2, ..., v_n, it is
       then determined whether l[v_1] compares lexicographically
       greater than or equal to l[v_2], which must in turn compare
       lexicographically greater than or equal to l[v_3] and so
       forth. If this condition is violated, the current loop is
       simply skipped, so that the next crossing configuration may
       be tested.
    */
    if ( cull_complete )
    {
        bool continue_main = false;
        carray< leda_edge > out_edges( out_edge_s, no_vertices );

        // Clear the degree lists of each vertex in G
        leda_node v;
        forall_nodes( v, G )
        {
            v_deg_list[ v ].clear();
        }

        leda_edge e;
        v = G.first_node();

        // Construct the degree list for the first vertex ...
        forall_adj_edges( e, v )
        {
            v_deg_list[ v ].push_back( L[ e ].size() );
        }
        // ... and sort the degree list
        sort( v_deg_list[ v ].begin(), v_deg_list[ v ].end(), greater< int >() );
    }

```

```

// enumerate each vertex v beyond the first vertex
for (leda_node v = G.succ_node(G.first_node()); v != leda_nil; v = G.succ_node( v ))
{
    leda_edge e;
    // Construct the degree list for v...
    forall_adj_edges( e, v )
    {
        v_deg_list[ v ].push_back( L[ e ].size() );
    }
    // ... and sort the degree list
    sort( v_deg_list[ v ].begin(), v_deg_list[ v ].end(), greater< int >() );

    /* lexicographically compare the degree list of v to the
       degree list of the predecessor vertex of v in G
    */
    leda_node pred_v = G.pred_node( v );
    if ( lexicographical_compare(v_deg_list[pred_v].begin(), v_deg_list[pred_v].end()
                                , v_deg_list[v].begin(), v_deg_list[v].end() ) )
    {
        // Note that the current iteration in main loop must be skipped
        continue_main = true;
        // and break out of this inner loop
        break;
    }
}

// If current iteration in main loop must be skipped, then do so
if ( continue_main )
    continue;
}

// Clear the contents of c_sets
c_sets.clear();
/*9,10*/
construct_crossing_sets( G, L, c_sets );

/*11*/
if ( PLANAR( G ) )
{
    // Expand the first crossing set
    /*12,13*/ c_sets.front()->expand();

    /*14*/ if ( test_planar( L, c_sets.front()->edges.begin(), c_sets.front()->edges.end()
                            , c_sets.begin(), c_sets.end(), G, no_edges ) )
    {
        /*15*/ result = true;
        /*15*/ break;
    }
}

// Delete all crossing sets for current crossing configuration
for_each( c_sets.begin(), c_sets.end(), del_ptr< crossing_set > );
}
while(next_choose(selected_crossings.begin(),selected_crossings.end(),lst_edge_pairs.end()));

forall_edges( e, G )
{
    if ( G.is_hidden( e ) )
        G.del_edge( e );
}
}

```



```

    return result;
}

```

## B.4.2 ConstructGraph and TestPlanar

The algorithms in this section implement Algorithm 5.5 (TestPlanar’) and Algorithm 5.3 (ConstructGraph). However, the implementations have been “factored” in a different way, so that the function `construct_graph_and_verify_planarity` contains the functionality of Algorithm 5.3, as well as some functionality from Algorithm 5.5, whilst the function `test_planar` performs part of the execution of Algorithm 5.5.

It is therefore indicated by large source comment headings that the first part of `construct_graph_and_verify_planarity` implements Algorithm 5.3 and that the second part implements some functionality from Algorithm 5.5. The source code of `test_planar` is annotated with the line numbers of the functionality in Algorithm 5.5 that it implements.

```

bool
construct_graph_and_verify_planarity( L_vec& L
                                     , crossing_set_vec::iterator c_set
                                     , crossing_set_vec::iterator end_c_set
                                     , leda_graph& G
                                     , size_t no_edges )
{
    size_t max_edges = SQR( (*c_set)->edges.size() );
    leda_node cross_to_vertex[ no_edges ][ no_edges ];

    STACK_ALLOC( leda_node, cross_vertices_s, max_edges );
    STACK_ALLOC( leda_edge, hidden_edges_s, max_edges );

    carray< leda_node > cross_vertices( cross_vertices_s, max_edges );
    carray< leda_edge > hidden_edges( hidden_edges_s, max_edges );

    /* The execution of the following code has the effect of constructing
       a subgraph with artificial vertices representing crossings in G.
       Unlike the pseudo-code for the Garey-Johnson algorithm in Chapter
       5, no new graph, G’, is constructed, instead, the graph G is itself
       modified. To achieve this, the edges involved in the crossing are
       hidden, and stored in a list hidden_edges, which will allow them to
       be restored when construct_graph_and_verify_planarity terminates.
       The vertices belonging to the subgraph to be
       constructed within G are stored in the list cross_vertices, so that
       they may be deleted when construct_graph_and_verify_planarity
       terminates.
    */

    //////////////////////////////////////
    // Algorithm 5.3 (ConstructGraph)
    //////////////////////////////////////

    /*1*/
    // This entire loop constructs the bijection of line 1
    for (edge_list::iterator itr = (*c_set)->edges.begin(); itr != (*c_set)->edges.end(); ++itr)
    {
        for(carray<leda_edge>::iterator e_itr = L[*itr].begin(); e_itr != L[*itr].end(); ++e_itr)
        {
            if ( index( *itr ) < index( *e_itr ) )

```

```

        {
            leda_node v = G.new_node();
            cross_to_vertex[ index( *itr ) ][ index( *e_itr ) ] = v;
            cross_to_vertex[ index( *e_itr ) ][ index( *itr ) ] = v;
            cross_vertices.push_back( v );
        }
    }
}

/*2*/
for (edge_list::iterator itr = (*c_set)->edges.begin(); itr != (*c_set)->edges.end(); ++itr)
{
    leda_edge e = *itr;
/*3*/ leda_node x = G.source( e );

/*4*/ for(carray<leda_edge>::const_iterator e_itr = L[e].begin(); e_itr != L[e].end(); ++e_itr)
    {
/*5*/     leda_node w = cross_to_vertex[ index( e ) ][ index( *e_itr ) ];
/*6*/     G.new_edge( x, w );
/*7*/     x = w;
    }

/*9*/ G.new_edge( x, G.target( e ) );

/*10*/
    G.hide_edge( e );
    hidden_edges.push_back( e );
}

////////////////////////////////////
// Lines 9--21 of Algorithm 5.5 (TestPlanar')
////////////////////////////////////

/*9*/
if ( PLANAR( G ) )
{
/*11*/
    ++c_set;
/*10*/
    if ( c_set != end_c_set )
    {
/*12*/     (*c_set)->expand();
/*13*/     if ( test_planar( L, (*c_set)->edges.begin(), (*c_set)->edges.end()
        , c_set, end_c_set, G, no_edges ) )
        {
/*14*/         return true;
        }
/*15*/     else
        {
/*16*/         (*c_set)->contract();
            goto false_result;
        }
    }
    else
    {
/*19*/     return true;
    }
}

false_result:
/* Restore the edges that were hidden */

```



```

for_each( hidden_edges.begin(), hidden_edges.end()
          , bind( &leda_graph::restore_edge, &G, _1 ) );

/* Delete the artificial vertices that represented crossings */
for_each( cross_vertices.begin(), cross_vertices.end()
          , bind( &leda_graph::del_node, &G, _1 ) );

return false;
}

```

### B.4.3 TestPlanar

```

bool
test_planar( L_vec&                L
            , edge_list::iterator  current_edge
            , edge_list::iterator  end_edge
            , crossing_set_vec::iterator current_crossing_set
            , crossing_set_vec::iterator end_crossing_set
            , leda_graph&          G
            , size_t                no_edges )
{
/*1*/
  sort( L[ *current_edge ].begin(), L[ *current_edge ].end(), cmp_edges );

/*2*/
  do
  {
/*3*/ if ( util::next( current_edge ) != end_edge )
      {
/*4*/   if ( test_planar( L, util::next( current_edge ), end_edge
                        , current_crossing_set, end_crossing_set, G, no_edges ) )
          {
/*5*/     return true;
          }
      }
}
/*7*/ else
  {
    /* Some of the code present in Algorithm 5.5 (TestPlanar'),
       has been placed in the routine construct_graph_and_verify_planarity.
       The following code is functionally equivalent to the actions
       performed in lines 9--21 of Algorithm 5.5. */
    if ( construct_graph_and_verify_planarity( L, current_crossing_set
                                              , end_crossing_set, G, no_edges ) )
        {
          return true;
        }
  }
}
while (next_permutation( L[ *current_edge ].begin(), L[ *current_edge ].end(), cmp_edges ));
/*25*/
return false;
}

```

## B.5 Miscellaneous algorithms

This section contains implementations that do not fit into the other categories. Both of the implementations in this section are in the Python language.

### B.5.1 The spine subdivision algorithm

The function `sub_divide_on_spine` implements an algorithm which accomplishes the subdivision construction described in Theorem 5.3.1. The algorithm constructs a new spine, stored in the list `new_spine`, from the existing spine. The spine is enumerated from left to right, and for each vertex  $v$  in the spine, all subdivision vertices to the left of  $v$  (*i.e.*, the subdivision vertices of edges incident to  $v$  in  $\mathcal{G}$  of which the opposite vertices occur to the left of  $v$  in the spine and which are drawn on the lower page) are inserted into the list `new_spine`, followed by  $v$  itself and finally by the subdivision vertices which occur to the right of  $v$  in the spine (*i.e.*, the subdivision vertices of edges incident  $v$  in  $\mathcal{G}$  of which the opposite vertices occur to the right of  $v$  in the spine and which are drawn on the upper page). The full operation of the algorithm is described in the numerous source code comments.

```
def sub_divide_on_spine( G, cr_G, spine, spine_inv, page, e_map, n ):
    G.make_undirected()
    E = list( G.edges() )
    card_V_G = G.num_vertices()
    out_edges = {}
    for v in G.vertices():
        out_edges[ v ] = list( G.out_edges( v ) )

    new_spine = []
    new_page = {}

    for v in spine:
        def rev_outgoing_edge_order( e, f ):
            return cmp( spine_inv[ G.opposite( f, v ) ], spine_inv[ G.opposite( e, v ) ] )

        # First find all edges of which the opposite vertex occurs to the left of v in the spine.
        left_down = [ e for e in out_edges[v] if spine_inv[ G.opposite( e, v ) ] < spine_inv[v] ]
        # Then, filter out all edges that do not occur on the lower page
        left_down = [ e for e in left_down if e_map[ e ] != None and page[ e_map[ e ] ] == 1 ]
        # Sort the edges so that those joining vertices that are furthest away come first
        # since the subdivision vertices for those edges must be placed on the spine first
        left_down.sort( rev_outgoing_edge_order )

        # First find all edges of which the opposite vertex occurs to the right of v in the spine.
        right_up = [ e for e in out_edges[v] if spine_inv[ G.opposite( e, v ) ] > spine_inv[v] ]
        # Then, filter out all edges that do not occur on the upper page
        right_up = [ e for e in left_down if e_map[ e ] == None or page[ e_map[ e ] ] == 0 ]
        # Sort the edges so that those joining vertices that are closest away come first
        # since the subdivision vertices for those edges must be placed on the spine first
        right_up.sort( rev_outgoing_edge_order )

        # for each left going edge e
        for e in left_down:
            # subdivide e, obtaining the vertex list, v_list, and the edge
            # list, e_list of the path generated from subdivision of e
            v_list, e_list = do_sub( G, e, n )
            # if the order of the vertex list, v_list is such that v is at
            # the first entry, reverse both the vertex list and the edge list
```

```

# so as to obtain a left-to-right order.
if v_list[ 0 ] == v:
    v_list.reverse()
    e_list.reverse()

# extend the new spine with the subdivision vertices of e
for u in v_list[ 1:-1 ]:
    new_spine.append( u )

# set the pages of all subdivision edges to the upper page
for f in e_list:
    new_page[ f ] = 0

# except for the first subdivision edge, which should be
# drawn on the lower page
new_page[ e_list[ 0 ] ] = 1

# Add the vertex v to the new spine
new_spine.append( v )

# for each right going edge e
for e in right_up:
    # subdivide e, obtaining the vertex list, v_list, and the edge
    # list, e_list of the path generated from subdivision of e
    v_list, e_list = do_sub( G, e, n )
    # if the order of the vertex list, v_list is such that v is not at
    # the first entry, reverse both the vertex list and the edge list
    # so as to obtain a left-to-right order.
    if v_list[ 0 ] != v:
        v_list.reverse()
        e_list.reverse()

    # extend the new spine with the subdivision vertices of e
    for u in v_list[ 1:-1 ]:
        new_spine.append( u )

    # set the pages of all subdivision edges to the lower page
    for f in e_list:
        new_page[ f ] = 1

    # except for the last subdivision edge, which should be
    # drawn on the upper page
    new_page[ e_list[ -1 ] ] = 0

# remove the old non-subdivided edges of G
for e in E:
    G.remove_edge( e )

# initialize space in the C data structure, spine, to accommodate all the
# new added subdivision vertices.
spine.resize( len( new_spine ) )
# re-initialize the vertex to spine index map, spine_inv, with G,
# meaning that it will contain a mapping for each vertex in G
spine_inv.init( G )

# fill the spine with the contents of new_spine and fill the entries
# in spine_inv
for i in xrange( 0, len( new_spine ) ):
    spine[ i ] = new_spine[ i ]
    spine_inv[ spine[ i ] ] = i

```

```

# re-initialize the edge to vertex map, which is of the type e_map: E(G) -> V(cr_G)
e_map.init( G )
# clear the intersection graph cr_G
cr_G.clear()
# reconstruct the intersection graph, cr_G, to correspond to the
# vertex arrangement and edge layout of the new spine
c_algorithms.build_intersection_graph( G, cr_G, spine, spine_inv, e_map )
# re-initialize the crossing vertex to page map
page.resize( G.num_edges() )

# refill the entries in page, using the information from the construction
for e in G.edges():
    # Crossing vertices with degree 0 are deleted from cr_G, so as to speed up
    # the enumeration of V(cr_G), which is frequently performed by
    # edge layout algorithms. When such a vertex does not exist, e_map[ e ] == None
    if e_map[ e ] != None:
        page[ e_map[ e ] ] = new_page[ e ]

```

## B.5.2 The graph planarization algorithm

The code in this section is an implementation of Algorithm 6.9 (`ComputePlanarOrderings`). The utility function `get_min_max` returns a tuple containing the minimum and maximum indices of the incident vertices of an edge in the spine, in that order. The second utility function, `edge_pair`, returns a tuple of a pair of edges such that the first tuple element contains the edge with the smallest index in  $G$ .

This implementation largely follows Algorithm 6.9, but there are a few points to clarify. Firstly, the computation of the left-going and right-going edges is made explicit in this algorithm. The left- and right-going edges for an edge  $e$  are computed by enumerating each vertex  $v$  that occurs between the incident vertices of  $e$  in the spine, and by testing whether each edge incident to  $v$  is left-going or right-going. The appropriate list, L or R is extended, depending on the situation.

Comparison functions were provided for the sorting that was applied to L and to R. In Python, a comparison function returns -1 if its first argument is smaller than its second, 0 if they are equal, and 1 otherwise.

An implementation of Algorithm 5.3 (`ConstructGraph`) is also present in the implementation of Algorithm 6.9. Its start is marked with a prominent source code comment, and the line number annotations correspond to the line numbers of Algorithm 5.3. The usage of the utility function `edge_pair` is apparent here. The mapping  $\phi$  should be indexed by unordered edge pairs. These pairs were represented as tuples in Python. Tuples are however ordered and this problem was avoided by ensuring that the same tuple order was always used, which was ensured by `edge_pair`.

```

def construct_planar( G, spine, spine_inv, e_map, page ):
    def get_min_max( e ):
        left_e = spine_inv[ G.source( e ) ]
        right_e = spine_inv[ G.target( e ) ]

        if left_e > right_e:
            left_e, right_e = right_e, left_e

        return left_e, right_e

    def edge_pair( e, f ):
        if int( e ) > int( f ):

```

```

    return f, e
else:
    return e, f

# 1
for e in G.edges():
    # This is true if the crossing vertex was isolated in the
    # intersection graph - i.e. if e is not crossed
    if e_map[ e ] == None:
        continue

    page_e = page[ e_map[ e ] ]

    left_index, right_index = get_min_max( e )

    L = {}
    R = {}
    # For each vertex in the spine between the incident vertices of e
    for v in spine[ left_index + 1 : right_index - 1 ]:
        # for all edges incident to v
        for f in chain( G.out_edges( v ), G.in_edges( v ) ):
            if e_map[ f ] != None and page[ e_map[ f ] ] == page_e:
                # if f is a left-going edge
                if spine_inv[ G.opposite( f, v ) ] < left_index:
# 2
                    L.append( v )
                # if f is a right-going edge
                elif spine_inv[ G.opposite( f, v ) ] > right_index:
# 3
                    R.append( v )

    # define the sorting order for edges in L, as in Algorithm 6.9
    def sort_L( e, f ):
        left_e, right_e = get_min_max( e )
        left_f, right_f = get_min_max( f )

        # Sort primarily in ascending order of the highest spine position
        if right_e < right_f:
            return -1
        elif right_e == right_f:
            # Sort secondarily in descending order of the lowest spine position
            if left_e > left_f:
                return -1
            elif left_e == left_e:
                return 0
            else:
                return 1
        else:
            return 0

    # define the sorting order for edges in R, as in Algorithm 6.9
    def sort_R( e, f ):
        left_e, right_e = get_min_max( e )
        left_f, right_f = get_min_max( f )

        # Sort primarily in ascending order of the lowest spine position
        if left_e < left_f:
            return -1
        elif left_e == left_f:
            # Sort secondarily in ascending order of the highest spine position
            if right_e < right_f:

```

```

        return -1
    elif left_e == left_e:
        return 0
    else:
        return 1
    else:
        return 0
# 4
L.sort( sort_L )
# 5
R.sort( sort_R )

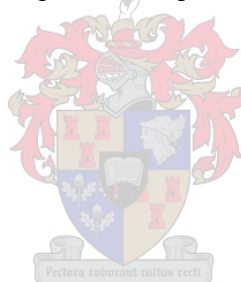
cross[ e ] = {}
# 6,7
for f in chain( L, R ):
# 9
    cross[ e ].append( f )

#####
# ConstructGraph
#####

phi = {}
# 1
# Construct a mapping by associating each vertex pair
# with an artificial vertex representing a crossing
for e in G.edges():
    for f in cross[ e ]:
        v = G.add_vertex()
        phi[ edge_pair( e, f ) ] = v

# 2
for e in G.edges():
# 3
    x = G.source( e )
# 4
    for f in cross[ e ]:
# 5
        w = phi[ edge_pair( e, f ) ]
# 6
        G.add_edge( x, w )
# 7
        x = w
# 9
        G.add_edge( x, G.target( e ) )
# 10
        G.remove_edge( e )

```



### B.5.3 A subgraph testing algorithm for complete multipartite graphs

The implementation described in this section determines whether a complete multipartite graph  $\mathcal{G}$  on  $n$  vertices is a subgraph of a complete multipartite graph  $\mathcal{H}$  on  $n$  vertices. If  $\mathcal{G}$  has  $n - 1$  vertices, then each of the vertices of  $\mathcal{H}$  may be suppressed, one at a time, and for each graph  $\mathcal{H}'$  resulting from the suppression of a vertex, it may be tested whether  $\mathcal{G}$  is a subgraph of  $\mathcal{H}'$ . Of course, if  $\mathcal{G}$  has less than  $n - 1$  vertices, this procedure may be generalized appropriately. However, such a generalization was not necessary for the purposes of this thesis. This is true, because *all* complete multipartite graphs of orders 6–13 were generated and for each graph of order  $n > 6$ , all subgraphs of order  $n - 1$  were found in addition to all subgraphs of order  $n$ .



Thus, for a complete multipartite graph of order  $n$ , all subgraphs of orders less than  $n$  may be found by considering the subgraphs of the subgraphs of the graph to the necessary depth.

A complete multipartite subgraph  $\mathcal{G}$  on  $n$  vertices may be obtained from a complete multipartite subgraph  $\mathcal{H}$  on  $n$  vertices by choosing a pair of partite sets  $P_1$  and  $P_2$  in  $\mathcal{H}$ , and by deleting *all* edges joining  $P_1$  to  $P_2$ , so as to form a new partite set  $P = P_1 \cup P_2$ . If some, but not all edges between  $P_1$  and  $P_2$  are deleted, the resulting graph will not be a complete multipartite graph. The operation of removing edges joining partite sets may be applied to various pairs of partite sets in  $\mathcal{H}$  to obtain different complete multipartite subgraphs. Clearly this is the only method to generate complete multipartite subgraphs from a complete multipartite graph (since otherwise, there will be some pair of partite sets of which not all joining edges will be deleted).

If the edges between the partite sets  $P_1$  and  $P_2$  above are deleted, it trivially follows that  $|P| = |P_1| + |P_2|$ , and therefore if  $\mathcal{H} \cong K_{|P_1|, |P_2|, \dots, |P_t|}$ , then  $\mathcal{G} \cong K_{|P_1|+|P_2|, \dots, |P_{t-1}|}$ . If the edges joining multiple pairs of partite sets in  $\mathcal{H}$  are deleted, then the cardinalities of the partite sets in  $\mathcal{G}$  are the sums of the cardinalities of the partite sets in  $\mathcal{H}$ . More formally, if  $\mathcal{G} \cong K_{n_1, n_2, \dots, n_i}$  is a complete multipartite subgraph of a graph  $\mathcal{H} \cong K_{m_1, m_2, \dots, m_j}$ , then  $\mathcal{G}$  may be obtained from  $\mathcal{H}$  by the deletion of edges joining  $j - i$  pairs of partite sets in  $\mathcal{H}$  and the partite set cardinalities of  $\mathcal{H}$  may be partitioned into sets  $M_1, M_2, \dots, M_i$  such that

$$\sum_{m \in M_k} m = n_k, \quad 1 \leq k \leq i.$$

Thus, the problem of determining whether a complete multipartite graph  $\mathcal{G}$  is a subgraph of a complete multipartite graph  $\mathcal{H}$  is the same problem as determining whether a partitioning of the cardinalities of the partite sets of  $\mathcal{H}$  exists such the sum of the cardinalities in each partition corresponds to a distinct partite set cardinality in  $\mathcal{G}$ . This problem may be reinterpreted as a *bin packing problem*, where each of the partite set cardinalities  $m_s$  of  $\mathcal{G}$  corresponds to a bin with capacity  $m_s$ , with  $1 \leq s \leq i$ , and where each of the partite set cardinalities  $n_t$  of  $\mathcal{H}$  corresponds to an item of size  $n_t$ , with  $1 \leq t \leq j$ . The question is then whether a packing of the  $j$  items into the  $i$  bins exists, such that each bin is filled exactly to capacity.

No efficient algorithm for solving this problem is known. It may be solved by a simple brute force, recursive backtracking algorithm. The idea is simple: the first item  $n_1$ , is placed into the first bin with sufficient capacity to hold it. All possible packings of the remaining items (besides  $n_1$ ) into bins are considered. If no feasible packing is found,  $n_1$  is placed into the next bin with sufficient capacity, and all possible packings for the remaining items are attempted. The recursive nature of the algorithm is due to the fact that the process performed for  $n_1$  is performed for every item.

In the implementation, **sub** represents an array of bin capacities and **super** represents an array of items. If an item at index  $i$  in **super** is placed into a bin at index  $j$ , the capacity of the bin is updated by the code `sub[i] -= super[j]`.

The algorithm is called recursively to consider each consecutive item — for each invocation, the parameter **super\_pos** is the index of the item to be examined. The parameter **super\_len** is the length of the array **super** — in other words, it is equal to the number of items to be packed into bins. If `super_pos >= super_len`, as tested in the first line of the algorithm, then all items were successfully packed into bins. Thus, the algorithm returns 1 (“TRUE”) when this occurs. Otherwise, `super[super_pos]` is an item that has not yet been packed into a bin. All bins are enumerated in the **for** loop, and if a bin with sufficient capacity for holding `super[super_pos]` is found, it is placed into the bin. The algorithm is then called recursively. If no bin with sufficient capacity for holding `super[super_pos]` can be found, the algorithm returns 0 (“FALSE”) to indicate the situation. If an item before the current item was considered, it will be placed into

another bin and the algorithm will again be called recursively. If, on the other hand, `super_pos == 0`, then no feasible packing of items into bins could be found, and the algorithm returns “FALSE.”

```
int rec_subgraph_of(int *sub, int sub_len, int *super, int super_len, int super_pos)
{
    if (super_pos >= super_len) return 1;

    for (i = 0; i < sub_len; i++) {
        if (sub[i] >= super[super_pos]) {
            sub[i] -= super[super_pos];

            if (rec_subgraph_of(sub, sub_len, super, super_len, super_pos + 1)) {
                sub[i] += super[super_pos];
                return 1;
            }

            sub[i] += super[super_pos];
        }
    }

    return 0;
}
```



# Bibliography

- [ACNS82] M. Ajtai, V. Chvátal, M. Newborn, and E. Szemerédi. *Crossing-free subgraphs*, volume 60 of *North-Holland Math. Stud.*, pages 9–12. North-Holland, Amsterdam, 1982.
- [AG76] V.B. Alekseev and V.S. Gončakov. The thickness of an arbitrary complete graph. *Math. Sbornik*, 30(2):187–202, 1976.
- [AR88] D. Archdeacon and R.B. Richter. On the parity of crossing numbers. *J. Graph Theory*, 12(3):307–310, 1988.
- [ARR96] M. Anderson, R.B. Richter, and P. Rodney. The crossing number of  $C_6 \times C_6$ . In *Proceedings of the Twenty-seventh Southeastern International Conference on Combinatorics, Graph Theory and Computing (Baton Rouge, LA, 1996)*, volume 118, pages 97–107, 1996.
- [ARR97] M. Anderson, R.B. Richter, and P. Rodney. The crossing number of  $C_7 \times C_7$ . In *Proceedings of the Twenty-eighth Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 1997)*, volume 125, pages 97–117, 1997.
- [Asa86] K. Asano. The crossing number of  $\mathcal{K}_{1,3,n}$  and  $\mathcal{K}_{2,3,n}$ . *J. Graph Theory*, 10(1):1–8, 1986.
- [BH65] L.W. Beineke and F. Harary. The thickness of the complete graph. *Canad. J. Math.*, 17:850–859, 1965.
- [BHM64] L.W. Beineke, F. Harary, and J.W. Moon. On the thickness of the complete bipartite graph. *Proc. Cambridge Philos. Soc.*, 60:1–5, 1964.
- [BK64] J. Blažek and N. Koman. A minimal problem concerning complete plane graphs. In *Theory of Graphs and its Applications (ed. M. Fiedler)*, pages 113–117. Czechoslovak Academy of Sciences., Prague, 1964.
- [BK79] F. Bernhart and P.C. Kainen. The book thickness of a graph. *J. Combin. Theory Ser. B*, 27(3):320–331, 1979.
- [BL84] S.N. Bhatt and F.T. Leighton. A framework for solving VLSI graph layout problems. *J. Computer and System Sciences*, 28:300–343, 1984.
- [BML00] C. Buchheim, Jünger M., and S. Leipert. A fast layout algorithm for  $k$ -level graphs. In J. Marks, editor, *Graph Drawing 2000*, volume 1984, pages 29 – 240. Springer Verlag, 2000.

- [BR80] L.W. Beineke and R.D. Ringeisen. On the crossing numbers of products of cycles and graphs of order four. *J. Graph Theory*, 4(2):145–155, 1980.
- [Bra21] H.R. Brahana. Systems of circuits on two-dimensional manifolds. *Ann. of Math. (2)*, 23(2):144–168, 1921.
- [CAH34] Ch. Chojnacki (Aka. Hanani). Über wesentlich unplättbare Kurven im dreidimensionalen Ruame. *Fund. Math.*, 23:135–142, 1934.
- [CCDG82] P.Z. Chinn, J. Chvátalová, A.K. Dewdney, and N.E. Gibbs. The bandwidth problem for graphs and matrices—a survey. *J. Graph Theory*, 6(3):223–254, 1982.
- [Cim92] R. Cimikowski. Graph planarization and skewness. In *Proceedings of the Twenty-third Southeastern International Conference on Combinatorics, Graph Theory, and Computing (Boca Raton, FL, 1992)*, volume 88, pages 21–32, 1992.
- [Cim96] R. Cimikowski. Topological properties of some interconnection network graphs. *Congr. Numer.*, 121:19–32, 1996.
- [Cim98] R. Cimikowski. Crossing number bounds for the mesh of trees. In *Proceedings of the Twenty-ninth Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 1998)*, volume 134, pages 107–116, 1998.
- [Cim02] R. Cimikoswki. Algorithms for the fixed linear crossing number problem. *Discrete Applied Maths*, 122:93–115, 2002.
- [CLR87] F.R.K. Chung, F.T. Leighton, and A.L. Rosenberg. Embedding graphs in books: a layout problem with applications to VLSI design. *SIAM J. Alg. Disc. Meth.*, 8(1):33–59, January 1987.
- [CLR97] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1997.
- [CO93] G. Chartrand and R. Oelermann. *Applied Algorithmic Graph Theory*. McGraw-Hill, New York, 1993.
- [Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.
- [CS96] R. Cimikoswki and P. Shope. A neural network algorithm for a graph layout problem. *IEEE Transactions on neural networks*, 7(2):341–349, March 1996.
- [CSSV04] E. Czabarka, O. Sýkora, L. Székely, and I. Vrto. Biplanar crossing numbers I: A survey of results and problems, to appear. In T. Fleiner and G.O.H. Katona, editors, *Finite and Infinite Combinatorics (To appear)*. Akademia Kiado, Budapest, 2004.
- [CY94] F.R.K. Chung and S.T. Yau. A near optimal algorithm for edge separators. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 1–8, New York, NY, USA, 1994. ACM Press.
- [Dar59] C.R. Darwin. *On the Origin of Species; originally On the origin of species by means of natural selection*. Random House; originally published by J. Murray, London, 1859.
- [dKMP<sup>+</sup>04] E. de Klerk, J. Maharry, D.V. Pasechnik, R.B. Richter, and G. Salazar. Improved bounds for the crossing numbers of  $\mathcal{K}_{m,n}$  and  $\mathcal{K}_n$ . *Submitted*, 2004.

- [DR95] A.M. Dean and R.B. Richter. The crossing number of  $C_4 \times C_4$ . *J. Graph Theory*, 19(1):125–129, 1995.
- [EG70] R.B. Eggleton and R.K. Guy. The crossing number of the  $n$ -cube. *Notices Amer. Math. Soc.*, 17:757, 1970.
- [EG73] P. Erdős and R.K. Guy. Crossing number problems. *Amer. Math. Monthly*, 80:52–58, 1973.
- [EH01] D. Evans and S. Howard. *Introducing Evolution*. Icon Books, UK and Totem Books, USA, 2001.
- [EHK81] G. Exoo, F. Harary, and J. Kabell. The crossing numbers of some generalized Petersen graphs. *Math. Scand.*, 48(2):184–188, 1981.
- [Far48] I. Fary. On straight line representing of planar graphs. *Acta. Sci. Math.*, 11:229–233, 1948.
- [FdF00] L. Faria and C.M.H. de Figueiredo. On Eggleton and Guy’s conjectured upper bound for the crossing number of the  $n$ -cube. *Math. Slovaca*, 50(3):271–287, 2000.
- [FG02] S. Fiorini and J.B. Gauci. New results and problems on crossing numbers. *Rend. Sem. Mat. Messina Ser. II*, 24(8):29–47, 2002.
- [Fio86] S. Fiorini. On the crossing number of generalized Petersen graphs. In *Combinatorics ’84 (Bari, 1984)*, pages 225–241. North-Holland, Amsterdam, 1986.
- [FKRS02] A. Foley, R. Krieger, A. Riskin, and I. Stanton. The crossing numbers of some twisted toroidal grid graphs. *Bulletin of the ICA*, 36:80–88, 2002.
- [GCC] Free Software Foundation, GCC Home Page - GNU Project - Free Software Foundation (FSF). [Online], [cited February 23, 2005], Available from: <http://gcc.gnu.org>.
- [GH73] R.K. Guy and A. Hill. The crossing number of the complement of a circuit. *Discrete Math.*, 5:335–344, 1973.
- [GJ83] M.R. Garey and D.S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods*, 4(3):312–316, 1983.
- [GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.
- [Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Ops. Res.*, 5:533–549, 1986.
- [GMP] GNU, GMP: The GNU MP Bignum Library. [Online], [cited February 23, 2005], Available from: <http://www.swox.com/gmp/>.
- [GMS01] E. Garcia-Moreno and G. Salazar. Bounding the crossing number of a graph in terms of the crossing number of a minor with small maximum degree. *J. Graph Theory*, 36(3):168–173, 2001.
- [Guy69] R.K. Guy. The decline and fall of Zarankiewicz’s theorem. In *Proof Techniques in Graph Theory (Proc. Second Ann Arbor Graph Theory Conf., Ann Arbor, Mich., 1968)*, pages 63–69. Academic Press, New York, 1969.

- [Har69] F. Harary. *Graph theory*. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969.
- [Hea87] M.T. Heath, editor. *Hypercube multiprocessors 1987*, Philadelphia, PA, 1987. Society for Industrial and Applied Mathematics (SIAM).
- [Hef91] L. Heffter. Über das Problem der Nachbargebiete. *Math. Ann.*, 38:477–508, 1891.
- [HH99] F.C. Harris, Jr. and C.R. Harris. A proposed algorithm for calculating the minimum crossing number of a graph. In *Combinatorics, graph theory, and algorithms, Vol. I, II (Kalamazoo, MI, 1996)*, pages 469–478. New Issues Press, Kalamazoo, MI, 1999.
- [HHW88] F. Harary, J.P. Hayes, and H. Wu. A survey of the theory of hypercube graphs. *Computer and Mathematics with Applications*, 15(4):277–289, 1988.
- [HJR85] N. Hartsfield, B. Jackson, and G. Ringel. The splitting number of the complete graph. *Graphs Combin.*, 1(4):311–329, 1985.
- [HKS73] F. Harary, P.C. Kainen, and A.J. Schwenk. Toroidal graphs with arbitrarily high crossing numbers. *Nanta Math.*, 6(1):58–67, 1973.
- [HT74] J.E. Hopcroft and R.E. Tarjan. Efficient Planarity Testing. *J. ACM*, 21:549–568, 1974.
- [JLMO97] M. Jünger, E.K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In *Graph Drawing '97 (Proc.)*. Lecture Notes in Computer Science, Springer-Verlag, 1997.
- [JMOS98] M. Jünger, P. Mutzel, T. Odenthal, and M. Scharbrodt. The thickness of a minor-excluded class of graphs. *Discrete Math.*, 182(1-3):169–176, 1998. Graph theory (Lake Bled, 1995).
- [JR84] B. Jackson and G. Ringel. The splitting number of complete bipartite graphs. *Arch. Math. (Basel)*, 42(2):178–184, 1984.
- [JR85] B. Jackson and G. Ringel. Splittings of graphs on surfaces. In Frank Harary, editor, *Graphs and applications, Proceedings of the 1st Colorado Symposium on Graph Theory*, pages 203–219, Colorado, 1985. Boulder.
- [JŠ82] S. Jendrol' and M. Ščerbová. On the crossing numbers of  $\mathcal{S}_m \times \mathcal{P}_n$  and  $\mathcal{S}_m \times \mathcal{C}_n$ . *Časopis Pěst. Mat.*, 107(3):225–230, 307, 1982. With a loose Russian summary.
- [JS01] H.A. Juárez and G. Salazar. Drawings of  $\mathcal{C}_m \times \mathcal{C}_n$  with one disjoint family. II. *J. Combin. Theory Ser. B*, 82(1):161–165, 2001.
- [Kai72] P.C. Kainen. A lower bound for crossing numbers of graphs with applications to  $\mathcal{K}_n$ ,  $\mathcal{K}_{p,q}$ , and  $\mathcal{Q}(d)$ . *J. Combinatorial Theory Ser. B*, 12:287–298, 1972.
- [Kle70] D.J. Kleitman. The crossing number of  $\mathcal{K}_{5,n}$ . *J. Combinatorial Theory*, 9:315–323, 1970.
- [Kle91] M. Klešč. On the crossing numbers of Cartesian products of stars and paths or cycles. *Math. Slovaca*, 41(2):113–120, 1991.
- [Kle94] M. Klešč. The crossing numbers of products of paths and stars with 4-vertex graphs. *J. Graph Theory*, 18(6):605–614, 1994.

- [Kle95] M. Klešč. The crossing numbers of certain Cartesian products. *Discuss. Math. Graph Theory*, 15(1):5–10, 1995.
- [Kle96] M. Klešč. The crossing number of  $\mathcal{K}_{2,3} \times \mathcal{P}_n$  and  $\mathcal{K}_{2,3} \times \mathcal{S}_n$ . *Tatra Mt. Math. Publ.*, 9:51–56, 1996. Cycles and colourings '94 (Stará Lesná, 1994).
- [Kle99a] M. Klešč. The crossing number of  $\mathcal{K}_5 \times \mathcal{P}_n$ . *Tatra Mt. Math. Publ.*, 18:63–68, 1999. Cycles and colourings '97 (Stará Lesná).
- [Kle99b] M. Klešč. The crossing numbers of products of a 5-vertex graph with paths and cycles. *Discuss. Math. Graph Theory*, 19(1):59–69, 1999.
- [Kle01a] M. Klešč. The crossing numbers of Cartesian products of paths with 5-vertex graphs. *Discrete Math.*, 233(1-3):353–359, 2001. Graph theory (Prague, 1998).
- [Kle01b] M. Klešč. On the crossing numbers of products of stars and graphs of order five. *Graphs Combin.*, 17(2):289–294, 2001.
- [Kle02] M. Klešč. The crossing number of  $\mathcal{K}_{2,3} \times \mathcal{C}_3$ . *Discrete Mathematics*, 251:109–117, 2002.
- [KM04] P. Kolman and J. Matoušek. Crossing number, pair-crossing number, and expansion. *J. Combin. Theory Ser. B*, 92(1):99–113, 2004.
- [KRS96] M. Klešč, R.B. Richter, and I. Stobert. The crossing number of  $\mathcal{C}_5 \times \mathcal{C}_n$ . *J. Graph Theory*, 22(3):239–243, 1996.
- [Kur30] K. Kuratowski. Sur le problème des Courbes Gauches en Topologie. *Fund. Math.*, 15:271–283, 1930.
- [LED] Algorithmic Solutions Software, Algorithmic Solutions Software GmbH: LEDA. [Online], [cited February 23, 2005], Available from: <http://www.algorithmic-solutions.com>.
- [Lee00] J.M. Lee. *Introduction to Topological Manifolds*. Springer-Verlag New York, Inc., 2000.
- [Lei83] F.T. Leighton. *Complexity issues in VLSI*. MIT Press, Cambridge, 1983.
- [Lei84] F.T. Leighton. New lower bound techniques for VLSI. *Math. Systems Theory*, 17:47–70, 1984.
- [Lei92] F.T. Leighton. *Introduction to parallel algorithms and architectures*. Morgan Kaufmann, San Mateo, CA, 1992. Arrays, trees, hypercubes.
- [Lev73] L. Levin. Universal search problems (in Russian). *Problemy Peredachi Informatsii*, 3:115–116, 1973.
- [Lie01] A. Liebers. Planarizing graphs—a survey and annotated bibliography. *J. Graph Algorithms Appl.*, 5(1):74 pp. (electronic), 2001.
- [LS97] M. Lovrečić Saražin. The crossing number of the generalized Petersen graph  $\mathcal{P}(10, 4)$  is four. *Math. Slovaca*, 47(2):189–192, 1997.
- [Mad91] T. Madej. Bounds for the crossing number of the  $n$ -cube. *J. Graph Theory*, 15(1):81–97, 1991.

- [McK90] B.D. McKay. nauty user's guide (version 1.5), <http://cs.anu.edu.au/bdm/nauty>. Technical report, Australian National University, Department of Computer Science, 1990.
- [MM97] Jünger M. and P. Mutzel. 2-Layer straightline crossing minimization: performance of exact and heuristic algorithms. *J. Graph Algorithms Appl.*, 1(1):1–25, 1997.
- [MNKF90] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Trans. Comput.*, 39(1):124–127, 1990.
- [MOS98] P. Mutzel, T. Odenthal, and M. Scharbrodt. The thickness of graphs: a survey. *Graphs Combin.*, 14(1):59–73, 1998.
- [MR92] D. McQuillan and R.B. Richter. On the crossing numbers of certain generalized Petersen graphs. *Discrete Math.*, 104(3):311–320, 1992.
- [Mut01] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM J. Optimization*, 11(4):1065–1080, 2001.
- [Mye98] N.C. Myers. The crossing number of  $C_m \times C_n$ : a reluctant induction. *Math. Mag.*, 71(5):350–359, 1998.
- [Nah03] N.H. Nahas. On the crossing number of  $K_{m,n}$ . *Electron. J. Combin.*, 10:Note 8, 6 pp. (electronic), 2003.
- [Nic68] T.A.J. Nicholson. Permutation procedure for minimising the number of crossings in a network. *Proc. IEEE*, 115(1):21–26, 1968.
- [P76] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14:359–364, 1976.
- [Pac00] J. Pach. Crossing numbers. In *Discrete and computational geometry (Tokyo, 1998)*, pages 267–273. Springer, Berlin, 2000.
- [PSS96] J. Pach, F. Shahrokhi, and M. Szegedy. Applications of the crossing number. *Algorithmica*, 16(1):111–117, 1996.
- [PT97] J. Pach and G. Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, 1997.
- [PT98] J. Pach and G. Tóth. Which crossing number is it anyway? In *Proc. 39th Annual Symposium on Foundation of Computer Science*, pages 617–626. IEEE Press, Baltimore, 1998.
- [PT00] J. Pach and G. Tóth. Thirteen problems on crossing numbers. *Geombinatorics*, 9(4):194–207, 2000.
- [Pyt] Python Foundation, Python Programming Language. [Online], [cited February 23, 2005], Available from: <http://www.python.org>.
- [RB78] R.D. Ringeisen and L.W. Beineke. The crossing number of  $C_3 \times C_n$ . *J. Combin. Theory Ser. B*, 24(2):134–136, 1978.
- [Ree93] C.R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, Osney Mead, Oxford OX2 0EL, 1993.



- [Rin65] G. Ringel. Das Geschlecht des vollständigen paaren Graphen. *Abh. Math. Sem. Univ. Hamburg*, 38:139–150, 1965.
- [Rot84] J.J. Rotman. *An introduction to the theory of groups*. Allyn and Bacon, Inc., 1984.
- [RS01] R.B. Richter and G. Salazar. The crossing number of  $C_6 \times C_n$ . *Australas. J. Combin.*, 23:135–143, 2001.
- [RS02] R.B. Richter and G. Salazar. The crossing number of  $\mathcal{P}(N, 3)$ . *Graphs Combin.*, 18(2):381–394, 2002.
- [RT95] R.B. Richter and C. Thomassen. Intersection of curve systems and the crossing number of  $C_5 \times C_5$ . *Discrete Comp. Geom.*, 13:149–159, 1995.
- [Ru96] R.B. Richter and J. Širáň. The crossing number of  $\mathcal{K}_{3,n}$  in a surface. *Journal of Graph Theory*, 21(1):51–54, 1996.
- [RY68] G. Ringel and J.W.T. Youngs. Solution of the Heawood map-coloring problem. *Proc. Nat. Acad. Sci. U.S.A.*, 60:438–445, 1968.
- [Saa69] T.L. Saaty. Symmetry and the crossing number for complete graphs. *J. Res. Nat. Bur. Standards Sect. B*, 73B:177–186, 1969.
- [Sal99] G. Salazar. Drawings of  $C_m \times C_n$  with one disjoint family. *J. Combin. Theory Ser. B*, 76(2):129–135, 1999.
- [Sal04] G. Salazar. On the crossing numbers of loop networks and Generalized Petersen Graphs. *Discrete Mathematics*, to appear, 2004.
- [SG04] G. Salazar and L. Glebsky. The crossing number of  $C_m \times C_n$  is as conjectured for  $n \geq m(m + 1)$ . *J. Graph Theory*, 47(1):53–72, 2004.
- [Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, New York, 1997.
- [SSSV] F. Shahrokhi, L.A. Székely, O. Sýkora, and I. Vrřo. Crossing number of meshes. Proc. Intl. Symposium on Graph Drawings '95, Lecture Notes in Computer Science, Springer Verlag, Berlin, 1996, see also: Intersections of curves and crossing numbers of  $C_m \times C_n$  on surfaces, submitted.
- [SSSV94] F. Shahrokhi, L.A. Székely, O. Sýkora, and I. Vrřo. Improved bounds for the crossing numbers on surfaces of genus  $g$ . In *Graph-theoretic concepts in computer science (Utrecht, 1993)*, pages 388–395. Springer, Berlin, 1994.
- [SSSV96a] F. Shahrokhi, O. Sýkora, L.A. Székely, and I. Vrřo. The crossing number of a graph on a compact 2-manifold. *Adv. Math.*, 123(2):105–119, 1996.
- [SSSV96b] F. Shahrokhi, L.A. Székely, O. Sýkora, and I. Vrřo. The book crossing number of a graph. *Journal of Graph Theory*, 21(4):413–424, 1996.
- [SSSV96c] F. Shahrokhi, L.A. Székely, O. Sýkora, and I. Vrřo. Drawings of graphs on surfaces with few crossings. *Algorithmica*, 16(1):118–131, 1996.
- [SSSV97a] F. Shahrokhi, O. Sýkora, L.A. Székely, and I. Vrřo. Crossing numbers: bounds and applications. In *Intuitive geometry (Budapest, 1995)*, pages 179–206. János Bolyai Math. Soc., Budapest, 1997.

- [SSSV97b] F. Shahrokhi, O. Sýkora, L.A. Székely, and I. Vrřo. On bipartite crossings, largest biplanar subgraphs, and the linear arrangement problem. In *On bipartite crossings, largest biplanar subgraphs, and the linear arrangement problem*, pages 55–68. Springer-Verlag, London, UK, 1997.
- [SSV95] F. Shahrokhi, L.A. Székely, and I. Vrřo. Crossing numbers of graphs, lower bound techniques and algorithms: a survey. In *Graph drawing (Princeton, NJ, 1994)*, pages 131–142. Springer, Berlin, 1995.
- [Sta78] S. Stahl. Generalized embedding schemes. *J. Graph Theory*, 2:41–52, 1978.
- [SU04] G. Salazar and E. Ugalde. An improved bound for the crossing number of  $C_m \times C_n$ : a self-contained proof using mostly combinatorial arguments. *Graphs and Combinatorics*, to appear, 2004.
- [SV93] O. Sýkora and I. Vrřo. On crossing numbers of hypercubes and cube connected cycles. *BIT*, 33(2):232–237, 1993.
- [SV94] O. Sýkora and I. Vrřo. On VLSI layout of the star graph and related networks. *Integration, the VLSI Journal*, 17:83–93, 1994.
- [SWI] The SWIG Team, Simplified Wrapper and Interface Generator. [Online], [cited February 23, 2005], Available from: <http://www.swig.org>.
- [Szé04] László A. Székely. A successful concept for measuring non-planarity of graphs: the crossing number. *Discrete Math.*, 276(1-3):331–352, 2004. 6th International Conference on Graph Theory.
- [Tar72] R. Tarjan. Sorting using networks of queues and stacks. *J. Assoc. Comput. Mach.*, 19:341–346, 1972.
- [Tho81] C. Thomassen. Kuratowski's theorem. *J. Graph Theory*, 5(3):225–241, 1981.
- [Tur77] P. Turán. A note of welcome. *J. Graph Theory*, 1:7–9, 1977.
- [Tut70] W.T. Tutte. Toward a theory of crossing numbers. *J. Combinatorial Theory*, 8:45–53, 1970.
- [Tut77] W. T. Tutte. Bridges and Hamiltonian circuits in planar graphs. *Aequationes Math.*, 15(1):1–33, 1977.
- [WB78] A.T. White and L.W. Beineke. Selected topics in graph theory. In *Topological Graph Theory*, pages 15 – 50. Academic Press, 1978.
- [Woo93] D.R. Woodall. Cyclic-order graphs and Zarankiewicz's crossing-number conjecture. *Journal of Graph Theory*, 15:657–671, 1993.
- [Yan86] M. Yannakakis. Four pages are necessary and sufficient for planar graphs. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computers*, pages 104–108, 1986.
- [Zar54] K. Zarankiewicz. On a problem of P. Turan concerning graphs. *Fund. Math.*, 41:137–145, 1954.

# Index

- $\lambda(ij, k\ell)$ , 46
- $n$ -planar crossing number, 228
- 3cnf-formula, 21
  
- adjacency, 5
- algorithmic complexity, 19
  - CLIQUE NUMBER, 22
- algorithmic complexity
  - 3SAT, 21
  - polynomial time reducible, 20
  - SAT, 21
  - the class **NP**, 20
  - the class **P**, 20
- arc, 14
- aspiration criteria, 147
- associativity, 17, 18
  
- Beneš graph, 69
- bipartite, 9
- biplanar crossing number, 39
- bisection width, 55
- book crossing number, 39
- book drawings, 31
- book thickness of a graph, 44
- boolean expression
  - 3cnf-formula, 21
  - cnf-formula, 21
  - conjunctive normal form, 21
  - satisfiable, 21
- boolean expression
  - clause, 21
- brick factory problem, 2
- brute force algorithms, 72
- butterfly graph, 69
  
- C language, 239
- C++ language, 239
- C-component, 231
- C-equivalent overlapping, 231
- cartesian product, 8
- certificate, 20
- chains, 46
  
- chromosomes, 138
- circular drawings, 33
- circular encoding, 141
- clause, 21
- clique, 7
- CLIQUE NUMBER, 22
- closure, 17
- cnf-formula, 21
- coarseness of a graph, 45
- commutativity, 18
- complement, 6
- complements of cycles, 71
- complete
  - graph, 9
  - multipartite graph, 9
- complexity, 19
- component, 8
- component-wise, 19
- conjunctive normal form, 21
- connected graph, 8
- contracted independent crossing subgraph, 105
- contraction, 7
- convergence
  - Genetic**, 163
  - GreedySide**, 161
  - tabu**, 166
- convex drawing of a graph, 80
- coset, 17
- cross-coboundary, 47
- crosscap, 15
- crossed toroidal grid graph, 65
- crossing, 10
- crossing number
  - independent-odd, 37
- crossing chain of drawing, 47
- crossing chains, 46
- crossing number
  - book, 39
  - odd, 37
  - pairwise, 36
  - Tutte, 48



- crossover, 138  
 cube connected cycles, 68  
 curve system, 58  
 cycle, 8  
  
 decision theory, 20  
 decomposition tree, 85  
 degree, 6  
   maximum, 6  
   minimum, 6  
 density, 162  
 Dijkstra’s shortest path algorithm, 126  
 dimension, 19  
 directed graph, 14  
 disconnected graph, 8  
 drawing  
   good, 30  
   graph, 26  
   nice, 30  
   normal form, 27  
   single-cross normal form, 30  
 drawings  
   book, 31  
 dynamic algorithms, 139  
  
 edge  
   congestion, 52  
   contraction, 7  
   layout, 129  
   neighbourhood, 6  
   twisting, 78  
 edge set partitioning, 56  
 edges, 5  
 elite solutions, 146, 166  
 embeddings, 34  
 end-vertex, 6  
 existence of inverses, 18  
 existence of inverses, 17  
 expanded independent crossing subgraph, 105  
  
 FFT network, 69  
 fixed linear crossing number problem, 89  
 force of a graph, 59, 60  
 frequency based memory, 149  
 from attributes, 146  
  
 Galois field, 18  
 Garey, Johnson, 72  
 general cross-coboundary, 47  
 genes, 138  
 genus of a graph, 43  
  
 GMP, 263  
 graph, 5  
   directed, 14  
   drawing, 26  
   planar, 10  
   subdivision, 10  
   underlying, 14  
   undirected, 5  
 graph thickness, 39  
 graph minors for lower bounds, 54  
 graph-to-graph embedding, 50, 53  
 greedy algorithm, 132  
 grid graph, 9  
 group, 17  
   Abelian, 17  
  
 Hamiltonian cycle, 254  
 handles sphere, 16  
 Harris algorithm, 73  
 Hopcroft-Tarjan algorithm, 101  
 hypercube, 67  
  
 identity element, 17, 18  
 imperfect copies, 137  
 independent crossing subgraphs, 104  
 independent-odd crossing number, 37  
 induced subgraph, 6  
 initial cross-coboundary, 47  
 intersection graph, 130  
 intersection-edges, 132  
 intersection-vertices, 132  
 intractable problem, 20  
 isolated vertex, 6  
 isomorphic, 6  
 isomorphism, 6  
  
 Johnson, Garey, 72  
  
 Kuratowski’s theorem, 12  
 Kuratowski, C., 229  
  
 layered drawings, 34  
 leaf, 14  
 LEDA, 239  
 left coset, 17  
 left-going edge, 157  
 lower bound algorithm, 117  
  
 make copies, 137  
 mating, 137  
 maximally planar, 11

- maximized average edge congestion, 122  
 maximum degree, 6  
 maximum induced planar subgraph problem, 41  
 mesh of trees, 70  
 minimized average edge congestion, 122  
 minimum degree, 6  
 minor, 7  
 move to solution, 144  
 multipartite, 9  
 mutation, 138
- natural selection, 137  
 nauty isomorphism testing library, 228  
 neighbourhood, 6  
 neighbourhood of a solution, 144  
 neighbourhood search, 144  
 neural network layout algorithm, 89  
 Nicholson’s heuristic, 88  
 non-orientable genus of a graph, 43  
 non-orientable surfaces, 15
- odd crossing number, 37  
 odd crossing number algorithm, 77  
 offspring, 137  
 open neighbourhood, 6  
 order, 5  
 order of magnitude, 19  
 orientable genus of a graph, 43  
 orientable surfaces, 16  
 overlap graph, 233  
 overlapping, 231
- Pósa’s algorithm, 254  
 page number of a graph, 44  
 pageness of a graph, 44  
 pairwise crossing number, 36  
 partial verification, 104  
 path, 8  
 Petersen graph, 70  
 pivot, 254  
 planar, 10, 229  
 planarity testing, 13  
 planarizing drawings, 156  
 plane drawing, 10  
 polynomial time, 19  
 polynomial time reducible, 20  
 population, 138  
 pre-optimization method, 153  
 principal cycles, 58
- Python, 239
- random graph, 162, 163  
 random mapping, 92  
 rays, 82  
 reconsidering edge congestion, 121  
 rectilinear drawings, 34  
 reduced mesh of trees, 70  
 region, 10  
 regular, 9  
 right coset, 17  
 right-going edge, 157  
 rotational transformation, 254  
 rotational embedding schemes, 74
- satisfiable, 21  
 scalar, xxix, 18  
 scout for solutions, 139  
 shortest paths, 123  
 simple curve, 26  
 single-edge graph-to-graph embedding, 54  
 size, 5  
 skew overlapping, 231  
 skewness of a graph, 42  
 soupcan construction of  $\mathcal{K}_n$ , 64  
 source, 14  
 spanning subgraph, 6  
 spine, 32, 129  
 spine drawings, 156  
 splitting number, 43  
 standard counting method, 49  
 star, 9  
 STL, 245  
 subdivision, 10  
 subgraph, 6  
     induced, 6  
     spanning, 6  
 subgroup, 17  
 surfaces  
     non-orientable, 15  
 survival of the fittest, 137  
 survive, 137  
 SWIG, 239
- tabu, 146  
 tabu attributes, 146  
 tabu tenure, 146  
 target, 14  
 tenure, 146  
 thickness, 13

thickness of a graph, 44  
time complexity, 19  
to attributes, 146  
toroidal grid graph, 9  
tournament selection, 138  
tractable problem, 20  
transform drawing to two–page layout, 110  
tree, 14  
    leaf, 14  
tree of meshes, 87  
triangulated, 11  
truncated tree of meshes, 88  
Tutte crossing number, 48  
Tutte, W., 45  
twisted toroidal grid graph, 65  
twisting, 78

underlying graph, 14  
undirected graph, 5

vector, xxix, 18  
vector space, xxix, 18  
vertex  
    congestion, 52  
    degree, 6  
vertex arrangement, 129  
vertex separation, 82  
vertex–transitive, 6  
vertices, 5

walk, 8  
wrap around, 138  
wrapped butterfly graph, 69

zoom in on solution, 139

