An Algorithmic Approach to the 2D Oriented Strip Packing Problem

Nthabiseng Ntene

Dissertation presented for the degree
Doctor of Philosophy
in the inter-departmental programme of Operational Analysis
at the University of Stellenbosch, South Africa

Promoter: Prof JH van Vuuren
Department of Logistics, University of Stellenbosch

December 2007

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation 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: 24/SEPTEMBER/2007

Abstract

Packing problems in industry may be categorised into the two classes of *bin packing* and *strip packing* problems. The former involves packing items into the minimum number of fixed sized bins, while in the latter the items are packed into a single open-ended bin (referred to as a *strip*) such that the total packing height is minimised. The items in both problem categories may not overlap. The entire set of items may be known in advance in which case the problem is referred to as an *offline* problem. On the other hand, in *online* packing problems, only one item is available at a time and the next item only becomes available once the current item has been packed. Problems where some information about the items to be packed (such as a sorting) is available in advance are referred to as *almost online* packing problems.

Offline strip packing problems may be solved using exact algorithms, level heuristics or plane heuristics while online packing problems may be solved using level heuristics, shelf heuristics or plane heuristics. In level heuristics the strip is divided into horizontal levels whose heights are equal to the heights of the tallest items packed on the levels, whereas in shelf algorithms the strip is also partitioned into horizontal levels, but with additional space above the tallest rectangles on the levels to cater for future variation of item heights. On the other hand, in plane algorithms, the strip is not partitioned—items may be packed anywhere within the strip.

Both online and offline two-dimensional rectangle strip packing problems are considered in this dissertation, and the rectangles may not be rotated. An algorithmic approach is employed whereby several algorithms (heuristic and exact) are implemented. A new offline level algorithm is introduced which seeks to fully utilise available space within a level. For online packing problems, a new approach is proposed when creating additional space via shelf algorithms. A new online plane algorithm is also presented. The study aims to find (among the new and a host of known algorithms) the best algorithm to use for different instances of two-dimensional strip packing problems. In reality, such problems often involve a large number of items—therefore the need arises for a computerised decision support system. Such a system, implementing all the (known and new) algorithms described and tested is also presented in this dissertation.

Opsomming

Inpakkingsprobleme in die nywerheid kan gewoonlik in een van twee groepe geklassifiseer word, naamlik *houer inpakkingsprobleme* en *strook inpakkingsprobleme*. Eersgenoemde bestaan daaruit dat 'n lys voorwerpe in die kleinste moontlike aantal houers van vaste grootte ingepak word, terwyl in die laasgenoemde klas voorwerpe in 'n enkele houer of strook van vaste wydte, maar onbeperkte hoogte ingepak word met die doel om die totale inpakkingshoogte te minimeer. In beide klasse probleme mag die voorwerpe nie oorvleuel nie. Die hele lys voorwerpe mag vooraf bekend wees, in welke geval die problem as 'n *aflyn* probleem bekend staan. In *aanlyn* probleme, daarenteen, word die eienskappe van die volgende voorwerp in die lys eers bekend sodra die vorige voorwerp ingepak is. Probleme waar gedeeltelike inligting omtrent die lys voorwerpe wat ingepak moet word (soos byvoorbeeld 'n sortering van een of ander aard) vooraf bekend is, word *bykans aanlyn* probleme genoem.

Aflyn inpakkingsprobleme mag deur middel van eksakte algoritmes, horisontale vlak heuristieke of platvlak heuristieke opgelos word, terwyl aanlyn inpakkingsprobleme tipies deur middel van horisontale vlak heuristieke, rak heuristieke of platvlak heuristieke opgelos mag word. In horisontale vlak heuristieke word horisontale vlakke in die strook gevorm, word die voorwerpe op hierdie vlakke gepak en is die vlakhoogtes gelyk aan die hoogste voorwerpe wat op die betrokke vlakke gepak is, terwyl by rak heuristieke die strook ook in horisontale vlakke verdeel word, maar met die verskil dat die vlakhoogtes hoër is as die hoogste voorwerpe wat op die betrokke vlakke gepak is, om voorsiening te maak vir die moontlikheid dat selfs hoër voorwerpe later op die vlakke gepak mag word. By platvlak heuristieke word die strook glad nie verdeel nie, en mag voorwerpe enige plek in die strook gepak word.

Beide aanlyn en aflyn twee-dimensionele strook inpakkingsprobleme word in hierdie proefskrif beskou, waar die voorwerpe reghoeke is, en nie geroteer mag word nie. 'n Algoritmiese benadering word gevolg waar verskeie algoritmes (heuristies en eksak) geïmplementeer word. 'n Nuwe aflyn horisontale vlak heuristiek word ontwikkel wat poog om die beskikbare vertikale spasie binne elke vlak volledig te benut. Vir aanlyn probleme word 'n nuwe benadering tot die ooplaat van vertikale spasie in rak heuristieke ook voorgestel. 'n Nuwe platvlak heuristiek word ook daargestel. Die studie het ten doel om te bepaal watter van die (nuwe en bekende) heuristieke die beste gepas is vir verskillende twee-dimensionele strook inpakkingsprobleem gevalle. Werklike strook inpakkingsprobleem gevalle behels gewoonlik die inpakking van 'n groot hoeveelheid voorwerpe—daarom ontstaan die behoefte om 'n gerekenariseerde besluitnemingsteunstelsel. So 'n stelsel, waarin al die algoritmes wat in hierdie proefskrif beskryf en getoets is, word ook daargestel.

Acknowledgements

This has truly been a long, challenging journey and the author would like to convey her heartfelt appreciation to everyone who has contributed in some way (big or small) in making this journey a bearable and an enjoyable experience. I would like to thank:

- my promoter, Prof JH van Vuuren, for his invaluable input, guidance, expertise and encouragement, but most of all for his patience. You made me feel welcome from day one and for that I will be eternally grateful.
- the Department of Mathematical Sciences at the University of Stellenbosch for the use of their office space and computing facilites.
- Mr Robert Kotze and the staff of the international office at the University of Stellenbosch for their efficiency and assistance on various matters.
- the Third World Organisation for Women in Science (TWOWS) for funding this study. Without their financial assistance this study would not have been possible. I would also like to thank the Harry Crossley Foundation administered by the bursary office at the University of Stellenbosch for their additional funding.
- Mr John Part for his helpful hints and suggestions on programming techniques.
- my colleagues Dr Werner Gründlingh, Mr Frank Ortmann and Mrs Margarete Bester to name but a few, for their input on various issues including typesetting assistance.
- a special thank you to my parents Kabelo and 'Mamosili Ntene for their constant love and unwavering support in this journey. I would also like to thank my sister Mosili for always being there for me and the frequent phone calls. I could not have asked for a better family and I love you. To my extended family in particular, *rangoane Thulo* for always taking an interest in my school work from a very early age.
- all my friends for their support, phone calls and e-mails.
- finally, saving the best for last, I would like to thank almighty God without whom none of this would be possible. Thank you Father for blessing me, loving me, giving me strength, wisdom and intelligence. Thank you for bringing the praying buddies into my life, because they have helped to enhance my spiritual intelligence.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The field of cutting and packing (C&P) problems has been researched extensively as a sub-discipline of operations research for many years, from as early as 1939 [67]. A cutting problem may be described as the problem of cutting a large object into a maximum number of smaller items of specified shape by minimising wastage or offcuts, while a packing problem is the problem of packing a maximum number of items into a specified volume by minimising wasted or empty space. There is a strong relationship between the typologies of C&P problems, resulting from the duality of material and space (*i.e.* the duality of a solid material body and the space occupied by it [41]). The continued interest in this field has been because of the need to develop automated packing layouts and cutting patterns for industry so as to yield better solutions than are manually possible.

1.1 Two C&P typologies from the literature

A typology, as defined by Wäscher *et al.* [114], is *"a systematic organisation of objects into homogeneous categories on the basis of a given set of characterising criteria."* Dyckhoff [41] developed the first typology of C&P problems by considering a number of problems with the same logical structure, but which appear under different names, depending on the discipline or context. The basic infrastructure of C&P problems comprises large objects and small items—the large objects are either cut into a number of small items or the small items are combined into patterns or geometric arrangements that are assigned to the large objects.

Dyckhoff's typology comprises four main characteristics, shown in Table 1.1. The first characteristic, *dimensionality*, indicates the geometric dimensions of the large objects and small items. The second characteristic, *kind of assignment*, distinguishes between whether the problem involves a selection of the small items to form patterns assigned to all large objects or whether all items are assigned to a selection of the large objects. The *assortment of large objects* is the third characteristic, making a distinction as to whether there is only one large object, many large identical objects or different large objects. Finally, the last characteristic considers the *assortment of small items*—whether there are few or many different shapes, many items of relatively few different shapes or whether all items are congruent.

Dyckhoff's C&P typology		Wäscher's improved C&P typology	
Dimensionality			
1	One-dimensional	**1**	One-dimensional
2	Two-dimensional	**2**	Two-dimensional
3	Three-dimensional	**3**	Three-dimensional
N	N-dimensional	**N**	N-dimensional
Kind of assignment			
B	All objects and a selection of items	**OM**	Output value maximisation
V	A selection of objects and all items	**IM**	Input value minimisation
Assortment of large objects			
O	One object	**O**	One object
I	Identical figures		**Oa** all fixed dimensions
D	Different figures		**Oo** one variable dimension
			Om more variable dimensions
		Sf	Several figures
			Si identical figures
			Sw weakly heterogeneous assortment
			Ss strongly heterogeneous assortment
Assortment of small items			
F	Few items (of different figures)	**IS**	Identical small items
M	Many items of many different figures	**W**	Weakly heterogeneous assortment
R	Many items of relatively few different (incongruent) figures	**S**	Strongly heterogeneous assortment
C	Congruent figures		

Table 1.1: *Dyckhoff's [41] and Wäscher's [114] typologies for C&P problems.*

Example 1.1 *Using Dyckhoff's typology, the classical one-dimensional cutting stock problem may be coded as 1/V/I/R. This is the problem of cutting smaller items from an unlimited number of stock sheets with the objective of using the minimum number of stock sheets. The coding indicates a one (1) dimensional packing where all items are assigned to a selection (V) of identical (I) stock sheets. There are many items of relatively similar shape and size (R).* □

According to Wäscher *et al.* [114], the typology by Dyckhoff [41] has certain drawbacks in terms of incorporating some of the latest developments in C&P problem research. One of the drawbacks which is relevant to this study is that of coding two-dimensional strip packing problems (packing 2D items into an open ended rectangular bin so as to attain the minimum packing height possible). Dyckhoff uses the notation 2/V/D/M, while Wäscher *et al.* [114] suggest the more plausible coding 2/V/O/M. The inconsistency arises as a result of treating the problem as a two-dimensional bin packing problem in [41], where one object of minimal length has to be selected from a number of larger objects available. It is recommended in [114] that a distinction be made between bin packing and strip packing. It is due to this and other drawbacks that an improved typology of C&P problems, which is an extension of Dyckhoff's typology, was proposed by Wäscher *et al.* [114]. It was also proposed in [47] that a distinction should be made in one-dimensional cutting between the standard material of different sizes when a large number of small items of relatively few different shapes is to be produced from an unlimited supply of such a standard material. The material may be partitioned into a few groups of identical sizes or the sizes may be entirely different—these two cases require different solution approaches.

The improvements proposed by Wäscher *et al.* [114] are shown in Table 1.1. The second characteristic, *kind of assignment*, was rather categorised into input value minimisation (all small items are assigned to a selection of large objects with minimal value) or output value maximisation (all large objects are used with a selection of small items of maximal value). The third characteristic, *assortment of large objects*, was categorised into one large object and several large objects and the fourth characteristic, *assortment of small objects*, was categorised into identical small items, weakly heterogeneous assortment (few items are of different shapes) or a strongly heterogeneous assortment (only a few elements are of identical shape and size). For an in-depth analysis of the improved typology, the reader is referred to [114].

Example 1.2 *The knapsack problem may be coded as */OM/*/S when using the improved typology. The * in the first and third fields implies that an arbitrary choice among any of the options in the field applies depending on the application. In the knapsack problem, items of different values and volumes (S) are packed into a space with the objective of maximising the total value associated with the packing (OM).* ◘

1.2 Dissertation Scope

The focus in this dissertation is on packing problems. Due to their diversity in application areas, these problems may be classified according to either spatial or non-spatial problems.

Example 1.3 *An example of spatial packing problem is the bin packing problem in which a collection of 2D shapes may be packed either into a single 2D bin (packing the maximum number of 2D shapes into a single closed 2D bin) or into multiple 2D bins (packing 2D shapes into the minimum number of 2D bins), depending on the application.* ◘

Example 1.4 *On the other hand, an example of a non-spatial packing problem is the capital budgeting problem in which, due to fixed or limited capital budgets, capital has to be rationed amongst projects with positive net present values. In this example, the capital represents the large object, while the projects represent the small items.* ◘

Non-spatial dimension applications of packing problems are beyond the scope of this study. In fact, only two-dimensional spatial packing problems are considered in this dissertation, as shown schematically in Figure 1.1.

Spatial 2D packing problems may be sub-classified further into the two main classes of *2D strip packing* problems and *2D bin packing* problems. Each of these types of 2D packing problems may be differentiated further into three categories: *offline problems, almost on-line problems* and *on-line problems*, as shown in Figure 1.1. The latter refers to the problem whereby only the specification (with respect to shape and size) of one 2D item is known at a time—the specification of the next item to be packed only becomes known once the previous item has been packed [6, 32, 33, 72, 91]. In contrast, specifications of the entire set of 2D items to be packed are known in advance in *offline* problems [29]. Finally, Bartholdi, *et al.* [8] define so-called *almost on-line* problems as those packing problems where some—but not much—foreknowledge about the items to be packed is exploited. Imreh [62] noted that one may often obtain better packing results if the

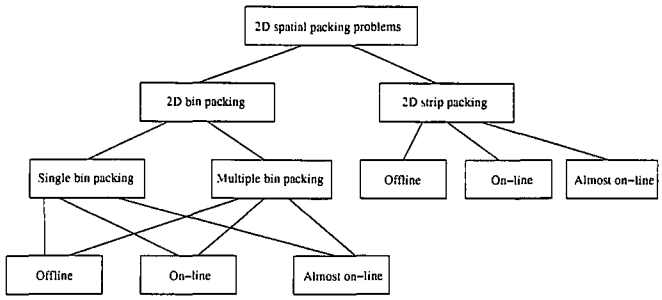

Figure 1.1: *Different classes of 2D spatial packing problems.*

problem is not completely *on-line*, i.e. when some information about the list of items to be packed is known before packing, such as, that the items are to be packed in the order of non-increasing height.

Packing problems are generally believed to be NP-hard[1]. Many instances of packing problems have, in fact, been proven to be NP-hard [15, 17, 21, 29, 70]. This means that it is highly unlikely that a time-efficient algorithm will be found which is capable of constructing optimal solutions to packing problems. This observation directs one's line of approach toward solving packing problems approximately [20]. For example, one may try to develop an efficient algorithm that always guarantees a 'nearly optimal' solution, rather than seeking an optimal solution at high computational cost. In fact, the vast majority of C&P problem research has concentrated on developing approximate or heuristic packing procedures, as will become evident later in this dissertation.

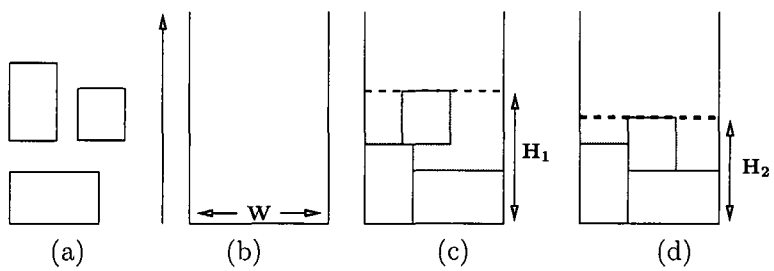

Figure 1.2: *A demonstration of 2D strip packing of rectangles. The items are packed into the strip of fixed width* $\mathbf{W}$ *such that they do not overlap. The heights* $\mathbf{H_1}$ *and* $\mathbf{H_2}$ *represent suboptimal and optimal packing heights respectively.*

1.3 Problem statement

The 2D strip packing problem was first proposed by Baker *et al.* [4] in 1980. This is the problem of packing small items (in this dissertation the small items are assumed to be rectangles, as shown in Figure 1.2(a)), without overlapping into an open ended rectangular bin (referred to as a strip) of fixed width ($\mathbf{W}$) and infinite height, as shown in Figure 1.2(b). The objective is to minimise the total height of the packing ($\mathbf{H}$). Sub-optimal (H_1) and optimal (H_2) packing heights are shown in Figures 1.2(c) and (d)

[1]The basic concepts of algorithmic complexity theory are discussed in Appendix A.

respectively. In an *oriented orthogonal* packing one is not allowed to rotate rectangles during the packing procedure; the side of each rectangle must be parallel to the sides of the strip, as shown in Figure 1.3(a). In a *non-orthogonal* packing, however, one is allowed to place rectangles at various angles in the strip, as shown in Figure 1.3(b). A *guillotine* packing is a packing for which it is possible to cut along the edges of the rectangles (in some order) by means of entirely edge-to-edge cuts, as shown in Figure 1.3(c), while in a *non-guillotine* packing this is not possible, as is the case in Figure 1.3(d).

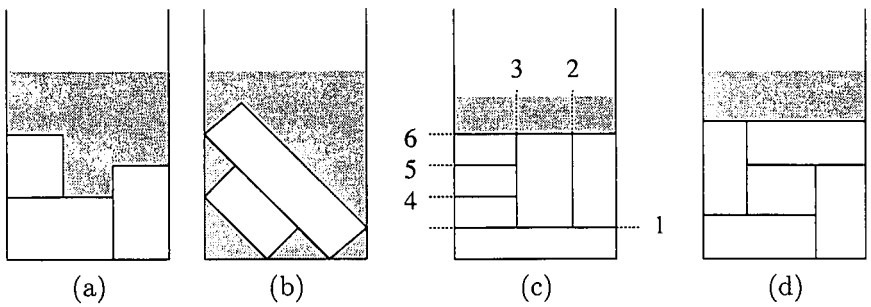

Figure 1.3: *An illustration of different types of packings: (a) an orthogonal packing, (b) a non-orthogonal packing, (c) a guillotine packing and (d) a non-guillotine packing. The rectangles in layout (c) may be isolated by performing edge-to-edge cuts in the order shown by the numerals in the figure, while in (d) none of the rectangles may be isolated by performing edge-to-edge cuts.*

In this study, all packings are assumed to be oriented and orthogonal, with no restriction on guillotinability. The aim is to determine fast and effective heuristic procedures for solving 2D strip packing problems, since good solutions may lead to considerable cost savings.

The heuristics considered in this study are grouped as *level*, *shelf* and *plane* algorithms. In level algorithms, the strip is partitioned into horizontal levels and the height of each level is determined by the height of the tallest rectangle packed in the previous level, as shown in Figure 1.4(a). Shelf algorithms use the same principle as the level algorithms of partitioning the strip, however, each shelf height of a newly created shelf is predetermined by various methods, as shown in Figure 1.4(b). Finally, in plane algorithms, the strip is not partitioned, as shown in Figure 1.4(c). The rectangles are packed anywhere inside the plane of the strip. These heuristics will be discussed fully in Chapters 2–3.

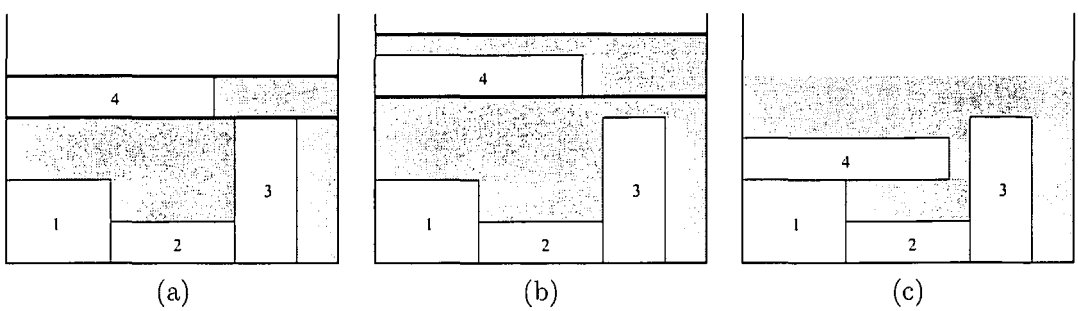

Figure 1.4: *An illustration of different groups of algorithms; (a) level, (b) shelf and (c) plane algorithms.*

1.4 Dissertation objectives

The primary focus in this dissertation is on reviewing as well as developing new, fast and effective heuristics for the 2D strip packing problem. Some exact algorithms will also be analysed. The five main objectives pursued in this dissertation are:

Objective 1: To develop a general unifying classification scheme for packing problems.

Objective 2: To review known methods from the literature capable of solving 2D strip packing problems (approximately or exactly) and to endeavour improving some of these algorithms.

Objective 3: To establish new

a) level algorithms for the offline strip packing problem,

b) shelf and plane algorithms for the online strip packing problem.

Objective 4: To implement all the algorithms in Objectives 2 and 3 (known and new) on a computer and to compare their performance

a) in terms of solution quality and time efficiency, using benchmark data sets,

b) and to investigate attributes of data sets that may influence algorithm performance.

Objective 5: To design and implement an active (computerised) decision support system capable of assisting managers in industry who have to solve 2D strip packing problems during the course of their work.

1.5 A new subtypology for packing problems

The typologies reviewed in §1.1 cover the broad-spectrum of C&P problems in general terms. However, in this section a novel classification or subtypology is especially developed for packing problems. This classification comprises six fields, denoted in the array format

$$\boxed{\alpha}\boxed{\beta}\boxed{\chi}\boxed{\gamma}\boxed{\lambda}\boxed{\tau}, \tag{1.1}$$

which is used throughout the remainder of this dissertation. The most important characteristic in packing problems is dimensionality. It describes the geometry of the items to be packed. Therefore the entry in the first field of (1.1), $\alpha \in \{1\text{D}, 2\text{D}, 3\text{D}, \text{HoD}\}$, denotes the dimension in which the packing takes place. Here $\alpha = n\text{D}$ indicates that the problem is an n dimensional packing problem, for $n = 1, 2, 3$. Furthermore, $\alpha = \text{HoD}$ denotes a higher dimensional packing problem.

In one-dimensional packing problems, the widths of the items to be packed are equal to the width of the bin but the items have varying heights; hence only one dimension (namely height) is of importance. In two-dimensional packing problems, the items to be packed have varying widths and heights, while in three-dimensional packing problems, the items have varying widths, heights and depths. Higher dimensional packing problems

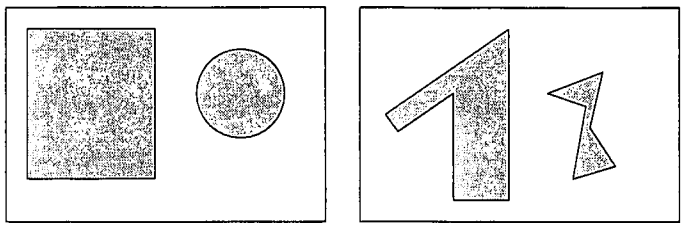

Figure 1.5: *Examples of (a) regular and (b) irregular shapes.*

involve packings in dimensions higher than three, where dimensions are not necessarily spatial—the fourth dimension may represent time (for example, three dimensional items may be required to be packed for a fixed period of time).

The second field characterises the shapes of the objects to be packed and this is closely related to dimensionality. A packing of either regularly or irregularly shaped items may be required; hence $\beta \in \{\mathrm{I}, \mathrm{R}\}$. Here $\beta = \mathrm{R}$ indicates that regular shaped items are packed, while $\beta = \mathrm{I}$ denotes the fact that irregular shaped items are packed. The notion of regular or irregular shapes is as defined by Hopper [56]. A regular shape is described as a shape defined by a few parameters, such as rectangles or circles, as shown in Figure 1.5(a). Irregular shapes, on the other hand, may include concavities and/or asymmetric shapes such as those patterns encountered in the clothing and textile industries, as shown in Figure 1.5(b).

In packing problems, smaller items are packed into a well defined region commonly called a bin or strip (depending on the application). The third field of the proposed classification notation in (1.1) defines these regions as $\chi \in \{\mathrm{MFB}, \mathrm{MVB}, \mathrm{SB}, \mathrm{SP}\}$. Here $\chi = \mathrm{MFB}$ denotes a multiple fixed sized bin packing, $\chi = \mathrm{MVB}$ denotes a multiple variable sized bin packing, $\chi = \mathrm{SB}$ corresponds to a single bin packing and lastly $\chi = \mathrm{SP}$ represents a strip packing problem.

In most applications, the items to be packed are drawn from a finite list. The fourth field in (1.1) differentiates the level of available information (specifications of the list of items to be packed) as $\gamma \in \{\mathrm{Off}, \mathrm{Aon}, \mathrm{On}\}$. Here $\gamma = \mathrm{Off}$ represents an offline packing problem, $\gamma = \mathrm{Aon}$ indicates an almost on-line packing problem, and $\gamma = \mathrm{On}$ denotes an on-line packing problem.

The critical issue in packing problems is to make efficient use of time and/or space [109]. The objective in a packing problem is either to minimise or maximise a particular quantity and is addressed in the fifth field of the notation in (1.1). In particular, $\lambda \in \{\mathrm{MaI}, \mathrm{MiA}, \mathrm{MiB}, \mathrm{MiC}, \mathrm{MiS}\}$. Here $\lambda = \mathrm{MaI}$ denotes maximising the number of items to be packed, $\lambda = \mathrm{MiA}$ denotes minimising the area of packing, $\lambda = \mathrm{MiB}$ denotes minimising the number of bins, $\lambda = \mathrm{MiC}$ denotes minimising the cost of the packing, while $\lambda = \mathrm{MiS}$ denotes minimising the strip height.

Similar to other problems, packings are typically also subjected to certain constraints. The more basic and common constraints encountered in packing problems are dealt with in binary fashion in the sixth field of (1.1), in which $\tau = \{\tau_o, \tau_p, \tau_m, \tau_g\}$ is a binary 4-vector. The parameter $\tau_o \in \{0, 1\}$ indicates whether the orientation of the items to be packed is fixed or not. Some applications allow for items within a bin or strip to be rotated while others do not. Here, $\tau_o = 0$ represents a fixed orientation, while $\tau_o = 1$

means that rotation is allowed.

The parameter $\tau_p \in \{0, 1\}$ indicates that constraints on the placement of items are present. An example of such a constraint application may be that boxes carrying fragile material may not be placed at the bottom of a packing. In particular, $\tau_p = 0$ means that no restriction on the placement of items is enforced, while $\tau_p = 1$ indicates that restrictions on the placement of items are present.

The parameter $\tau_m \in \{0, 1\}$ indicates whether the shape of the items to be packed may be modified. Shape modification may arise in the scheduling of tasks on a computer, where the length and width of an item may represent time and resource required for completing a particular task. Shape modification usually takes place by either lengthening the item (thereby using less resource and more time) or widening the item (hence using more resource, but taking less time to complete a task). Here, $\tau_m = 0$ indicates that the shapes of items may not be modified, while $\tau_m = 1$ means that modification of shapes is allowed.

The parameter $\tau_g \in \{0, 1\}$ represents the constraint of guillotine cuts. Some applications may disallow a certain packing pattern unless it may be disentangled by performing edge to edge cuts parallel to the edge of the bin or strip (*i.e.* unless it is a guillotine packing). Here, $\tau_g = 0$ means that there is no restriction on guillotinability, while $\tau_g = 1$ means that a guillotine packing is required.

Finally, the convention is adopted that an asterisk in any field of (1.1) denotes the fact that the contents of that field is not specified, resulting in a class of packing problems rather than in a single packing problem. Due to the considerable diversity of real-world packing problems, this new classification scheme does not in any way cover all possible properties of packing problems. The characteristics covered are considered to be basic, but representative for packing problems. The classification has, however, been constructed to be flexible and may easily be adapted to suit any problem by adding additional properties to the fields in (1.1). Some fields, such as the constraints field, may be more detailed and the list of possibilities may increase when considering certain special cases or variants of packing problems. The classification notation introduced above is illustrated by means of an example and a summary is provided in tabular form in Appendix F.

Example 1.5 *The focus of this study will be on 2D strip packing of oriented rectangles coded as*

2D	*R*	*SP*	*	*MiS*	*0,0,*,**

indicating that there is no restriction on the level of information or on the modification of shapes. There is also no restriction on guillotinability or the placement of rectangles, and the objective is to minimise the total packing height. □

1.6 Benchmark instances

The use of benchmark problems is a standard approach towards the appraisal of new algorithmic procedures, facilitating comparisons between the space and time efficiencies, and the solution qualities of such procedures with those of existing ones. Previously researchers often used to test their algorithms on data sets newly created by themselves, but then failed to make these data available when they published the performance appraisals

of their algorithms. Such failures defeat the purpose of benchmark testing by causing researchers to keep on generating more and more instances of test data instead of reverting to existing data[2]. This issue has been addressed, since some researchers [18, 23, 56, 57], have started publishing their work on packing problems along with the test instances they used and/or generated. A number of on–line libraries (see, for example, [37, 60, 100, 107]) were created to publish benchmark data on the internet for the purposes of testing algorithms designed to solve a large variety of well–documented problems in the operations research literature. The author follows suit, by making available on the internet all the test data used in the evaluations of the algorithms in this dissertation [113], although new test data have not been created for testing purposes. The methods used by different researchers to generate these benchmarks are discussed briefly in §1.6.1–1.6.5.

1.6.1 The Mumford-Valenzuela benchmark data

The data sets described in this section were created to test algorithmic procedures for packing problems of the form

2D	R	SP	Off	MiS	1,0,0,1

,

where 90 degree rotations were allowed and guillotine cut restrictions were present. Mumford-Valenzuela *et al.* [94] generated two classes of data sets. The first class is called the class of *Nice* data sets and each data set in this class consists of rectangles of similar sizes and shapes, whilst the second class is referred to as the class of *Path* data sets (short for pathological) and each data set in this class consists of rectangles with significantly varying shapes and sizes.

The procedure used to generate these data sets allowed for the aspect and area ratios of the rectangles to be specified. The *Nice* and *Path* data sets have aspect ratios within the ranges $1/4 \leq h/w \leq 4$ and $1/100 \leq h/w \leq 100$ respectively, where h and w respectively denote the height and width of the newly generated rectangles. The maximum ratios of the areas of any two rectangles were set to 7 for the *Nice* data set and to 100 for the *Path* data set. Guillotineable data sets of size n with known optimal solutions were created by performing $n-1$ guillotine cuts on a 100×100 square. At each cutting stage, all constraints of aspect and area ratios were satisfied. For each class of data sets the sizes (*i.e.* the number of rectangles) $n = 25, 50, 100, 200, 500$ were selected and these instances are denoted Nice.n and Path.n. Fifty test instances for each problem size within each class were created, except for the case $n = 500$, for which only ten test instances were created, resulting in a total of four hundred and twenty data sets. The data sets generated have the characteristic that all rectangle dimensions are real numbers. The pseudocode of the algorithm used during the generation process is available in [116, p. 386]. These data sets may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/Mumford-Valenzuela/.

[2]This practice of duplication brings with it the added disadvantage of even having to re-implement existing algorithms in order merely to compare the qualities of solutions obtained by them to those of new algorithms, instead of using published results for the existing algorithms and only implementing the new algorithms in such comparisons.

1.6.2 The Burke benchmark data

The data sets generated by Burke *et al.* [18], were applied to problems of the form

2D	R	SP	Off	MiS	1,0,0,0

where rotations of 90 degrees were allowed. They generated thirteen test instances of which only twelve are used in this dissertation (denoted $B_1, \ldots, B_{12}$)[3]. These instances were randomly generated by cutting a large rectangle repeatedly, either vertically or horizontally such that two new rectangles were generated after each cut, as depicted in Figure 1.6(a). A list of the rectangles generated was maintained after each cut, from which a rectangle was selected and a further cut performed on it. While ensuring that the dimensions of the rectangles thus generated were not smaller than some specified minimum dimension, the process was carried out until the desired number of smaller rectangles was obtained. These data sets may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/Burke/.

1.6.3 The Hopper and Turton benchmark data

Hopper and Turton [57] developed benchmark data for problems of the form

2D	R	SP	Off	MiS	1,0,0,0

They developed data sets denoted C_i^j, divided into seven categories ($i = 1, \ldots, 7$), each category comprising three instances ($j = 1, 2, 3$). The number of rectangles in each category ranges from 17 to 197 rectangles and the categories are arranged according to an increasing number of rectangles (for example, C_1^4 involves more rectangles than does C_1^2). These data sets were generated randomly, maintaining a maximum aspect ratio of 7 and the optimal solutions to all test instances are known, because a large rectangle of known dimensions was cut into the required number of smaller rectangles to generate the data sets. Each category has a large rectangle associated with it and in all the categories these large rectangles have aspect ratios varying between 1 and 3. The data may be accessed via the on–line libraries [37, 60, 107] and may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/HopperandTurton1/.

Other test instances by Hopper and Turton [58] were meant for problems of the form

2D	R	SP	Off	MiS	0,0,0,*

They generated classes of both guillotineable data sets (denoted T_i^j) and non-guillotineable data sets (denoted H_i^j), each with a maximum aspect ratio of 7. Each class ($i = 1, \ldots, 7$) comprises five instances ($j = 1, \ldots, 5$) which were cut from a 200×200 square in order to eliminate bias due to dimensions of the large rectangle. The guillotineable test problems were generated by repeatedly selecting a random point in a rectangle and making vertical and horizontal cuts through that point to generate four new rectangles, as depicted in Figure 1.6(b). At each cutting stage the maximum aspect ratio was observed, failing which a new random point was selected. In the case of the non-guillotineable test problems, two random points were selected within a rectangle, which served as the opposite

[3]The thirteenth instance was discarded because it involves 3150 rectangles, and it was decided to use test instances with up to 500 rectangles.

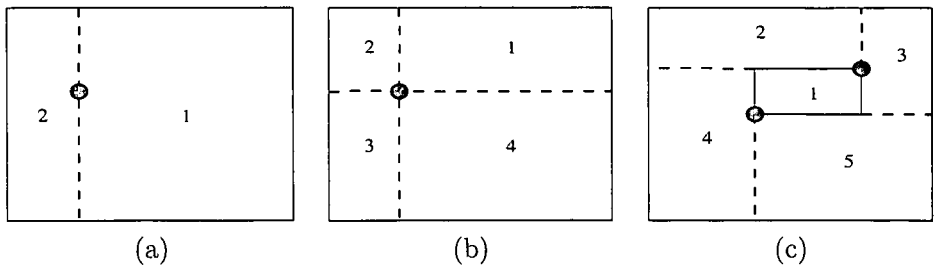

Figure 1.6: *Generation of guillotine and non-guillotine cuts; (a) guillotine cut resulting in two rectangles, (b) two guillotine cuts resulting in four rectangles and (c) four non-guillotine cuts resulting in five rectangles.*

corner points to a smaller rectangle lying wholly within the larger rectangle, as depicted in Figure 1.6(c). The sides of the smaller rectangle were extended up to a point where they intersect the sides of the larger rectangle—thus forming five new rectangles. In both cases, the process was repeated until the required number of rectangles was obtained. These data sets are available on–line [107] and may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/HopperandTurton2/.

1.6.4 The Christofides and Whitlock benchmark data

Some benchmark problems proposed for cutting stock problems were transformed and adapted to strip packing instances by taking the width of the large rectangle from which the smaller rectangles in each test instance were cut as the strip width. The data set generated by Christofides and Whitlock [23] consists of three instances, denoted G_1, G_2, G_3 and were intended for the constrained guillotine cutting problem. The areas of the n smaller rectangles (denoted α_i, $i = 1, \ldots, n$) were generated by sampling from a uniform distribution in the range $[0, 0.25A]$, where A denotes the area of the initial large rectangle. After obtaining the areas of the n rectangles, the height of each rectangle, $h(L_i)$, was obtained by again sampling from a uniform distribution in the range $[0, \alpha_i]$, rounding up to the nearest integer. Finally, the width of each rectangle, $w(L_i)$, was simply computed using the formula $w(L_i) = \lceil \alpha_i / h(L_i) \rceil$. The optimal strip packing solutions for these data sets are unknown, the data are available on–line [100, 107] and may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/ChristofidesandWhitlock/.

1.6.5 The Beasley benchmark data

Beasley [9] generated a number of data sets, of which only four are available online (denoted U_1, U_2, U_3, U_4), for the unconstrained guillotine cutting problem. Integers were sampled from uniform distributions in the ranges $[H/4, 3H/4]$ and $[W/4, 3W/4]$ for respectively the height $h(L_i)$ and width $w(L_i)$ of each rectangle generated.

Beasley [10] also produced twelve non-guillotineable, random problem instances (denoted $V_1, \ldots, V_{12}$), by generating n real numbers r_i $(i = 1, \ldots, m)$ from a uniform distribution in the range $(0, HW/4)$. The height, $h(L_i)$, of a rectangle was generated by sampling an integer from a uniform distribution in the range $[1, H]$ and the width was set to $w(L_i) =$

$\lceil r_i/h(L_i) \rceil$. Optimal solutions to the corresponding problems are unknown for both these classes of data sets, the data may be accessed on–line [100] and may also be found on the CD accompanying this dissertation, within the directory BenchmarkData/Beasley/.

1.7 Preview of dissertation layout

The workings of all algorithms considered in this dissertation are illustrated by means of a single example instance and formalised by means of pseudocode listing. The use of pseudocode is preferred instead of a mere description or specific programming language listing, because of its precision, structure and universality [66].

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}
Width, $w(L_i)$	14	2	9	10	4	4	2	8	3	5
Height, $h(L_i)$	4	4	3	5	10	1	6	2	3	1

Table 1.2: *Rectangle dimensions used as an example throughout this dissertation to illustrate the steps of the various algorithms.*

The general notation used throughout the dissertation is that, for a given list $\mathcal{L}$ of rectangles, $h(L_i)$ and $w(L_i)$ denote respectively the height and width of rectangle L_i, for all $i = 1, \ldots, n$ (here n denotes the number of rectangles to be packed). The dimensions of the rectangles to be packed into a strip of width 16 spatial units in the example used throughout the dissertation may be found in Table 1.2.

The remainder of the dissertation is structured such that offline and online strip packing problems are considered heuristically in Chapters 2 and 3 respectively. Each chapter begins with a description of known heuristics from the literature (in §2.1 and §3.1 for offline and online packing problems respectively). This is followed by proposed improvements to some of these known heuristics (in §2.2 and §3.2 for offline and online packing problems respectively). New heuristics for offline and online packing problems are presented in §2.3 and §3.3 respectively.

Exact algorithms producing guillotineable and non-guillotineable packings are described in Chapter 4. These include the use of branch and bound methods as well as other linear programming relaxation techniques. The performances and efficiencies of the offline and online algorithms (exact and heuristic) are compared in Chapters 5 and 6 respectively. The comparisons are carried out in terms of solution quality (*i.e.* the mean packing height achieved for the benchmark data described in §1.6), the frequency with which the smallest packing height is obtained for benchmark data and execution times.

A computerised decision support system for strip packing problems is introduced and described in Chapter 7. A flow chart of its working, as well as various components of the decision support system, are shown by means of screen shots. In Chapter 8, a summary of the work accomplished in this dissertation is presented (§8.1). Several suggested avenues of packing problems to explore for future work are provided in §8.2.

Some of the main concepts in algorithmic complexity and performance analyses are described in Appendix A. This is followed, in Appendix B, by a description of some of the principles of one-dimensional bin packing used in the Kenyon-Remila algorithm. When

analysing and comparing the performance of algorithms in Chapters 5 and 6, a number of statistical tools are employed—these are explained in Appendix C. The mean packing heights and execution times achieved by the online heuristics for the benchmark instances are summarised in tabular form in Appendix D for three data traversal orders. Finally, all the necessary information about a compact disc accompanying this dissertation is provided in Appendix E. The compact disc contains among other things the source code of the decision support system, an electronic copy of the dissertation along with the LaTeX source files and the benchmark data.

Chapter 2

Heuristics for offline problems

A number of heuristic procedures have been put forward in the literature to solve problems relating to the offline 2D strip packing problem and some of these are described in this chapter. This chapter opens with a discussion on known heuristics from the literature (§2.1), followed by suggested modifications to some of these heuristics (§2.2) and finally a presentation of a new heuristic (§2.3).

2.1 Known heuristics

Heuristics are strategies for solving optimisation problems approximately by constructing "good", but not necessarily optimal solutions at a reasonable computational cost. Because the strip packing problem is NP-hard and its real world applications normally lead to large scale problems, practitioners usually resort to heuristic methods rather than to exact methods. The first class of heuristics, referred to as *level algorithms*, is described in §2.1.1 and this is followed by the description of another class of heuristics known as *plane algorithms* (in §2.1.2).

2.1.1 Level algorithms

As mentioned in the introduction, each rectangle is packed in the order given on any one of a collection of horizontal levels drawn across the strip in a level algorithm. These algorithms are primarily used when dealing with *offline* packing problems and the first level of the strip is the bottom of the strip. The height of each level is determined by the height of the tallest rectangle placed on the previous level (a horizontal line is drawn through the top of the tallest rectangle placed on the previous level).

2.1.1.1 The next fit decreasing height algorithm

The so-called *next fit decreasing height* (NFDH) algorithm [30] was developed in 1980 to solve problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

The list of rectangles to be packed is pre-ordered according to non-increasing height. This allows for the first rectangle placed on a level to determine the height of the next level. A rectangle is placed on the current level, left justified, if it fits there. However, if it does not fit, then a new level is created above the current level (which becomes the new current level) and the rectangle is placed there. The packing progresses from left to right per level and from the bottom of the strip upwards level-wise. Levels lower than the current level are never revisited. Rectangles of equal height retain their original order in the packing list relative to each other. When analysing the performance of the NFDH algorithm, Coffman *et al.* [30] found the asymptotic performance bound[1]

$$\text{NFDH}(\mathcal{L}) \leq 2\,\text{OPT}(\mathcal{L}) + 1 \tag{2.1}$$

in the limit as $n \to \infty$, where $\text{NFDH}(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the NFDH heuristic, and where $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. In (2.1) the multiplicative constant 2 is the smallest possible. The steps of the NFDH algorithm are given in pseudocode as Algorithm 1.

Algorithm 1 The next fit decreasing height (NFDH) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height of a packing obtained in the strip.

level $\leftarrow 0$; $h(\text{level}) \leftarrow 0$; $w(\text{level}) \leftarrow 0$; $i \leftarrow 1$
Renumber the rectangles in order of non-increasing height such that $h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$
Pack rectangle L_i left-justified at the bottom of the strip
$h(\text{level}) \leftarrow h(L_i)$; $w(\text{level}) \leftarrow w(L_i)$
for $i = 2, \ldots, n$ **do**
  if $W - w(\text{level}) \geq w(L_{i+1})$ **then**
    pack rectangle L_{i+1} to the right of rectangle L_i
    $w(\text{level}) \leftarrow w(\text{level}) + w(L_{i+1})$
  else $[W - w(\text{level}) < w(L_{i+1})]$
    create a new level above the previous one and pack rectangle L_{i+1} on the new level
    level $\leftarrow$ level $+ 1$; $w(\text{level}) \leftarrow w(L_{i+1})$; $h(\text{level}) \leftarrow h(\text{level} - 1) + h(L_{i+1})$
  end if
end for
print $H = h(\text{level})$

Example 2.1 *The rectangles in Table 1.2, which are required to be packed into a strip of width 16 units, are first arranged in non-increasing order by height as* $\{L_5, L_7, L_4, L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$ *and then packed, as shown in Figure 2.1(a). Rectangles* $\{L_5, L_7, L_4\}$ *are packed on the first level. This level is closed off and a new level is created where rectangles* $\{L_2, L_1\}$ *are packed. Packing progresses in this manner until all rectangles are packed and a total packing height of 20 units is obtained.* □

[1] Asymptotic performance bounds are often employed to measure the efficiency of algorithms. These bounds are a result of a theoretical analysis of the worst-case performance of the algorithm [75], as discussed in detail in Appendix A.

Since lines 6–12 of Algorithm 1 have constant time complexity, the for–loop spanning lines 5–13 has a time complexity of $O(n)$, where n is the number of rectangles in the list $\mathcal{L}$. However, the complexity of the algorithm is dominated by line 2 which has a worst-case time complexity of $O(n \log n)$, since line 1, lines 3–4 and line 14 also have constant time complexity. This worst-case time complexity may be obtained when using an efficient sorting procedure such as the merge-sort algorithm which uses a divide-and-conquer technique [66]. Therefore the worst-case time complexity of the NFDH algorithm is $O(n \log n)$.

2.1.1.2 The first fit decreasing height algorithm

In the so-called *first fit decreasing height* (FFDH) algorithm [30], the list of rectangles is also pre-ordered according to non-increasing height. The algorithm also dates from 1980 and may again be applied to problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

Rectangles of equal height retain their original order relative to each other in the packing list. A rectangle is placed left justified on the lowest level with sufficient space. The strip is searched level-wise from the bottom upwards for sufficient packing space and if the current rectangle does not fit into any of the existing levels, a new level is created above the current top level (which becomes the new top level) and the rectangle is placed there. The difference between the NFDH and FFDH algorithms is therefore that in the latter, previously packed levels are always searched for sufficient space to pack a rectangle whereas in the former, previously packed levels may not be revisited. When analysing the performance of the FFDH algorithm, Coffman *et al.* [30] found the asymptotic performance bound

$$\text{FFDH}(\mathcal{L}) \leq 1.7\,\text{OPT}(\mathcal{L}) + 1 \tag{2.2}$$

in the limit as $n \to \infty$, where $\text{FFDH}(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the FFDH heuristic and $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. The multiplicative constant 1.7 in (2.2) is smaller than that of the NFDH algorithm—hence the FFDH algorithm generally performs better than the NFDH algorithm for large packing lists. In fact for all packing lists $\text{FFDH}(\mathcal{L}) \leq \text{NFDH}(\mathcal{L})$ [30]. A pseudocode listing for the FFDH algorithm is given as Algorithm 2.

Example 2.2 *The rectangles are ordered in the same manner as in the NFDH algorithm, according to non-increasing height as* $\{L_5, L_7, L_4, L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$. *The packing produced is similar to the NFDH algorithm packing, except that rectangle* L_6 *is packed on the lowest level with sufficient space, which is the third level. A total packing height of 19 units is obtained for the example instance in Table 1.2, as shown in Figure 2.1(b).* ■

Because lines 6–14 of Algorithm 2 have constant time complexity and the while-loop on line 7 does not depend on n, the for-loop spanning lines 5–18 has a time complexity of $O(n)$. Line 1, lines 3–4 and line 19 also have constant time complexity. Line 2 therefore dominates the time complexity of the algorithm and has a worst-case time complexity of $O(n \log n)$ when using an efficient sorting procedure such as the merge-sort algorithm [66]. Consequently, the overall worst-case time complexity of the FFDH algorithm is also $O(n \log n)$.

Algorithm 2 The first fit decreasing height (FFDH) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height of a packing obtained in the strip.

1: level $\leftarrow 0$; $h(\text{level}) \leftarrow 0$; $i \leftarrow 1$; LevelNum $\leftarrow 1$
2: Renumber the rectangles in non-increasing order by height such that $h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$
3: Pack rectangle L_i left justified at the bottom of the strip; $h(\text{level}+1) \leftarrow h(L_i)$
4: **for** $i = 2, \ldots, n$ **do**
5: search all levels (starting with the bottom) for the lowest with sufficient space
6: **if** such a level exists **then**
7: pack rectangle L_i left justified on that level
8: **else** [there is insufficient space in all existing levels]
9: LevelNum $\leftarrow$ LevelNum $+ 1$; level $\leftarrow$ LevelNum; $h(\text{level}) \leftarrow h(\text{level}-1) + h(L_i)$
10: pack rectangle on new level
11: **end if**
12: **end for**
13: **print** $H = h(\text{level})$

2.1.1.3 The best fit decreasing height algorithm

The *best fit decreasing height* (BFDH) algorithm [26] dates from 1990 and may also be applied to problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

It is analogous to the FFDH algorithm, except that in this algorithm rectangles are placed left justified towards the right of the last rectangle packed on the level with *minimum residual horizontal space*. This means that, to pack the next rectangle, all existing levels are searched for sufficient space and the area of the horizontal space towards the right of the level packing that would remain unutilised if the rectangle were to be placed in any of the levels is computed. The rectangle is placed on the level that leaves the smallest horizontal space. Dogrusoz [38] found the asymptotic performance bound of the BFDH algorithm also to be

$$\text{BFDH}(\mathcal{L}) \leq 1.7\ \text{OPT}(\mathcal{L}) + 1 \tag{2.3}$$

in the limit as $n \rightarrow \infty$, where $\text{BFDH}(\mathcal{L})$ denotes packing height achieved for the list of rectangles $\mathcal{L}$ by the BFDH heuristic and $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. Coffman [27] performed an asymptotic average-case performance analysis[2] to establish the expected value of the height of a packing produced by the BFDH algorithm as

$$E[\text{BFDH}(\mathcal{L})] = n/4 + \Theta(\sqrt{n}\ \log^{3/4} n) \tag{2.4}$$

[2] A probabilistic analysis is performed in terms of expected performance (*i.e.* the expected height of a packing produced by a given algorithm, where the expectation is computed over all $w(L_i)$ and $h(L_i)$, $1 \leq i \leq n$), in the limit as $n \rightarrow \infty$.

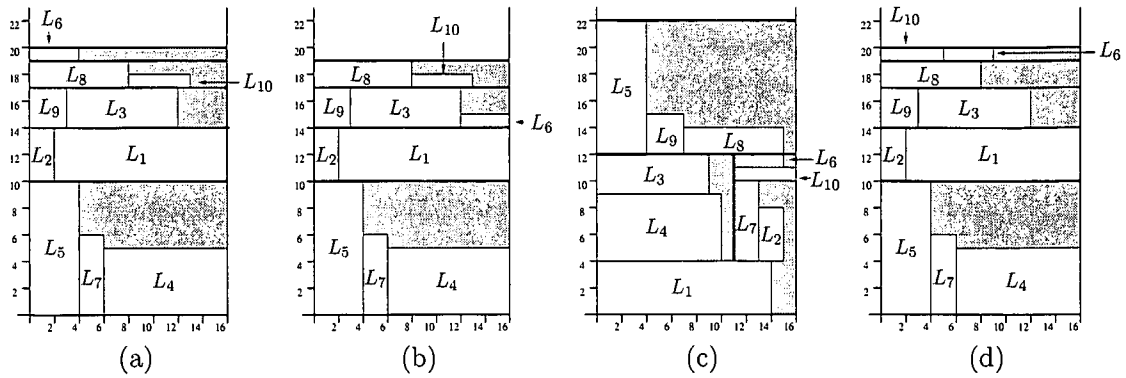

Figure 2.1: *Comparison of packing heights produced by different algorithms when applied to the list of rectangles in Table 1.2: (a) the NFDH algorithm packing, (b) the FFDH, BFDH and KP01 packings, (c) the SF packing and (d) the AlgorithmJOIN packing.*

as $n \to \infty$, where n is the number of rectangles in the list $\mathcal{L}$, where BFDH($\mathcal{L}$) denotes packing height achieved for the list of rectangles $\mathcal{L}$ by the BFDH heuristic, where $E[\cdot]$ denotes the expected value operator and where Θ denotes the asymptotically tight order notation (described in Appendix A). Coffman [29] assumed that all $2n$ variables $w(L_1), \ldots, w(L_n), h(L_1), \ldots, h(L_n)$ are independent and uniformly random samples from the interval $[0, 1]$ and that the strip width is 1. The average-case analysis in (2.4) seeks to find a function $f(n)$ for expressing the expected packing height obtained by an algorithm such that

$$E[\mathrm{A}(\mathcal{L})] = \sum_{i=1}^{n} w(L_i)h(L_i) + \Theta(f(n)), \tag{2.5}$$

where $\mathrm{A}(\mathcal{L})$ is the packing height produced by an algorithm A for the list $\mathcal{L}$ and $\Theta(f(n))$ represents the wasted area of the packing [38]. The first term in (2.4) is $n/4$ because the n rectangles are assumed to be independent; hence the packing produced on average by any algorithm must occupy at least $n/4$ units of space. This is because both the average width and average height of the rectangles in the uniform model is $1/2$—therefore the average area of a rectangle is $1/4$. For proof of (2.4), the reader is referred to [27]. The algorithm appears in pseudocode form as Algorithm 3.

Example 2.3 *When the BFDH algorithm is applied to the example instance in Table 1.2, a total packing height of 19 units is also obtained. The packing produced is the same as that produced by the FFDH algorithm, as shown in Figure 2.1(b). Rectangle L_6 is again packed on the third level, which is the level with minimum residual horizontal space.* ◙

The overall worst-case time complexity of the BFDH algorithm is also $O(n \log n)$. This is due to the dominant worst-case complexity in line 2 of $O(n \log n)$ when using an efficient sorting procedure such as the merge-sort algorithm [66], since lines 1, 3–4 and 6–17 have constant time complexity. The for-loop spanning lines 5–19 has worst-case time complexity $O(n)$.

Algorithm 3 The best fit decreasing height (BFDH) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.
Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.
Output: The height of a packing obtained in the strip.

1: level $\leftarrow 0$; h(level) $\leftarrow 0$; $i \leftarrow 1$; LevelNum $\leftarrow 1$
2: Renumber the rectangles in non-increasing order by height such that $h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$
3: Pack rectangle L_i left justified at the bottom of the strip; $h(\text{level}+1) \leftarrow h(L_i)$
4: **for** $i = 2, \ldots, n$ **do**
5: search all existing levels for the level with sufficient space and has minimum residual horizontal space
6: **if** such a level exists **then**
7: pack rectangle L_i left justified
8: **else** [there is insufficient space in all existing levels]
9: create a new level above the top-most level and pack rectangle L_i
10: LevelNum $\leftarrow$ LevelNum $+ 1$; level $\leftarrow$ LevelNum; h(level) $\leftarrow h(\text{level}-1) + h(L_i)$
11: **end if**
12: **end for**
13: **print** $H = h$(level)

2.1.1.4 The knapsack algorithm

The *knapsack* (KP01) algorithm [78] dates from 1998 and was originally developed as a first phase in solving a bin packing problem. Since bin packing is beyond the scope of this dissertation, the KP01 algorithm is described here in the context of strip packing for problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

In the KP01 algorithm, rectangles are pre-ordered according to non-increasing height. A level is initialised by packing a rectangle, L_{j*} (say), with greatest height amongst all the remaining rectangles. After initialisation, the knapsack problem,

$$\left.\begin{aligned} &\text{maximise} && \sum_{i=1}^{n_1} h(L_i)w(L_i)x_i, \\ &\text{subject to} && \sum_{i=1}^{n_1} w(L_i)x_i \leq W - w(L_{j*}), \\ & && x_i \in \{0,1\} \quad (i = 1, \ldots, n_1), \end{aligned}\right\} \tag{2.6}$$

is solved for the particular level, where n_1 represents the number of unpacked rectangles at each stage of the packing. Clearly, before any packing takes place $n_1 = n - 1$ since the tallest rectangle has initialised the level. The solution of the knapsack problem identifies those rectangles that should be packed on the particular level (*e.g.* $x_2 = 1, x_3 = 0$ means that rectangle L_2 should be packed and L_3 should not be packed on the particular level in question). The list of rectangles $\mathcal{L}$ is updated by removing all rectangles selected and the value of n_1 in (2.6) decreases as more rectangles are packed and more levels created. The algorithm continues in this manner taking all unpacked rectangles into consideration, until all rectangles are packed. A computational analysis was performed by applying the

KP01 algorithm to bin packing benchmarks [78]. In that analysis it was found that in general the KP01 algorithm outperformed other bin packing heuristics from the literature when comparing the ratios of the solution objective function values obtained to known lower bounds. However, Lodi [75] reckons that it should be possible to analyse the KP01 algorithm theoretically for strip packing by utilising the wealth of known results on knapsack problems. The algorithm is given in pseudocode form as Algorithm 4.

Example 2.4 *The rectangles are ordered according to non-increasing height* $\{L_5, L_7, L_4, L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$. *Rectangle* L_5 *initialises the first level and upon solving the knapsack problem,*

$$\left.\begin{array}{ll} \text{maximise} & \sum_{i=1}^{9} h(L_i)w(L_i)x_i, \\ \text{subject to} & \sum_{i=1}^{9} w(L_i)x_i \leq W - w(L_5), \\ & x_i \in \{0,1\} \quad (i = 1,\ldots,9), \end{array}\right\} \tag{2.7}$$

rectangles L_7 *and* L_4 *are selected to be packed on the first level. The list is then updated to* $\{L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$. *The first unpacked rectangle* L_2 *initialises the second level and the results of the knapsack,*

$$\left.\begin{array}{ll} \text{maximise} & \sum_{i=1}^{6} h(L_i)w(L_i)x_i, \\ \text{subject to} & \sum_{i=1}^{6} w(L_i)x_i \leq W - w(L_2), \\ & x_i \in \{0,1\} \quad (i = 1,\ldots,6), \end{array}\right\} \tag{2.8}$$

indicate that only one rectangle L_1 *is selected for packing on this level. The same procedure continues until all rectangles are packed and a total packing height of 19 units is obtained, as shown in Figure 2.1(b).* ∎

Line 1 of Algorithm 4 has a worst-case time complexity of $O(n \log n)$ when using an efficient sorting procedure such as the merge-sort algorithm, while lines 1, 4–5 and 7–8 all have constant time complexity. The body of the while–loop has a worst-case time complexity of $O(n)$. However, solving the KP01 instance on line 6 has pseudo-polynomial time[3] complexity of $O(nK)$ [34] where $K = W - w(L_j*)$ is the total width per level not to be exceeded. Hence the overall worst-case time complexity of the KP01 algorithm is $O(nK)$.

2.1.1.5 The split fit algorithm

In 1980 Coffman *et al.* [30] developed an algorithm called the *split fit* (SF) algorithm, which is slightly more complicated than the NFDH and FFDH algorithms, but whose

[3]For algorithms whose running times not only depend on the problem size, but also on the actual contents of the problem instance, this gives rise to pseudo-polynomial running times. An algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input (which is exponential in the length of the input–its number of digits (or bits)) [119].

Algorithm 4 The knapsack (KP01) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height of a packing obtained in the strip.

1: Order the rectangles according to non-increasing height such that $h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$
2: level $\leftarrow 0$
3: **while** there are unpacked rectangles **do**
4: pack first unpacked rectangle, L_i say
5: $h(\text{level}) \leftarrow h(\text{level}) + h(L_i)$
6: solve KP01 instance
7: pack selected rectangles
8: level $\leftarrow$ level $+ 1$
9: **end while**
10: **print** $H = h(\text{level})$

performance is slightly better. The SF algorithm was also developed for strip packing problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

In the SF algorithm, all rectangles have width $1/m$ or less, where $m \geq 1$ is the largest integer satisfying this constraint. The list of rectangles $\mathcal{L}$ to be packed is then partitioned into two sublists $\mathcal{L}_1$ and $\mathcal{L}_2$, where $\mathcal{L}_1$ contains rectangles of width greater than $1/(m+1)$, while $\mathcal{L}_2$ contains rectangles that have width at most $1/(m+1)$. The rectangles in $\mathcal{L}_1$ are then packed into the strip, using the FFDH algorithm. In the FFDH algorithm, the rectangles that are packed on a level form a *block* of the packing. The blocks of the packing are then rearranged so that all blocks having total width greater than $(m+1)/(m+2)$ are below those blocks of width at most $(m+1)/(m+2)$. This rearrangement creates a region $\mathcal{R}$ of width $1/(m+2)$, which is the region just above rectangles of width greater than $(m+1)/(m+2)$ and to the right of those of width at most $(m+1)/(m+2)$. The rectangles in $\mathcal{L}_2$ are then packed into the region $\mathcal{R}$ using the FFDH algorithm and if there is insufficient space, they are packed above the $\mathcal{L}_1$ rectangles. The height of the region $\mathcal{R}$ is the sum of the heights of the blocks of width at most $(m+1)/(m+2)$.

This algorithm was found to have an asymptotic performance bound of

$$\text{SF}(\mathcal{L}) \leq (m+2)/(m+1)\ \text{OPT}(\mathcal{L}) + 2, \tag{2.9}$$

where $\text{SF}(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the SF algorithm and $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. The multiplicative constant $(m+2)/(m+1)$ in (2.9) is best possible. Clearly, in the SF algorithm the condition that once placed, a rectangle may not be shifted is violated, since blocks are moved around, which accounts for the typically superior performance of the SF algorithm compared to the NFDH and FFDH algorithms. The SF algorithm is expected to achieve better performance as $m \to \infty$ because as the value of m increases, the multiplicative constant decreases. In fact, the SF algorithm becomes exact in the limit as $m \to \infty$. A pseudocode listing for the SF algorithm follows as Algorithm 5.

Algorithm 5 The split fit (SF) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height of a packing obtained in the strip.

1: Let $m \geq 1$ be the largest integer for which all rectangles in $\mathcal{L}$ have width at most $1/m$
2: Partition the list of rectangles $\mathcal{L}$ into two sublists $\mathcal{L}_1$ and $\mathcal{L}_2$ such that $\mathcal{L}_1$ is a list of rectangles of width greater than $1/(m+1)$, while $\mathcal{L}_2$ is a list of rectangles of width at most $1/(m+1)$
3: Pack the $\mathcal{L}_1$ rectangles into the strip, using the FFDH algorithm
4: Rearrange the blocks of this packing such that those of width greater than $(m+1)/(m+2)$ are below those of width at most $(m+1)/(m+2)$
5: Pack rectangles of width at most $1/(m+2)$ into the region $\mathcal{R}$ using FFDH algorithm such that no rectangle overlaps the top of $\mathcal{R}$ and those failing to fit in $\mathcal{R}$ are packed above the packing of $\mathcal{L}_1$
6: Output the height of the strip, found by adding the height of each level

Example 2.5 *Without loss of generality, the strip width is first scaled from* $W = 16$ *to* $W = 1$. *Hence the scaled rectangle dimensions in Table 1.2 reduce to those shown in Table 2.1. It is then evident that* $m = 1$ *is the largest integer for which all rectangles have width at most* $1/m$. *Partitioning* $\mathcal{L}$ *into the two required sublists results in* $\mathcal{L}_1 = \{L_4, L_1, L_3\}$ *and* $\mathcal{L}_2 = \{L_5, L_7, L_2, L_9, L_8, L_{10}, L_6\}$. *The result of this packing is depicted in Figure 2.1(c) with a total packing height of 22 units. Rectangles* L_7, L_2, L_{10}, L_6 *are packed in the region* $\mathcal{R}$, *which is indicated by means of the dashed vertical line, while rectangles* L_5, L_9, L_8 *are packed above the* $\mathcal{L}_1$ *rectangles, since they do not fit into region* $\mathcal{R}$. ∎

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}
Width, $w(L_i)$	0.8750	0.1250	0.5625	0.6250	0.2500	0.2500	0.1250	0.5000	0.1875	0.3125
Length, $h(L_i)$	0.2500	0.2500	0.1875	0.3125	0.6250	0.0625	0.3750	0.1250	0.1875	0.0625

Table 2.1: *Rectangle dimensions from Table 1.2 scaled such that* $\{w(L_i), h(L_i)\} \in (0, 1]$ *and the strip width is 1.*

The overall worst-case time complexity of the SF algorithm is $O(n \log n)$. This is because lines 1 and 2 of Algorithm 5 have a worst-case time complexity of $O(n)$ when using a procedure such as the linear search algorithm, while lines 3 and 5 dominate the complexity of the algorithm by having worst-case time complexity of $O(n \log n)$. Line 6 has constant time complexity.

2.1.1.6 AlgorithmJOIN

In 2003 Martello *et al.* [86] developed a heuristic, called *AlgorithmJOIN*, to solve packing problems of the form

2D	R	SP	Off	MiS	0,0,0,1

approximately. The list of rectangles is first ordered according to non-increasing height. The resulting list is then scanned for pairs of consecutive rectangles L_j and L_{j+1} (say),

which differ in height by no more than a given fraction γ. If such a pair is found and if $w(L_j)+w(L_{j+1}) \leq W$, then the pair is replaced by a "super rectangle" of height $h(L_j)$ and width $w(L_j) + w(L_{j+1})$ and the scan proceeds to rectangle L_{j+2}. The NFDH and FFDH algorithms are then executed on the resulting problem instance. From the best solution obtained, a feasible packing of the original instance is reconstructed. Typical values of γ range between 0 and 10%, reflecting the percentage difference of rectangle heights—since the algorithm is effective when there is a small difference in height between the conjoined rectangles.

A variation of AlgorithmJOIN is that it may also be executed by sorting the rectangles according to non-increasing width. In this case the super rectangles are constructed by vertically joining pairs of consecutive rectangles whose widths differ by no more than a given constant γ. The performance of AlgorithmJOIN was not analysed theoretically by Martello *et al.* [86], because the algorithm was used within an exact procedure to provide an inital feasible solution. A pseudocode listing of AlgorithmJOIN follows as Algorithm 6.

Algorithm 6 AlgorithmJOIN

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$, the constant γ as a percentage and the strip width W.

Output: The height of a packing obtained in the strip.

1: Renumber the rectangles according to non-increasing height such that $h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$
2: $j = 1$
3: **while** $j + 1 \leq n$ **do**
4: **if** $\frac{h(L_j)-h(L_{j+1})}{h(L_j)} \times 100 \leq \gamma$ **and** $w(L_j) + w(L_{j+1}) \leq W$ **then**
5: $w(L_j) = w(L_j) + w(L_{j+1})$
6: $j \leftarrow j + 2$
7: **else**
8: $j \leftarrow j + 1$
9: **end if**
10: **end while**
11: Execute the NFDH and FFDH algorithms to pack the rectangles (See Algorithms 1 and 2)
12: From the best solution, construct a feasible packing of the original instance
13: Output the height of the strip, found by adding the height of each level

Example 2.6 *The rectangles are first ordered according to non-increasing height as* $\{L_5, L_7, L_4, L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$. *Suppose* $\gamma = 5\%$, *so that the list is scanned for pairs of consecutive rectangles which differ in height by no more than 5%. Three "super rectangles" whose combined width is at most 16 are formed between rectangles* $\{L_2, L_1\}$, $\{L_9, L_3\}$ *and* $\{L_{10}, L_6\}$. *If super rectangles are denoted by a double subscript,* $L_{i,j}$, *which indicates a combination of rectangles* L_i *and* L_j, *then the NFDH and FFDH algorithms both yield a total packing height of 20 units for the list* $\{L_5, L_7, L_4, L_{2,1}, L_{9,3}, L_8, L_{10,6}\}$, *as shown in Figure 2.1(d).* ∎

Lines 2 and 4–8 of Algorithm 6 have constant time complexity. Hence the body of the while–loop spanning lines 4–8 has a worst-case time complexity of $O(n)$. The overall

worst-case time complexity of AlgorithmJOIN is also $O(n \log n)$, because of the dominating worst-case time complexity of lines 1 and 11. Lines 12–13 also have constant time complexity.

2.1.1.7 The floor ceiling no rotation algorithm

The *floor ceiling no rotation* (FCNR) algorithm [79] was developed in 1999 for problems of the form

2D	R	SP	Off	MiS	0,0,0,*

.

In this algorithm, rectangles are pre-ordered according to non-increasing height. Within a level, a *floor* is defined as the horizontal line coinciding with the bottom edges of rectangles packed on the level, while a *ceiling* is a horizontal line coinciding with the upper edge of the tallest rectangle packed on that level, as shown in Figure 2.2(a). If there is sufficient space to accommodate a rectangle on a floor then the rectangle is said to be *floor feasible*. Rectangles are packed on the floor from left to right, the left-hand edge of each rectangle coinciding with the right-hand edge of the previous rectangle. If, when packing a floor feasible rectangle, the distance between the top edge of the rectangle and the ceiling is insufficient to accommodate any of the unpacked rectangles, then the right-hand edge of such a rectangle forms a *left boundary* (s_1 in Figure 2.2(b) is a left boundary). The first rectangle to be placed on a ceiling (because of insufficient space on the floor) is packed with its right-hand edge coinciding with the right-hand edge of the strip and such a level is said to be *ceiling-initialised*. Before a level is ceiling initialised, the right-hand edge of the strip is referred to as the *right boundary* and rectangles are packed on a ceiling from right to left, producing a non-guillotineable packing (see Figure 2.2(a)).

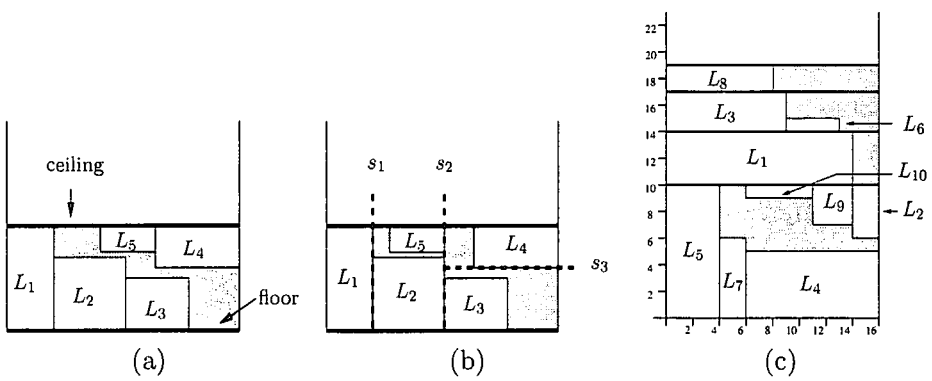

Figure 2.2: *Floor-Ceiling-No-Rotation algorithm demonstration generating; (a) a non-guillotineable packing and (b) a guillotineable packing. The packing in (c) is produced when applying FCNR algorithm to the rectangles in Table 1.2.*

Alternatively, to generate a guillotine packing, a series of vertical cuts (s_1 and s_2 in Figure 2.2(b)) are performed along the right-hand edges of rectangles packed on the floor, thereby forming slices. The last right-most slice is determined by a horizontal cut s_3 extending from the right-hand strip boundary, along the lower edge of the rectangle initialising the ceiling, L_{init} (say), up to a point where it coincides with the right-hand edge of a rectangle packed on the floor. If, on the ceiling, a rectangle does not fit within this slice, a gap remains while packing resumes in the next slice which is to the left of the previous

slice and right justified along the new right-hand boundary (this is the right-hand edge of the slice; in Figure 2.2(b) s_2 becomes the new right-hand boundary and rectangle L_5 is packed right justified along this boundary).

The FCNR algorithm is based on the same principle as the BFDH algorithm (see §2.1.1.3), in that a rectangle is packed into the level with minimum residual horizontal space. On the floor, the residual horizontal space is the distance between the right-hand edge of a rectangle and the right-hand edge of the strip. On the ceiling, the residual horizontal space is the distance between the left boundary and the left-hand edge of a rectangle. Ceiling initialisation is always preferred over floor packings, because they delay creation of new levels. The residual horizontal ceiling space on each level is computed first and if none of the rectangles can initialise or be packed on the ceiling, the residual floor space on each level is computed. A new level is created when none of the rectangles can fit onto a ceiling or floor in all existing levels. Computational analyses were carried out for bin packing by Lodi [79] and it was found that new algorithms they had proposed for solving bin packing problems, outperformed other heuristics from the literature. However, for instances where this was not possible, the floor-ceiling approach was the best. In this dissertation, such analyses will be carried out in the context of strip packing problems. The algorithm is given in pseudocode form as Algorithm 7.

Algorithm 7 The floor-ceiling no rotation (FCNR) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.
Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.
Output: The height of a packing obtained in the strip.

```
1: Renumber the rectangles according to non-increasing height such that h(L_1) ≥ h(L_2) ≥
   ... ≥ h(L_n)
2: for i = 1, ..., n do
3:   if L_i is ceiling feasible then
4:     pack L_i on ceiling with minimum residual space
5:   else [L_i is not ceiling feasible]
6:     if L_i is floor feasible then
7:       pack L_i on the floor with minimum residual space
8:     else [L_i is not floor feasible]
9:       level ← level + 1
10:    end if
11:  end if
12: end for
13: Output the height H of the strip, found by adding the height of each level
```

Example 2.7 *When the FCNR algorithm is applied to the rectangles in Table 1.2, a total packing height of 19 units is obtained, as shown in Figure 2.2(c). The rectangles* $\{L_5, L_7, L_4\}$ *are packed on the floor of the first level. Because rectangle* L_2 *is not floor feasible, it initialises the ceiling packing on the first level. The next rectangle* L_1 *does not fit on either the ceiling or floor of the first level, hence a new level is created and it is packed there. To pack rectangle* L_9*, the ceiling of the first level is searched first for sufficient space and there is enough room; hence it is placed next to rectangle* L_2*.*

The packing proceeds in this manner until all rectangles are packed. In this example, a guillotineable packing is produced. ∎

Line 1 of Algorithm 7 has a worst-case time complexity of $O(n \log n)$ when using an efficient sorting procedure such as the merge-sort algorithm, while line 2, line 9 and line 13 have constant time complexity. Lines 4 and 7 may each be implemented in time $O(n^2)$ when using a best fit algorithm. Since the for–loop spanning these lines 3–11 has a worst-case time complexity of $O(n)$, the overall worst-case time complexity of the FCNR algorithm is $O(n^3)$.

2.1.2 Plane algorithms

Plane algorithms refer to procedures by which rectangles are packed into any suitable position in the strip without partitioning the strip into levels. They are sometimes also referred to as non-level or non-shelf algorithms in the literature.

2.1.2.1 The Sleator algorithm

Sleator [108] developed an algorithm in 1980 to solve packing problems of the form

2D	R	SP	Off	MiS	0,0,0,0

.

In this algorithm, the rectangle dimensions are in the range $(0, 1]$, without loss of generality and they are packed into a strip of width 1. Rectangles of width greater than $1/2$ are selected and stacked from the bottom of the strip upwards to a height of H_{stack}. The remaining unpacked rectangles are then sorted according to non-increasing height—suppose the height of the tallest unpacked rectangle is H_{tall}. The rectangles are packed from the left to right of the strip until there is insufficient space or there are no more rectangles to pack. At this point the strip is partitioned, starting at a height of H_{stack}, into two equal sides (left and right segments), each of width $1/2$. The height of the left segment denoted by H_{left} is initially equal to H_{tall}, while the height of the right segment, denoted by H_{right}, is given by the first rectangle packed within the right segment. However, the rectangle initialising the height of the right segment may belong partially to both segments and if there is such a rectangle, the packing produced becomes non-guillotineable (rectangle 4 in Figure 2.3(a) is such a rectangle). The heights of each segment correspond to the upper edge of the tallest rectangle packed on either segment. Before packing commences, a choice should be made into which segment packing should take place. Depending on the heights H_{left} and H_{right}, packing commences from left to right in the segment of minimum height (*i.e.* in the left (resp. right) segment, packing starts from the left-hand edge (resp. middle) of the strip and ends in the middle (resp. right-hand edge) of the strip).

The Sleator algorithm is able to pack a set of rectangles with an asymptotic worst-case performance of

$$\text{Sleator}(\mathcal{L}) \leq 2.5\ \text{OPT}(\mathcal{L}) \tag{2.10}$$

in the limit as $n \to \infty$, where $\text{Sleator}(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the Sleator algorithm and $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. The reader is referred to [108] for a detailed proof of this worst-case bound. The algorithm is given in pseudocode form as Algorithm 8.

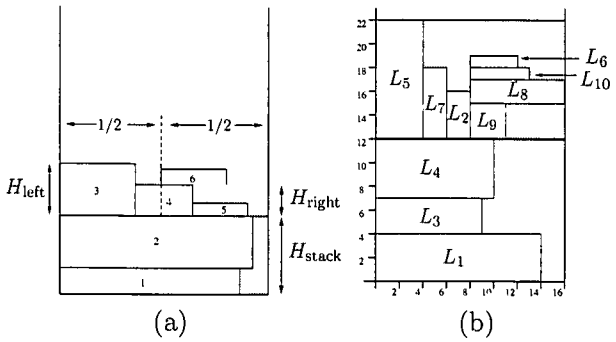

Figure 2.3: *Illustration of the Sleator algorithm: (a) partitioning of the strip into the different segments along with their associated heights and (b) packing produced when the algorithm is applied to the list of rectangles in Table 1.2.*

Example 2.8 *The rectangle dimensions in Table 1.2 are rescaled and reduced to those shown in Table 2.1 such that* $W = 1$ *and* $(w(h_i), h(h_i)) \in (0, 1]$. *The rectangles of width greater than 1/2* (L_1, L_3, L_4) *are stacked in any order from the bottom of the strip upwards. The remaining unpacked rectangles are ordered according to non-increasing height and packed from the left to the right of the strip above* L_4 *until there is insufficient space to pack rectangle* L_8. *The strip is then partitioned into two equal segments along the right edge of rectangle* L_2 *and left edge of rectangle* L_9. *Because* $\min\{H_{\text{left}}, H_{\text{right}}\} = H_{\text{right}}$, *packing resumes from the left to the right of the right segment where only rectangle* L_8 *fits. The right segment still has minimum height, therefore rectangle* L_{10} *is packed there and is followed by rectangle* L_6. *A total packing height of 22 units is obtained for the example instance, as shown in Figure 2.3(b).* □

The Sleator algorithm has a worst-case time complexity of $O(n \log n)$ [108]. The complexity is dominated by the sorting stage in line 4 of Algorithm 8. When an efficient sorting procedure, such as the merge-sort algorithm, is used this step has a time complexity of $O(n \log n)$. Line 1 and the while-loop spanning lines 10 to 12 have a running time of $O(n)$. The rest of the packing stages have a constant time complexity.

2.1.2.2 The Burke algorithm

In 2004, Burke *et al.* [18] developed an algorithm for packing problems of the form

2D	R	SP	Off	MiS	1,0,0,0

producing non-guillotineable packings in which 90 degree rotations of rectangles are allowed. Because rotation is beyond the scope of this dissertation, the Burke algorithm is described here for problems in which rotations are disallowed and are of the form

2D	R	SP	Off	MiS	0,0,0,0

According to Kendall [69], the performance bound as well as time complexity of the algorithm were not analysed.

A best-fit methodology is used (see §2.1.1.3) and it involves two basic steps which are carried out before packing each rectangle in the list: (1) the method dynamically selects

Algorithm 8 The Sleator algorithm

Description: Pack a list of rectangles into an open ended strip of fixed width and infinite potential height, to minimise the total packing height. The list $\mathcal{L}$ of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height of a packing obtained in the strip.

1: Partition $\mathcal{L}$ into two sublists $\mathcal{L}_1$ and $\mathcal{L}_2$ consisting of rectangles of width greater than $1/2$ and at most $1/2$ respectively.
2: Stack all the rectangles in $\mathcal{L}_1$ left justified on top of one another starting at the bottom of the strip. Compute H_{stack}
3: Packing will continue above H_{stack}
4: Sort the rectangles in $\mathcal{L}_2$ according to non-increasing height such that $h(L_i) \geq h(L_{i+1})$ for $i < n$
5: Let H_{tall} be the height of the tallest rectangle in list $\mathcal{L}_2$.
6: Pack the rectangles, left justified from the left to the right edge of the strip until there is insufficient space to pack a rectangle or all of the rectangles have been packed
7: Partition the strip with a vertical line into two equal segments. There is possibly one rectangle whose interior may be intercepted by the vertical line.
8: Let H_{right} and H_{left} be the height of the rectangle on the right (resp. left) half of the strip whose left (resp. right) edge is adjacent to the vertical line or whose interior is intercepted by the vertical line
9: **while** there are unpacked rectangles **do**
10: Draw horizontal lines of length half across the rectangles whose height is H_{left} and H_{right}
11: All subsequent packing will either be on the left or right segment of the strip
12: Select the segment with minimum height and pack rectangles from the edge of the strip to the vertical line until all rectangles have been packed or there is a rectangles which does not fit
13: **end while**
14: **print** $H = \max\{H_{\text{left}}, H_{\text{right}}\}$

the rectangle to be packed and (2) where it should be packed. The latter step involves finding the lowest available space (gap) where a rectangle may be placed, while the former requires the entire list of rectangles to be examined in order to determine the best fitting rectangle. As each of these two steps are performed before each rectangle is packed, it is obvious that this will have an impact on the execution time.

Finding the position of the lowest available gap

Before packing commences, the lowest available gap is the entire width of the strip, as shown in Figure 2.4(a). After some rectangles have been packed, a linear array is used to determine the position of the lowest available gap. Each element in the array represents the total height of the packing at that x-coordinate of the sheet. The smallest valued entry of the array corresponds to the lowest available gap and the width of the gap is given by the number of consecutive elements in the array of that particular value (see Figure 2.4(b)).

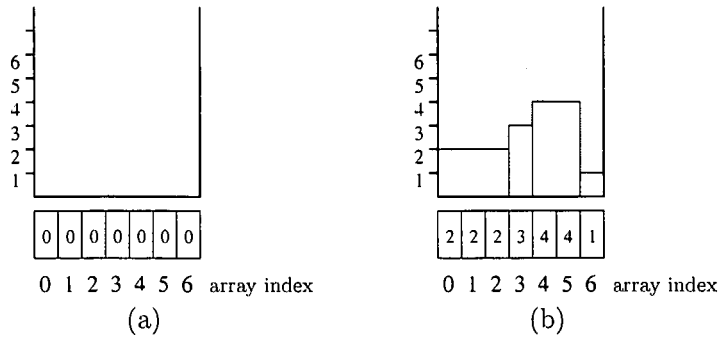

Figure 2.4: *Linear array representation of a partial packing in the Burke algorithm: (a) the elements of the array are all zeros showing that the lowest available gap is the entire width of the strip and (b) the lowest gap available is at $x = 6$ and has width 1.*

Selecting the best fitting rectangle

The list of rectangles is ordered according to non-increasing width before packing commences. Rectangles of equal widths are resolved by decreasing heights. There are three possible ways in which a rectangle may fit into a particular gap:

- The rectangle may have a width that fits exactly into the gap.
- There may be a rectangle whose width is smaller than the gap.
- No rectangle may fit the gap.

If the algorithm comes across a rectangle that fits perfectly into the gap, then it is selected immediately and the search stops. If there are several rectangles that fit exactly into the gap, then the rectangle with the largest area is selected. If a rectangle has width smaller than the gap, then the rectangle that consumes the largest portion into the gap is selected. If, on the other hand, none of the available rectangles fit into the gap, then the gap is written off as wastage. At this point, each neighbour of the gap is examined and the array elements that define the gap are raised to the height of the lowest neighbour. After a rectangle is selected, the next decision to be made is how the rectangle should be placed into the gap.

Placing a rectangle into a gap

There are three possible ways in which a rectangle that has been selected may be placed into a gap.

Place on left: A rectangle is placed on the left side of the gap.

Place next to tallest neighbour: The rectangles surrounding the gap are examined and the rectangle is placed next to the tallest neighbour. If the gap is defined by a rectangle and a strip side then the rectangle is placed next to the strip side.

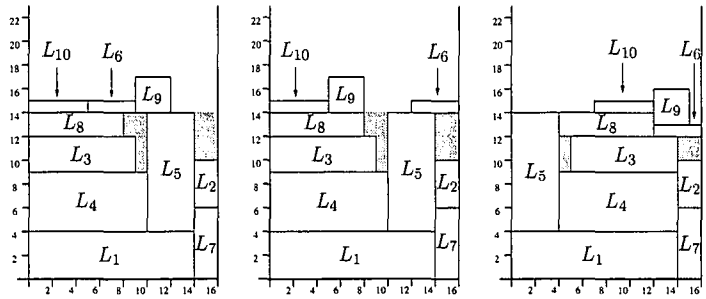

Figure 2.5: *Comparison of packing heights produced by different placement policies in the Burke algorithm, when applied to the list of rectangles in Table 1.2: (a) the leftmost placement, (b) tallest neighbour placement and (c) shortest neighbour placement.*

Place next to shortest neighbour: This is a mirror image of the tallest neighbour placing, because it places a rectangle next to the shortest neighbour. If the gap is defined by a rectangle and a strip side then the rectangle is placed next to the rectangle.

The algorithm is run for each of the placement methods and the one resulting in the shortest strip height after all rectangles have been packed is selected. Based on computational experiments, the algorithm was able to give solutions in less than two seconds when benchmarks of sizes ranging from 10 to 3152 rectangles were solved on a computer with an 850MHz processor and 128MB of RAM. The algorithm is given in pseudocode form as Algorithm 9.

Example 2.9 *In this example, only one placement policy, namely the leftmost placement, is described. The rectangles in Table 1.2 are sorted according to non-increasing width as* $\{L_1, L_4, L_3, L_8, L_{10}, L_6, L_5, L_9, L_7, L_2\}$. *The lowest gap is at* $x = 0$ *and rectangle* L_1 *is placed at this coordinate and it is removed from the list. The array elements* $x = 0$ *to* $x = 13$ *are then updated to 4, the height of rectangle* L_1. *The lowest gap is now at* $x = 14$ *and has width 2 into which rectangle* L_7 *fits exactly. The search stops and rectangle* L_7 *is removed from the list and placed into the gap. The array elements* $x = 14$ *to* $x = 15$ *are updated to 6. The lowest gap is at* $x = 0$ *and has width 14 into which rectangle* L_4 *is packed (the array elements* $x = 0$ *to* $x = 9$ *are updated to 9). The process is continued until all rectangles are packed. Three total packing heights of 17, 17, and 16 units are obtained respectively for the leftmost, tallest and shortest neighbour placement policies, as shown in Figures 2.5(a)-(c), hence the shortest neighbour placement policy is selected, because it yields the smallest packing height.* ∎

When using an efficient sorting procedure, such as the merge-sort algorithm, line 1 in Algorithm 9 has a time complexity of $O(n \log n)$. Finding the lowest gap (line 4), raising array elements (lines 7 and 9), packing a best fitting rectangle (line 6) and comparison of heights (line 14) all have a constant time complexity. Since $n-i+1$ rectangles have to be considered during a sequential search in iteration i of the while-loop spanning line 3–11 of the algorithm, the while-loop has time complexity $\sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n} i = n(n+1)/2$. Consequently, the worst-case time complexity of the Burke algorithm without rotation is $O(n^2)$.

Algorithm 9 The Burke algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height of a packing obtained in the strip.

1: Sort the rectangle according to non-increasing width such that $w(L_1) \geq w(L_2) \geq \ldots \geq w(L_n)$
2: **for** each placement policy (leftmost, tallest neighbour, smallest neighbour) **do**
3: **while** Rectangles not packed **do**
4: Find lowest gap
5: **if** $w(L_i) \leq$ GapWidth **then**
6: Place best fitting rectangle using placement policy
7: Raise elements of array to appropriate height
8: **else**
9: Raise gap to height of the lowest neighbour
10: **end if**
11: **end while**
12: **end for**
13: The elements of the array with greatest entry give the total height of the packing
14: Compare total packing heights obtained by each placement policy and return the best solution

2.1.3 The Kenyon-Remila algorithm

In this section a theoretical algorithm is described which is significant in the sense that it provides insight into structural properties of strip packing problems. As early as 1981 Fernandez de la Vega and Lueker [44] introduced an approximation scheme[4] for solving bin packing problems, demonstrating that it is indeed possible to solve bin packing problems by means of approximation schemes—not all optimisation problems have approximation schemes. For any instance of a one-dimensional bin packing problem $\mathcal{L}$ and a fixed positive real number ϵ (used to eliminate small rectangles), the performance ratio

$$\mathrm{A}(\mathcal{L}) \leq (1+\epsilon)\mathrm{OPT}(\mathcal{L}) + O(\epsilon^{-2})$$

was obtained as $n \to \infty$, where $\mathrm{A}(\mathcal{L})$ denotes the number of bins utilised by the Fernandez de la Vega and Lueker algorithm and where $\mathrm{OPT}(\mathcal{L})$ is the minimum number of bins required to solve the bin packing instance $\mathcal{L}$. The execution time of this algorithm is linear in n and exponential in $1/\epsilon$. A year later, Karmarkar and Karp [68] expanded on this idea by introducing the notion of a *fractional bin packing problem* which may be solved in polynomial time and involves solving a linear programming relaxation, whose solution is interpreted in the sense that a bin can be used a fractional number of times instead of an integral number of times.[5] Several algorithms for the one-dimensional bin

[4] An approximation scheme as defined in [106] is a suboptimal approach that provably works fast and that provably yields solutions of very high quality. The word "scheme" is used to designate a class of approximation algorithms whose solution is guaranteed to be within a factor of $(1+\epsilon)$ of the optimal solution over all values of some parameter $0 < \epsilon < 1$. Some of the main areas and basic definitions in approximability are discussed further in Appendix A.

[5] Other techniques used in [68] for developing an approximation scheme for one-dimensional bin packing problem are discussed in Appendix B for the sake of completeness.

packing problem whose running times are polynomial functions of $1/\epsilon$ have subsequently been presented in [68].

In 1998 Fernandez and Zissimopoulos [45] developed an approximation scheme for strip packing whose performance ratio is within $(1+\epsilon)$ of the optimal solution, for any fixed parameter values $\delta, \epsilon > 0$, where the rectangle dimensions are bounded from below by the parameter δ and where the parameter ϵ is used to eliminate small rectangles. The execution time of this algorithm is an exponential function of $1/\epsilon$.

However, in 2000 Kenyon and Remila [70] presented an asymptotic fully polynomial time approximation scheme (AFPTAS)[6] for packing problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

The scheme is based on ideas in [44, 45, 68], some of which were described earlier in this section, and it is the first study of its kind for strip packing problems [104]. Given a list $\mathcal{L}_{general}$ of n rectangles to be packed into a strip, the algorithm uses a method of *elimination* to partition the list $\mathcal{L}_{general}$ into two sublists, $\mathcal{L}_{narrow}$ and $\mathcal{L}_{wide}$ based on the value of a parameter ϵ. By relaxing the constraints, a *fractional strip packing* is found, which allows for horizontal cuts on rectangles, in a new list $\mathcal{L}_{sup}$ derived from $\mathcal{L}_{wide}$ by applying a process of so-called *grouping*. A full strip packing of $\mathcal{L}_{sup}$ is then deduced from this fractional strip packing. The rectangles in $\mathcal{L}_{narrow}$ are finally added to this strip packing.

For any fixed constant $0 < \epsilon < 1$, the asymptotic performance ratio

$$\mathrm{KR}(\mathcal{L}) \leq (1+\epsilon)\mathrm{OPT}(\mathcal{L}) + O(1/\epsilon^2)$$

as $n \to \infty$ is associated with the algorithm, where $\mathrm{KR}(\mathcal{L})$ is the packing height achieved by the Kenyon-Remila algorithm for the list of rectangles $\mathcal{L}$ and $\mathrm{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. A proof of this performance measure may be found in [70].

The various steps of the algorithm are described below in some detail and the various phases of the working of the algorithm are illustrated by means of a series of examples, in which the scaled rectangle dimensions listed in Table 2.1 are to be packed into a strip of width 1.

The process of elimination

Suppose ϵ' is a real number between 0 and 1. Then the list $\mathcal{L}_{general}$ of n rectangles, $\{L_1, L_2, \ldots, L_n\}$, is partitioned into a list, $\mathcal{L}_{narrow}$, of rectangles whose widths are at most ϵ' (called narrow rectangles) and a list, $\mathcal{L}_{wide}$, of rectangles whose widths are greater than ϵ' (called wide rectangles). The algorithm first deals with the sublist of wide rectangles, whilst the narrow rectangles are only considered during the final stages of the algorithm.

Example 2.10 *The rectangles in Table 2.1 are first ordered according to non-increasing width to obtain the list* $\mathcal{L}_{general} = \{L_1, L_4, L_3, L_8, L_{10}, L_6, L_5, L_9, L_7, L_2\}$. *Suppose* $\epsilon' =$

[6] This is an algorithm that runs for any fixed $\epsilon > 0$ with asymptotic worst-case ratio $(1+\epsilon)$ as $n \to \infty$, and has a running time which is polynomial in both n and $1/\epsilon$ [65], where ϵ is a parameter used to eliminate small rectangles and n denotes the number of rectangle to be packed.

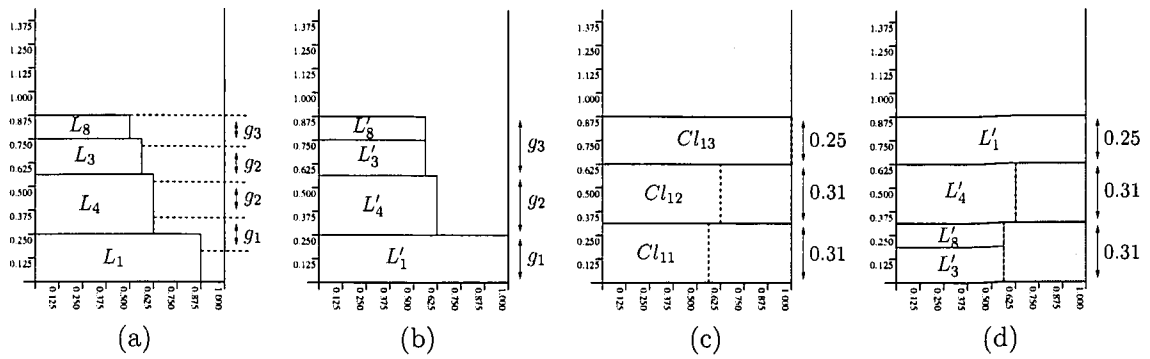

Figure 2.6: *From grouping to fractional strip packing: (a) The rectangles in $\mathcal{L}_{wide}$ are stacked left justified and divided into equal groups by means of threshold rectangles, (b) the width of the rectangles in each group are rounded up to the height of the threshold rectangles at the bottom of each group, (c) the strip is partitioned into horizontal levels and in each level different columns associated with the widths of the groups are drawn (where Cl_{uv} represents the u^{th} column of level v) and (d) a fractional strip packing is produced by filling up the columns with the corresponding group*

0.45. *Then the list $L_{general}$ is partitioned into the two sublists, $\mathcal{L}_{wide} = \{L_1, L_4, L_3, L_8\}$ and $\mathcal{L}_{narrow} = \{L_{10}, L_6, L_5, L_9, L_7, L_2\}$.* □

The process of grouping

A left–justified stack of the rectangles in $\mathcal{L}_{wide}$ is formed within the strip in order of non-increasing widths to obtain a total stack height $h(\mathcal{L}_{wide})$. The stack is partitioned into at most m groups, by means of at most $(m-1)$ distinct threshold rectangles, where a *threshold rectangle* is defined as a rectangle whose interior or lower boundary is intersected by the horizontal line $y = ih(\mathcal{L}_{wide})/m$, for some integral value of i between 1 and $(m-1)$ inclusive, where $m = \lceil (1/\epsilon')^2 \rceil$. The stack is thus divided into at most m regions of equal height. At this stage there is a choice between two conventions; one convention is to round up the widths of the rectangles in the first (bottom-most) group to 1, while another convention is to round up the widths of the rectangles in the first group to the width of the rectangle in the first level (*i.e.* the bottom-most rectangle). In both conventions, the widths of the rectangles in each subsequent group are rounded up to the width of the threshold rectangle at the bottom of that group. This process is referred to as *grouping* and defines another list, $\mathcal{L}_{sup}$, which approximates $\mathcal{L}_{wide}$ in the sense that $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$[7] and contains rectangles of at most m distinct widths. The process of grouping the rectangles according to their widths is essential in that it ensures that the number of distinct widths of the wide rectangles is bounded from above by m.

Example 2.11 *(Continuation of Example 2.10)*

The value $m = \lceil (1/0.45)^2 \rceil = 5$ is found for the choice of ϵ' in Example 2.10. The rectangles in $\mathcal{L}_{wide}$ are stacked left-justified in the strip to a height of 0.875 and then divided into three groups by means of so-called threshold rectangles.

[7]The relation $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$ is defined as an injection from $\mathcal{L}_{wide}$ to $\mathcal{L}_{sup}$ such that the rectangles in $\mathcal{L}_{wide}$ have smaller width and height than the associated rectangles of $\mathcal{L}_{sup}$.

In this case, all the rectangles are threshold rectangles as indicated by dashed horizontal lines in Figure 2.6(a). The stack of rectangles in $\mathcal{L}_{wide}$ has thus been partitioned into three groups; $g_1 = L_1$, $g_2 = L_4$, $g_3 = L_3, L_8$ forming a new list $\mathcal{L}_{sup} = \{g_1, g_2, g_3\}$.

The interior of rectangle L_4 is intersected twice (when $i = 2$ and $i = 3$). The question at this point is: how can rectangle L_4 be a threshold rectangle twice? According to Remila [104], this special case is resolved by assuming that rectangle L_4 is partitioned into 2 rectangles of equal widths (i.e. there are four widths, namely $0.8750, 0.6250, 0.6250, 0.5625$).

Using the first convention, the widths of rectangles in groups g_1, g_2, g_3 are rounded as shown in Figure 2.6(b), hence $\mathcal{L}_{sup}$ has rectangles of three distinct widths, $1, 0.6250$, 0.5625. The rectangles in the various groups of $\mathcal{L}_{sup}$ are denoted by L'_i since their widths have been rounded and a distinction has to be made between rectangles in $\mathcal{L}_{wide}$ and those in $\mathcal{L}_{sup}$. This shows that $m = 5$ is only an upper bound to the number of distinct widths that may be present in $\mathcal{L}_{sup}$. □

Determining a fractional strip packing

As mentioned, Kenyon and Remila [70] adapted certain basic ideas typically employed in 1D bin packing to the situation in 2D strip packing. They introduced the notion of *fractional strip packing* of a list of rectangles $\mathcal{L}$ as a packing of any list of rectangles $\mathcal{L}'$ obtained from $\mathcal{L}$ by subdividing some of the rectangles in $\mathcal{L}$ by means of horizontal cuts: the height of each rectangle $h(L_i)$ in $\mathcal{L}$ is replaced by a sequence $h(L_{i_1}), h(L_{i_2}), \ldots, h(L_{i_{k_i}})$ in $\mathcal{L}'$ such that $h(L_i) = \sum_j h(L_{ij})$. The following linear program is associated with the *fractional strip packing problem*:

$$\left.\begin{array}{lrcl} \text{Minimise} & z & = & \underline{1}.\underline{x}, \\ \text{Subject to} & \mathbf{A}\underline{x} & \geq & \underline{b}, \\ & \underline{x} & \geq & 0, \end{array}\right\} \tag{2.11}$$

where the objective function value, z, represents the total strip height, $\underline{1}$ is a q-vector of all ones, the j^{th} entry of the q-vector $\underline{x}$, namely x_j, is the strip height obtained when rectangles are packed using a so-called configuration C_j and $\mathbf{A}$ is the $m \times q$ matrix whose $(i,j)^{th}$ entry, $\mathbf{A}_{ij}$, denotes the number of occurrences of the width $w(L'_i)$ in configuration C_j. Here, m denotes the number of distinct widths of rectangles in $\mathcal{L}_{sup}$ to be packed and the i^{th} entry of the m-vector $\underline{b}$, namely b_i, is the sum of the heights of all rectangles of width $w(L'_i)$. A *configuration* is defined as a non-empty set of widths whose sum is less than or equal to 1. The widths are chosen from the m distinct widths and their sum is called the *width of the configuration*. All possible distinct configurations are computed beforehand through a brute force method which attempts all width combinations until a configuration is found.

After solving the linear program and thus obtaining a solution vector $\underline{x} = (x_1, x_2, \ldots, x_q)$, a *fractional strip packing* is constructed in the following manner:

- The strip (of width 1 and height $\sum_j x_j$) is partitioned into j horizontal levels, each of width 1 and of height x_j, $(1 \leq j \leq q)$.
- In the j^{th} level, $\mathbf{A}_{ij}$ columns of width $w(L'_i)$ and height x_j are drawn for each $1 \leq i \leq m$, provided that $\mathbf{A}_{ij} \neq 0$.

- For each $1 \leq i \leq m$, the columns of width $w(L'_i)$ are filled up with the input rectangles of width $w(L'_i)$ in a greedy manner, cutting the rectangles horizontally if necessary, so as to fill each column exactly up to height x_j. This means that a column may contain an entire rectangle or the bottom part of a rectangle which is too tall to fit into the column and had to be cut in such a way that the top part of the rectangle is placed in another column.

Example 2.12 *(Continuation of Examples 2.10 and 2.11)*

There exist three distinct configurations C_1, C_2 and C_3, based on the three distinct widths in $\mathcal{L}_{sup}$. They are listed in the table below.

	C_1	C_2	C_3
$\mathbf{A}_{1j}$	0	0	1
$\mathbf{A}_{2j}$	0	1	0
$\mathbf{A}_{3j}$	1	0	0
Widths of C_j	0.5625	0.6250	1.0000

Solving the linear programming problem,

$$\left.\begin{array}{lrcl} \text{Minimise} & 1.x_1 + 1.x_2 + 1.x_3 & = & z, \\ \text{Subject to} & 0.x_1 + 0.x_2 + 1.x_3 & \geq & 0.2500, \\ & 0.x_1 + 1.x_2 + 0.x_3 & \geq & 0.3125, \\ & 1.x_1 + 0.x_2 + 0.x_3 & \geq & 0.3125, \end{array}\right\} \tag{2.12}$$

the vector $\underline{x} = (0.3125, 0.3125, 0.25)$ is obtained. The strip of width 1 and height $x_1 + x_2 + x_3 = 0.875$ is partitioned into three levels of width 1 and heights of $0.3125, 0.3125, 0.25$ corresponding to configurations C_1, C_2, C_3 respectively. For each i and j satisfying $\mathbf{A}_{ij} \neq 0$ and $x_j \neq 0$, $\mathbf{A}_{ij}$ columns are drawn in each level of the strip (Figure 2.6(c)). Finally, each column of width $w(L'_i)$ is greedily filled up to a height of x_j by means of input rectangles of width $w(L'_i)$. From Figure 2.6(d), it is clear that rectangles in g_1 fit exactly into column 1 of level 3, rectangles in g_2 fit into column 1 of level 2, rectangles in g_3 fit into column 1 of level 1. A fractional strip packing of height $h = 0.875$ has therefore been constructed. □

Construction of a full strip packing from the fractional strip packing

Kenyon and Remila [70] extended a bin packing theorem to strip packing such that there exits an algorithm with positive *maximum permissible error* t whose running time is polynomial in $m, \sum_i b_i$ and t, giving a solution with at most m non-zero coordinates.

Lemma 2.1 *([70]) If $\mathcal{L}$ has a fractional strip packing $(x_1, \ldots, x_q)$ of height h and with at most $2m$ non-zero x_j values, then $\mathcal{L}$ has an (integral) strip packing of height at most $h + 2m$.* □

A full strip packing may be derived from the fractional strip packing as follows:

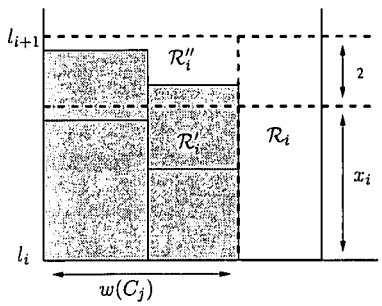

Figure 2.7: *A well structured level that results from the strip packing in the Kenyon-Remila algorithm. The top-most region of the level has height at most 2: one from solving the fractional strip packing with maximum permissible error of 1 and adding h_{max} which has height at most one to x_i [104].*

- Suppose the height $h = \sum_i x_i$ is obtained in the fractional strip packing with at most m non-zero coordinates x_i's and let h_{max} denote the maximum height of any rectangle of $\mathcal{L}_{general}$. Assume that the non-zero coordinates in the solution to (2.11) are $x_{1'}, x_{2'}, \ldots, x_{m'}$ where $m' \leq m$.
- Each configuration is taken in turn while filling the strip from the bottom upwards. Let $x_{j'} > 0$ denote the variable corresponding to the current configuration. Configuration C_j is packed between levels $l_j = (x_{1'} + h_{max}) + \ldots + (x_{j-1'} + h_{max})$ and $l_{j+1} = l_j + x_{j'} + h_{max}$. Initially, the first level corresponds to the bottom of the strip, where $l_1 = 0$.
- For each $1 \leq i \leq m'$, $\mathbf{A}_{ij}$ columns of width $w(L_i')$, ranging from level l_j to level l_{j+1} are drawn (if $\mathbf{A}_{ij} \neq 0$). These columns are denoted by $Cl+$.

The columns Cl_{uv} of width $w(L_i')$ and height x_j in the fractional strip packing may be associated with those columns $Cl+_{uv}$ of width $w(L_i')$ and height $x_{j'} + h_{max}$ in the full strip packing. Rectangles fully packed in Cl_{uv} and those whose lower boundaries are in Cl_{uv} and whose upper boundaries are in another column $Cl_{(u+1)v}$ are placed in $Cl+_{uv}$. This strip packing leaves a well-structured free space. Within level i there is a region which is fully packed, denoted by $\mathcal{R}_i'$, another region partially filled, denoted by $\mathcal{R}_i''$ and finally a region that is completely free, denoted by R_i, which is used to pack the narrow rectangles (those of width less than or equal to ϵ'). These regions are depicted in Figure 2.7, taken directly from [70]. Using the relation $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$, a packing of $\mathcal{L}_{wide}$ may be deduced by placing each rectangle of $\mathcal{L}_{wide}$ inside the position of the associated rectangle of $\mathcal{L}_{sup}$.

Example 2.13 *(Continuation of Examples 2.10, 2.11 and 2.12)*

Each configuration is taken in turn and the strip is filled from the bottom upwards. Since $x_{1'} > 0$, configuration C_1 is packed between $l_1 = 0$ and $l_2 = l_1 + x_{1'} + h_{max} = 0.9375$. Configuration C_2 is packed between $l_2 = 0.9375$ and $l_3 = l_2 + x_{2'} + h_{max} = 1.875$. Configuration C_3 is packed between $l_3 = 1.875$ and $l_4 = l_3 + x_{3'} + h_{max} = 2.75$. The various columns of different levels are shown in Figure 2.8(a). The columns are then filled up with groups of appropriate width from $\mathcal{L}_{sup}$, as shown in Figure 2.8(b). The packing of $\mathcal{L}_{wide}$ is then deduced by using the relation $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$ to obtain a strip packing of height

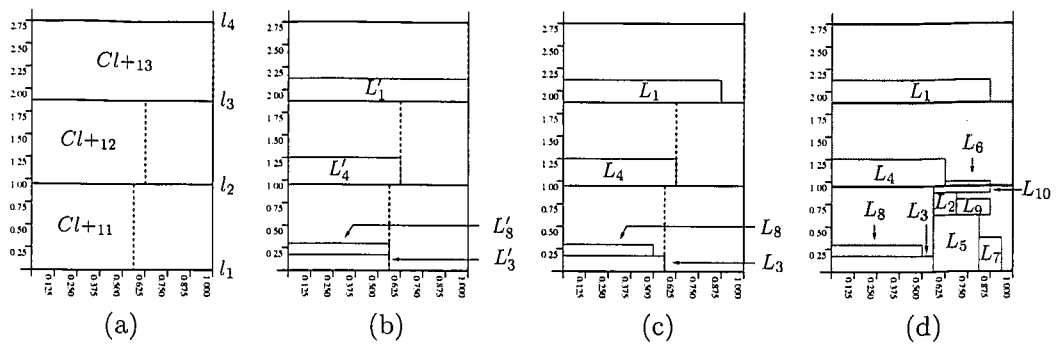

Figure 2.8: *From fractional strip packing to strip packing: (a) the strip is divided into horizontal levels, each with columns $Cl{+}_{uv}$ associated with columns Cl_{uv}, (b) the columns are filled up with rectangle groups of appropriate widths, (c) using the relation $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$, the groups are replaced by the corresponding rectangles from $\mathcal{L}_{wide}$ and (d) a strip packing of $\mathcal{L}_{general}$ results after insertion of rectangles from $\mathcal{L}_{narrow}$.*

$h + m \times h_{max} = 0.875 + 3 \times 0.625 = 2.75$, as depicted in Figure 2.8(c), where h is the height of the fractional strip packing, where h_{max} is the height of the tallest rectangle in the list $\mathcal{L}$ and where $m' = 3$ is the number of non-zero coordinates in the fractional strip packing LP-solution. □

Packing rectangles in $\mathcal{L}_{narrow}$ into the strip

Finally, the smaller rectangles in $\mathcal{L}_{narrow}$ may be inserted into the unoccupied rectangular areas $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_{m'}$. The narrow rectangles are first ordered according to non-increasing height and then inserted by means of the NFDH algorithm. If a new level cannot be created in $\mathcal{R}_i$, a new packing is resumed at the bottom left corner of region $\mathcal{R}_{i+1}$. When a rectangle cannot be packed in either $\mathcal{R}_1, \mathcal{R}_2, \ldots,$ or $\mathcal{R}_{m'}$, the remaining rectangles may be packed into the strip of width 1 starting above $\mathcal{R}_{m'}$ at level $l_{m'+1}$. A pseudocode listing of the Kenyon-Remila algorithm is given as Algorithm 10.

Example 2.14 *(Continuation of Examples 2.10, 2.11, 2.12 and 2.13)*

The rectangles in $\mathcal{L}_{narrow}$ are ordered according to non-increasing height, $\{L_5, L_7, L_2, L_9, L_{10}, L_6\}$. Applying the NFDH algorithm, these rectangles are then inserted into the regions $\mathcal{R}_1$ and $\mathcal{R}_2$. Because the dimensions of the rectangles were rescaled such that the width of the strip is equal to 1, the true height of the packing is given by $2.75 \times 16 = 44$ units. □

The Kenyon-Remila algorithm has running time[8]

$$O\left((n \log n)\epsilon^{-8}\text{polylog}(\epsilon) + (\log^2 n)\epsilon^{-16}\text{polylog}(\epsilon)\right) \tag{2.13}$$

for any fixed ϵ as $n \to \infty$, which is polynomial in both n and $1/\epsilon$. The running time is based on a result from [68, Theorem 1]. If, for example, $\epsilon = 0.1$, then the running time

[8] A polylogarithmic (polylog) function in ϵ is a polynomial in the logarithm of ϵ, that is $\text{polylog}(\epsilon) = \sum_{i=0}^{s} c_i \log^i \epsilon$, where s is the degree of the polynomial [120].

Algorithm 10 The Kenyon-Remila algorithm

Description: Pack a list $\mathcal{L}_{general}$ of rectangles into an open ended strip of fixed width and infinite potential height. The list of rectangles $\mathcal{L}_{general}$ is fully specified in advance, before packing commences.

Input: A list $\mathcal{L}_{general}$ of rectangles to be packed and a constant $0 < \epsilon < 1$.

Output: The height H of the packing obtained in the strip.

1: $\epsilon' = \epsilon/(2+\epsilon)$; $m \leftarrow \lceil (1/\epsilon')^2 \rceil$
2: Partition the list of rectangles $\mathcal{L}_{general}$ into two sub-lists $\mathcal{L}_{narrow}$ and $\mathcal{L}_{wide}$ to set aside rectangles of width less than ϵ'
3: Stack up all rectangles of $\mathcal{L}_{wide}$ left justified in order of non-increasing width such that; $w(L_1) \geq w(L_2) \geq \ldots \geq w(L_m)$
4: Form m groups of rectangles and round up the width in each group relative to the threshold rectangle below a particular group and a list $\mathcal{L}_{sup}$ such that $\mathcal{L}_{wide} \preceq \mathcal{L}_{sup}$ is formed
5: Solve fractional strip packing on $\mathcal{L}_{sup}$ with tolerance 1
6: From the fractional strip packing, construct an integral strip packing of $\mathcal{L}_{sup}$ which results in a well-structured strip packing of $\mathcal{L}_{wide}$
7: Sort $\mathcal{L}_{narrow}$ in order of decreasing height and add the rectangles of $\mathcal{L}_{narrow}$ to the strip packing of $\mathcal{L}_{wide}$ using the NFDH algorithm heuristic
8: Compute and output the height of the strip H

is $O(n \log n)$, similar to algorithms such as the FFDH and NFDH algorithms, since the polylog function does not depend on n.

The Kenyon-Remila algorithm is a theoretical rather than a practical algorithm. This is because the list $\mathcal{L}$ of rectangles to be packed has to be very large for the algorithm to yield good solutions due to the free spaces intentionally left by the algorithm in a packing and also because there is a large constant involved in the asymptotic worst-case running time (2.13) [104]. It is for these reasons that the Kenyon-Remila algorithm is not included in the computational analysis to be carried out in Chapter 5; only benchmark data sets with up to 500 rectangles are considered in Chapter 5 as mentioned in §1.6.2 and these instances are not large enough for the Kenyon-Remila algorithm to perform favourably.

2.2 Possible improvements to known heuristics

As mentioned in the introduction, several possible improvements are proposed in this section to the algorithms described in §2.1.

2.2.1 The NFDHIW algorithm

A newly proposed procedure dubbed the *next-fit decreasing height increasing width* (NFDHIW) algorithm, is obtained by modifying the NFDH algorithm (see §2.1.1.1) slightly—the only difference being that the pre-ordering of rectangles of equal height is additionally resolved by non-decreasing width. The total height of the packing achieved by this procedure is 20 units when applied to rectangles in Table 1.2, as shown in Figure 2.9(a).

2.2.2 The NFDHDW algorithm

The *next-fit decreasing height decreasing width* (NFDHDW) algorithm is another newly proposed procedure, which is also similar to the NFDH algorithm (see §2.1.1.1), except that the pre-ordering of rectangles of equal height is additionally resolved by non-increasing width. A total packing height of 20 units is achieved by this procedure when applied to rectangles in Table 1.2, as shown in Figure 2.9(b).

2.2.3 The FFDHIW algorithm

A slight variation on the FFDH algorithm (see §2.1.1.2), called the *first-fit decreasing height increasing width* (FFDHIW) algorithm is proposed, which is analogous to the FFDH algorithm—the only difference being that the pre-ordering of rectangles of equal height is additionally resolved by non-decreasing width. A total packing height of 19 units is obtained when applying this procedure to rectangles in Table 1.2, as illustrated in Figure 2.9(c).

2.2.4 The FFDHDW algorithm

The *first-fit decreasing height decreasing width* (FFDHDW) algorithm is another newly suggested modification to the FFDH algorithm (see §2.1.1.2), in which the pre-ordering of rectangles of equal height is additionally resolved by non-increasing width. When applied to the rectangles in Table 1.2, this procedure results in a total packing height of 19 units, as shown in Figure 2.9(d).

2.2.5 The BFDHIW algorithm

A proposed modification to the BFDH algorithm (see §2.1.1.3) is the *best-fit decreasing height increasing width* (BFDHIW) algorithm. The BFDHIW algorithm is similar to the BFDH algorithm, except that rectangles of equal height are additionally resolved by ordering them according to non-decreasing width. When the BFDHIW algorithm is applied to rectangles in Table 1.2, a total packing height of 19 units is achieved, as shown in Figure 2.9(c).

2.2.6 The BFDHDW algorithm

Another proposed modification to the BFDH algorithm (see §2.1.1.3) is the *best-fit decreasing height decreasing width* (BFDHDW) algorithm. This algorithm differs from the BFDH algorithm in that rectangles of equal height are additionally ordered according to non-increasing width. When the BFDHDW algorithm is applied to rectangles in Table 1.2, a total packing height of 19 units is obtained, as shown in Figure 2.9(d).

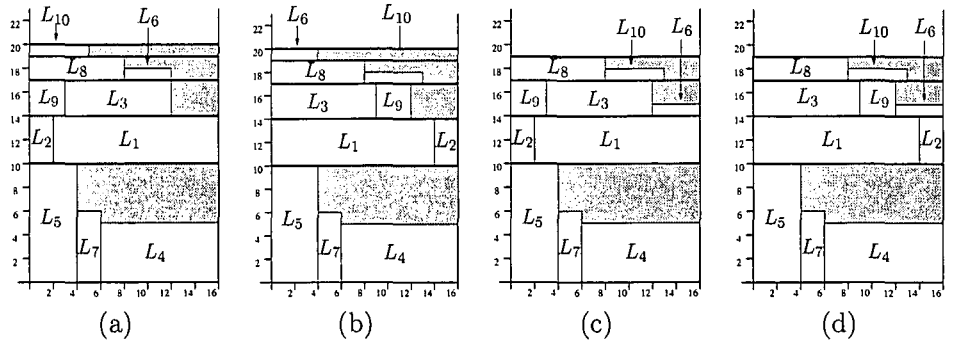

Figure 2.9: *Packings produced by the modified next fit, first fit and best fit classes of algorithms. (a) The NFDHIW algorithm, (b) the NFDHDW algorithm, (c) the FFDHIW and BFDHIW algorithms and (d) the FFDHDW and BFDHDW algorithms.*

2.2.7 The modified split fit algorithm

An investigation was carried out with respect to the possibility of improving the SF algorithm (see §2.1.1.5), considering the case where the region $\mathcal{R}$ is not fixed. In the proposed *modified split fit* (SF_{mod}) algorithm, when rectangle L_i does not fit into region $\mathcal{R}$, it is packed above the packing of $\mathcal{L}_1$. If L_i is the first rectangle packed on this level, then the height of region $\mathcal{R}$ is increased by $h(L_i)$. As new levels are created above the packing of $\mathcal{L}_1$, the height of region $\mathcal{R}$ also increases. This leaves more room for smaller rectangles to be packed in region $\mathcal{R}$.

Example 2.15 *When the* SF_{mod} *algorithm was applied to the rectangles shown in Table 2.1, a total packing height of 19 units was obtained, as shown in Figure 2.10(a). Because the height of region* $\mathcal{R}$ *is not fixed, rectangle* L_5 *is packed into region* $\mathcal{R}$*. The next rectangle,* L_7*, does not fit into* $\mathcal{R}$*, therefore it is packed above the rectangles in* $\mathcal{L}_1$*. The height of region* $\mathcal{R}$ *is increased by* $h(L_7) = 6$*. Rectangles* L_2 *and* L_9 *are packed into region* $\mathcal{R}$*, followed by rectangle* L_8*, above* $\mathcal{L}_1$*. A new level is created above the* $\mathcal{L}_1$ *packing and rectangle* L_{10} *is packed there. Since this is the first rectangle on the level, the height of region* $\mathcal{R}$ *is increased by* $h(L_{10}) = 1$*; this allows for rectangle* L_6 *to be packed into the new region* $\mathcal{R}$*.* ◘

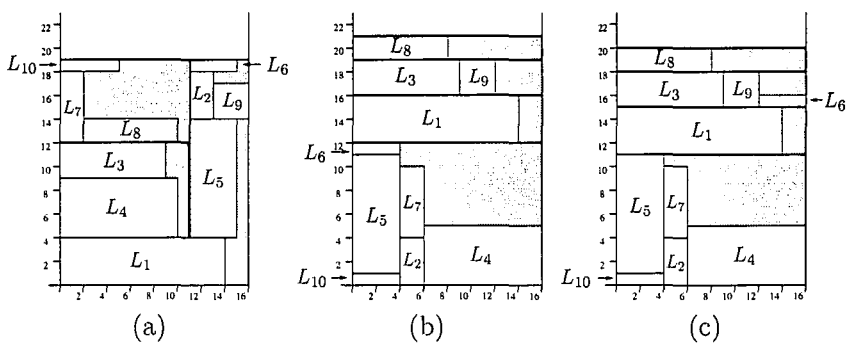

Figure 2.10: *Comparison of the packing produced by the* SF_{mod} *algorithm,* $\text{AlgorithmJOIN}_{mod}$*, and AlgorithmJOIN when applied to the rectangles in Table 2.1. (a) The* SF_{mod} *algorithm packing, (b) the* $\text{AlgorithmJOIN}_{mod}$ *packing and (c) the AlgorithmJOIN packing.*

2.2.8 The modified AlgorithmJOIN

As mentioned in §2.1.1.6, consecutive pairs of rectangles L_j and L_{j+1} whose heights $h(L_j)$ and $h(L_{j+1})$ differ by no more than a given γ percentage are conjoined in AlgorithmJOIN provided $w(L_j) + w(L_{j+1}) \leq W$. If the difference between the heights exceeds the given $\gamma\%$, the algorithm then moves on to the next pair L_{j+1} and L_{j+2}. If, on the other hand, the difference is at most $\gamma\%$, the algorithm moves on to L_{j+2} and L_{j+3}. The process continues until all the rectangles have been scanned. In the proposed *modified AlgorithmJOIN* (AlgorithmJOIN$_{mod}$) slight modifications were introduced in AlgorithmJOIN, not only to compare the heights of consecutive pairs of rectangles but also to compare the height of the "super" rectangle to the height of the subsequent rectangle. When a "super" rectangle is formed, its height is compared to that of the next rectangle and if the differences in height are at most $\gamma\%$, then this rectangle is appended to the "super" rectangle, provided that $w(L_j) + w(L_{j+1}) + w(L_{j+2}) \leq W$.

A variation of this algorithm may also be considered when the difference of widths $w(L_j)$ and $w(L_{j+1})$ does not exceed $\gamma\%$, in which case a "super" rectangle of height $h(L_j) + h(L_{j+1})$ and width $w(L_j)$ is formed. When a "super" rectangle is formed, its width is compared to that of the next rectangle and if the differences in width are at most $\gamma\%$, this rectangle is appended to the "super" rectangle of height $h(L_j) + h(L_{j+1}) + h(L_{j+2})$.

Example 2.16 *To illustrate the steps of* AlgorithmJOIN$_{mod}$ *slight adjustments are made to rectangle* L_{10} *in Table 1.2 by changing its width from 5 units to 4 units. The rectangles are then ordered according to non-increasing width as* $L_1, L_4, L_3, L_8, L_{10}, L_5, L_6, L_9, L_2, L_7$. *Suppose* $\gamma = 5\%$. *Then the list is searched for rectangles whose widths differ by no more than 5%. Two "super" rectangles* $\{L_{10}, L_5, L_6\}$ *and* $\{L_2, L_7\}$ *are formed with combined heights* 12 *and* 11 *respectively. Using the NFDH and FFDH algorithms, the rectangles are reordered according to non increasing height as* $L_{L_{10},6,5}, L_{2,7}, L_4, L_1, L_3, L_9, L_8$, *and as before the double subscript represent the super rectangle. A total height of 21 units is obtained by both the NFDH and FFDH algorithms, as shown in Figure 2.10(b). AlgorithmJOIN produces a total packing height of 20 units for this modified set of rectangles as shown in Figure 2.10(c).* □

2.2.9 The modified floor ceiling no rotation algorithm

As mentioned in §2.1.1.7, gaps are left on the ceiling in the FCNR algorithm (see §2.1.1.7) to allow for guillotine packing, as shown in Figure 2.2(b). The proposed *modified floor ceiling no rotation* (FCNR$_{mod}$) algorithm differs from the FCNR algorithm when searching for sufficient space to pack rectangles, in terms of the residual horizontal ceiling space. In the latter algorithm, when searching the ceiling for sufficient space, only the residual horizontal ceiling space is taken into consideration and the gaps are ignored, as shown in Figure 2.11(a). In the FCNR$_{mod}$ algorithm, on the other hand, the widths of the gaps along with the residual horizontal ceiling space, are taken into consideration when searching for sufficient space to pack a rectangle, as shown in Figure 2.11(b). The rectangle is packed on the ceiling with smallest packing space where it fits. Clearly, due to the additional computation of gap widths on each level, this modification will have an impact on the execution time of the FCNR$_{mod}$ algorithm. However, it is believed that the packing height may be reduced if more rectangles fit into the gaps, since this contributes towards

the delay in creating new levels. Because the example instance shown in Figure 2.2(c) produces a guillotineable packing with no gaps on the ceiling, the FCNR$_{mod}$ algorithm yields the same packing in this special case.

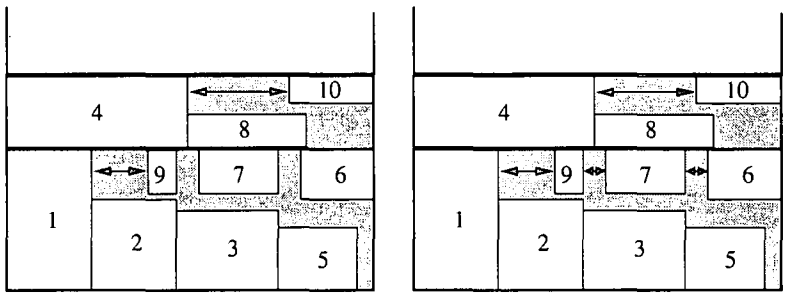

Figure 2.11: *Illustration of the FCNR$_{mod}$ algorithm. (a) Two residual horizontal ceiling spaces compared in the original FCNR algorithm and (b) the gap widths along with residual horizontal spaces are compared in the* FCNR$_{mod}$ *algorithm. The gap widths and residual horizontal spaces are indicated by double sided arrows.*

2.2.10 The modified Sleator algorithm

It is believed that the total packing height produced by the Sleator algorithm (see §2.1.2.1) may be reduced provided that it is possible to pack the tallest unpacked rectangle alongside one of the stacked rectangles. This is the motivation for the suggested *modified Sleator* (Sleator$_{mod}$) algorithm, where rectangles whose widths are greater than 1/2 are ordered according to non-increasing width and then stacked left justified from the bottom of the strip upwards. The unpacked rectangles are ordered according to non-increasing height and suppose the height of the tallest rectangle L_i (say), is denoted by H_{tall}. Starting with the bottom-most packed rectangle, the distance between the right-hand edges of the stacked rectangles and the right-hand boundary of the strip is computed until there is sufficient space or there is not enough room to pack the rectangle. If there is sufficient space, the lower edge of the stacked rectangle L_j (say) contributing to this space is referred to as the *lower boundary* (lb), as shown in Figure 2.12(a)–(b). The combined height of all rectangles below the lower boundary with insufficient horizontal space is denoted by H_{low}. Rectangle L_i is then packed right justified with its lower edge along the same level as the lower boundary, directly above H_{low} (see Figure 2.12(a)–(b), where rectangle L_4 is packed). If there is still some space left between the right-hand edge of L_j and the left-hand edge of L_i, subsequent rectangles may be packed there right justified.

After stacking rectangles of width greater than 1/2 in the Sleator$_{mod}$ algorithm, one of three cases arise:

Case 1: Rectangle L_i does not fit into any of the horizontal spaces between the right-hand edge of the stacked rectangles and right-hand edge of the strip. In this case the Sleator$_{mod}$ algorithm continues as described in §2.1.2.1, by packing the rectangles from the left to the right of the strip, and the strip is then divided into two equal segments.

Case 2: Rectangle L_i may be packed alongside one of the stacked rectangles and $H_{\text{tall}} + H_{\text{low}} \leq H_{\text{stack}}$, as shown in Figure 2.12(a). If there is still some space left between the

right-hand edge of L_j and the left-hand edge of L_i, subsequent rectangles are packed there right justified. If there is insufficient space, the unpacked rectangles are packed above H_{stack} from the left-hand to the right-hand edges of the strip, as described in §2.1.2.1.

Case 3: Rectangle L_i may be packed along side one of the stacked rectangles and $H_{\text{tall}} + H_{\text{low}} > H_{\text{stack}}$, as shown in Figure 2.12(b). The remaining unpacked rectangles are packed above H_{stack} from the left-hand edge of the strip to the left-hand edge of the first rectangle whose height combined with H_{low} is greater than H_{stack}. If there are still some unpacked rectangles, then the strip is partitioned into two equal segments and the height of the right segment is initially given by $H_{\text{tall}} + H_{\text{low}}$. The packing procedure then continues as described in §2.1.2.1.

Example 2.17 *When the* Sleator$_{mod}$ *algorithm is applied to the rectangles in Table 1.2, the packing shown in Figure 2.12(c) results, with a total packing height of 17 units. Rectangles L_1, L_4 and L_3 are stacked from the bottom of the strip upwards. The lower boundary is given by rectangle L_4, because the distance between the right-hand edge of rectangle L_4 and the right-hand edge of the strip is 6 units, which is sufficient to pack the tallest unpacked rectangle L_5 since it has width of 4 units. Rectangle L_5 is packed right justified and the remaining 2 units are sufficient to place the next unpacked rectangle, L_7. This is case 3 above. Therefore the remaining unpacked rectangles are packed above rectangle L_3 from the left-hand edge of the strip to the left-hand edge of rectangle L_5 since $h(L_7)+H_{\text{low}} < H_{\text{stack}}$. Only rectangles L_2 and L_9 fit and the strip is divided into two equal segments, with the left segment of width $H_{\text{left}} = 16$ and $H_{\text{right}} = 14$. Packing resumes in the right segment, because it has the smallest height and only rectangle L_8 fits. Rectangles L_{10} and L_6 are finally packed in the left and right segments respectively.* □

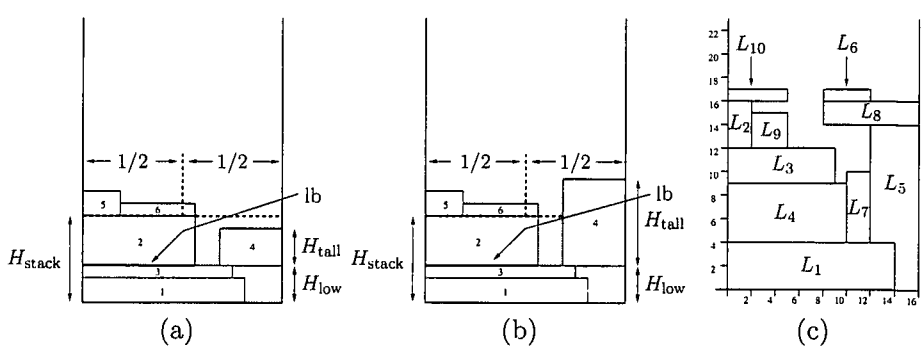

Figure 2.12: *Packings produced by the modified Sleator algorithm. (a) Illustration of case 2, (b) illustration of case 3 and (c) the packing produced when the* Sleator$_{mod}$ *algorithm is applied to the rectangles in Table 1.2.*

2.2.11 The Kenyon-Remila Insertion algorithm

In the original strip packing procedure of the Kenyon-Remila algorithm (see §2.1.3), the layer between levels l_j and l_{j+1} has a well-structured space that may be partitioned into three rectangular regions; (1) region $\mathcal{R}_1$ has width $w(C_j)$ and height x_j, and is fully packed with wide rectangles, (2) region $\mathcal{R}_2$ is completely empty, and has width $w(l_j) - w(C_j)$ and height $x_j + h_{max}$, and (3) region $\mathcal{R}_3$ is partially filled with rectangles overlapping from

region $\mathcal{R}_1$, whose free space is considered as waste and whose height is $l_{j+1} - x_j$. In the newly proposed *Kenyon-Remila insertion* (KeInsert) algorithm, it is suggested that the unused part of region $\mathcal{R}_3$ should not be regarded as waste — instead this region should be divided further into two rectangular regions, $\mathcal{R}_{31}$ and $\mathcal{R}_{32}$. Region $\mathcal{R}_{31}$ is similar to the original region $\mathcal{R}_3$; the only difference being that the height ranges from x_j to the height of the tallest rectangle overlapping from region $\mathcal{R}_1$. Region $\mathcal{R}_{32}$, on the other hand, is originally left completely empty, with its height ranging from the top of region $\mathcal{R}_{31}$ to l_{j+1}. Region $\mathcal{R}_{32}$ is then used as a second insertion region for narrow rectangles, with region $\mathcal{R}_2$ being the first insertion region. The various divisions of the layers are illustrated in Figure 2.13. A total packing height of $2.75 \times 16 = 44$ units was obtained, as shown in Figure 2.14(a), when the KeInsert algorithm was applied to the rectangles in Table 1.2.

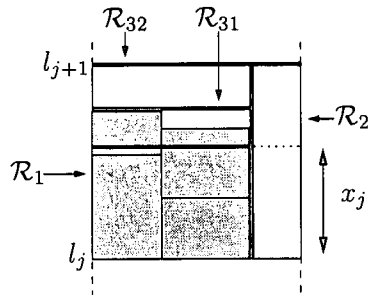

Figure 2.13: *Division of a level into four regions.*

2.2.12 The Kenyon-Remila max height algorithm

It was realised that the addition of h_{max} to each level in the original strip packing sometimes creates unnecessarily large unused spaces towards the top of the layer. The proposed "so-called" *Kenyon-Remila max height* (KeMaxHeight) algorithm avoids adding h_{max} to each x_j in the original strip packing. At each level the height of the tallest rectangle per configuration is determined and it is this height that is instead added to each x_j when computing the height of a level. This means that at each level different heights of rectangles will be added to x_j corresponding to the configuration used. It is believed that this modification will create levels with shorter heights thereby reducing the overall height of the original strip packing, thus leading to an improvement of the solution quality.

Example 2.18 *When the KeMaxHeight algorithm was applied to the rectangles in Table 1.2, a total packing height of* $(1.625 \times 16) + 3 = 29$ *units was obtained, as shown in Figure 2.14(b). All steps are similar to the original Kenyon-Remila algorith, except when constructing the full strip packing. In this case each configuration is taken in turn and the strip is filled from the bottom upwards. Recall, from Example 2.12, that* $\underline{x} = (0.3125, 0.3125, 0.25)$. *Configuration* C_1 *is packed between* $l_1 = 0$ and $l_2 = l_1 + x_1 + 0.1875 = 0.5$. *Next, configuration* C_2 *is packed between* $l_2 = 0.5$ and $l_3 = l_2 + x_2 + 0.3125 = 1.125$. *Finally, configuration* C_3 *is packed between* $l_3 = 1.125$ and $l_4 = l_3 + x_3 + 0.25 = 1.625$. *The narrow rectangles* L_5, L_7, L_2 *are inserted into the empty regions, while the remaining rectangles* L_9, L_{10}, L_6 *are packed on a new level* l_5. ∎

2.2.13 The Kenyon-Remila max height insertion algorithm

The *Kenyon-Remila max height insertion* (KeMaxInsert) algorithm is another newly proposed combination of the two modifications described in §2.2.11 and §2.2.12. Even though the *KeMaxHeight* algorithm creates shorter levels, it may still be possible to insert some narrow rectangles in region $\mathcal{R}_{32}$, and as such this possibility is explored. When applied to the rectangles in Table 1.2, a total packing height of $1.625 \times 16 = 26$ units was obtained, as shown in Figure 2.14(c).

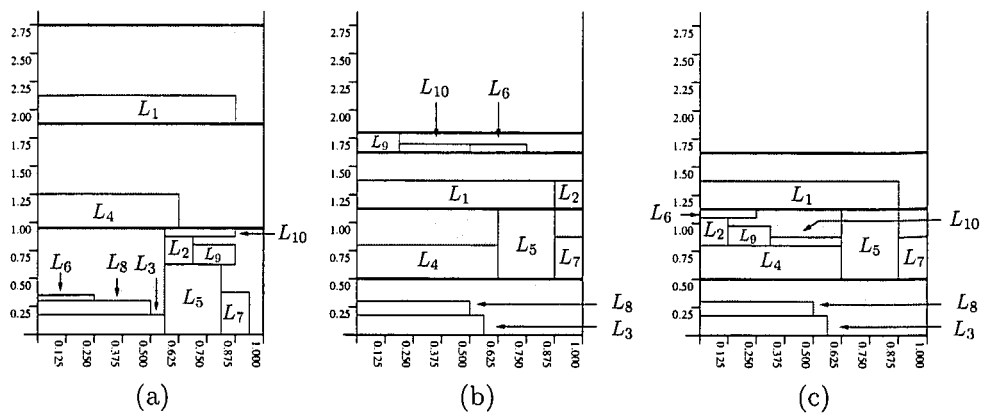

Figure 2.14: *Packings produced by the modified Kenyon-Remila algorithms, when applied to rectangles in Table 1.2. (a) the KeInsert packing, (b) the KeMaxHeight packing, and (c) the KeMaxInsert packing.*

2.2.14 The Kenyon-Remila algorithmJOIN

In the *Kenyon-Remila algorithmJOIN*, it is proposed that instead of using threshold rectangles to group rectangles together, AlgorithmJOIN$_{mod}$ should rather be used. This suggestion is due to the fact that the *Kenyon-Remila* algorithm aims to have rectangles of distinct widths and the modified algorithmJOIN is able to achieve this. The Kenyon-Remila algorithm described in §2.1.3 along with previously described modifications in §2.2.11–§2.2.13 respectively called KenyonJOIN, KeInsertJOIN, KeMaxHeightJOIN and KeMaxInsertJOIN, were reimplemented using the new idea of grouping, while all other steps remained exactly the same.

Example 2.19 *Using* AlgorithmJOIN$_{mod}$*, four groups are formed:* $g_1 = L_8, g_1 = L_3, g_3 = L_4$ *and* $g_4 = L_1$*, when* $\gamma = 5\%$*. Total packing heights of* $53, 53, 30$ *and* 27 *units were obtained by the KenyonJOIN algorithm, the KeInsertJOIN algorithm, the KeMaxHeightJOIN algorithm and the KeMaxInsertJOIN algorithm respectively, as shown in Figures 2.15(a)–(d).* □

2.3 A new offline strip packing heuristic

It was observed that when the difference in heights of rectangles fitting into one level (see Figure 2.16(a)) becomes extreme, the FFDH algorithm performs badly in relation

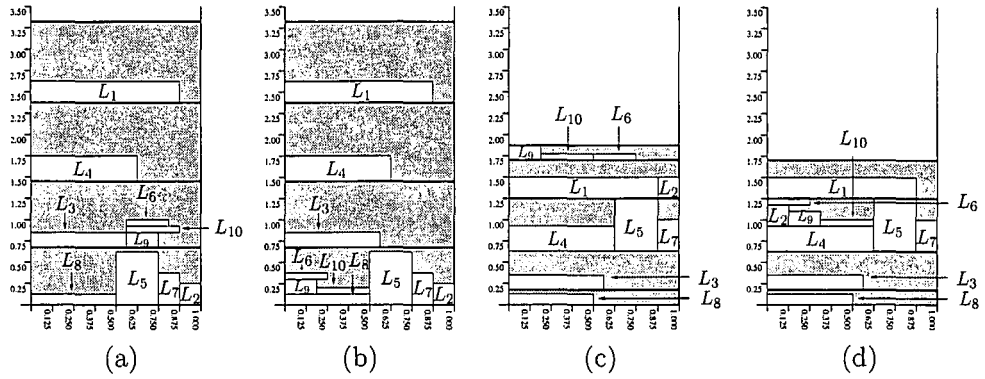

Figure 2.15: *Comparison of packings produced using the new idea of grouping in the Kenyon-Remila algorithms, when applied to rectangles in Table 1.2. (a) the KenyonJOIN algorithm, (b) the KeInsertJOIN algorithm, (c) the KeMaxHeightJOIN algorithm and (d) the KeMaxInsertJOIN algorithm.*

to the optimal solution. This observation provided motivation for the development of a new algorithm, called the *Size Alternating Stack* (SAS) algorithm, which is applicable to problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

In this algorithm, the list $\mathcal{L}$ of n rectangles is partitioned into two sublists $\mathcal{L}_1$ and $\mathcal{L}_2$ consisting of rectangles satisfying $h(L_i) > w(L_i)$ and $h(L_j) \leq w(L_j)$ respectively. The n_1 rectangles in $\mathcal{L}_1$ are called *narrow* rectangles and the n_2 rectangles in $\mathcal{L}_2$ are referred to as *wide* rectangles ($n = n_1 + n_2$). Rectangles in the list $\mathcal{L}_1$ are ordered according to non-increasing height, while rectangles in list $\mathcal{L}_2$ are ordered according to non-increasing width. Each level of the packing is initialised by comparing the heights of the first rectangle in both lists (*i.e.* tallest rectangle in list $\mathcal{L}_1$ and widest rectangle in $\mathcal{L}_2$)—packing the rectangle of largest height. The height of this rectangle becomes the height of the level and a horizontal line is drawn coinciding with the top edge of the rectangle to the right-hand edge of the strip to demarcate the upper boundary of the level. A level may be initialised by more than one rectangle provided all (say k) rectangles have heights equal to the largest height and there is sufficient horizontal space.

The main idea in this algorithm is to alternate between the *narrow* and *wide* rectangles while packing from the left to the right on each level of the strip (*i.e.* if the rectangle initialising a level is from $\mathcal{L}_1$, alternate to a rectangle in $\mathcal{L}_2$ and *vice-versa*). Once the list from which to pack has been identified, the rectangles in that particular list are stacked on top of each other starting from the lower boundary of a level until the upper boundary is reached, or until the vertical space between the upper boundary of the level and the top edge of the top-most rectangle in the stack is insufficient to fit in any more of the unpacked rectangles in that list.

If the widths of subsequent rectangles L_i and L_{i+1} are not equal when stacking the wide rectangles, an empty rectangular region $\mathcal{R}_j$, whose left-hand boundary is the right-hand edge of rectangle L_{i+1} remains, as shown in Figure 2.16(b). The height of each rectangular region extends from the top edge of rectangle L_i to the upper boundary of a level, while the width of each region is given by $w(\mathcal{R}) = w(L_i) - w(L_{i+1})$. These rectangular regions are used to pack the *narrow* rectangles. The *narrow* rectangles are stacked by selecting all the rectangles whose widths do not exceed the width of the bottom-most narrow

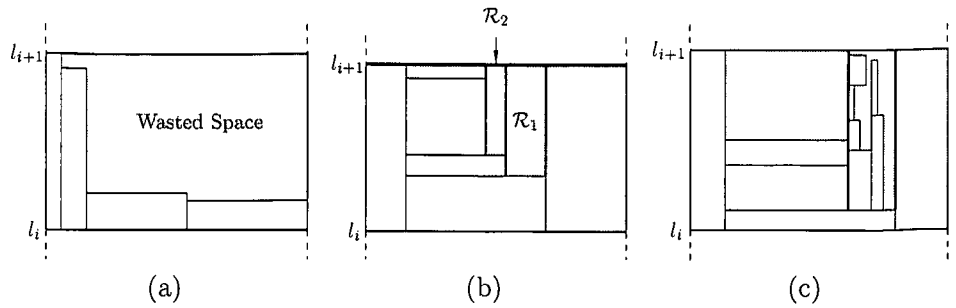

Figure 2.16: *(a) Motivation for developing the SAS algorithm, (b) empty rectangular regions left when stacking* wide *rectangles by means of the SAS algorithm and (c) stacking of narrow rectangle in region* $\mathcal{R}_j$.

rectangle, but also fit height-wise within a level. The stacking of narrow rectangles in $\mathcal{R}_j$ is shown in Figure 2.16(c) where the rectangles are stacked to fill the width of $\mathcal{R}_j$, until there is insufficient horizontal space or there are no more *narrow* rectangles to pack.

The algorithm is flexible—if, through interchange of lists, there is insufficient horizontal space on a level to pack any of the rectangles in the designated list, then the rectangles in the alternative list may be packed provided that there is sufficient space. A new level is initialised if none of the rectangles in either $\mathcal{L}_1$ or $\mathcal{L}_2$ fit into the horizontal space between the right-hand boundary of the strip and the right-hand edge of the right-most rectangle packed.

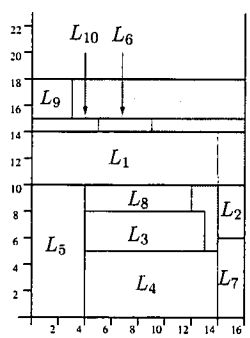

Figure 2.17: *Packing produced by the SAS algorithm, when applied to the rectangles in Table 1.2.*

Note that the SAS algorithm produces a guillotineable packing. The algorithm is given in pseudocode form as Algorithm 11.

Example 2.20 *When the SAS algorithm was applied to the rectangles in Table 1.2, a total packing height of 18 units is obtained, as shown in Figure 2.17. Partitioning the list into two sublists* $\mathcal{L}_1 = L_5, L_7, L_2$ *and* $\mathcal{L}_2 = L_1, L_4, L_3, L_8, L_{10}, L_6, L_9$ *of narrow and wide rectangles respectively. Comparing the heights of the first rectangle in either list,* L_5 *is selected, because it has the greatest height. Therefore it initialises the first level. Alternating to* $\mathcal{L}_2$*, rectangles* L_4, L_3, L_8 *are stacked on top of each other until there is insufficient space. None of the narrow rectangles may be inserted into the regions created by the stacked rectangles. Alternating to* $\mathcal{L}_1$*, rectangles* L_7, L_2 *are stacked and the first*

level is full. Therefore a new level is created above the first level. Only wide rectangles remain and they are packed on levels with enough room or on newly created levels. ∎

The procedure *pack narrow* in Algorithm 11 has a time complexity of $O(n)$, because the entire list of narrow rectangles is searched to find suitable rectangles to pack, while the procedure *pack wide* has a time complexity of $O(n^2)$, because for every wide rectangle stacked, narrow rectangles are stacked. The partitioning of the list in line 1 has a running time of $O(n)$ when using a linear search technique and the sorting of the lists in line 2 has an execution time of $O(n \log n)$, when using an efficient sorting procedure such as the merge-sort algorithm. The while-loop spanning lines 3–11 has a time complexity of $O(n)$. The overall worst-case time complexity of the SAS algorithm is therefore $O(n^2)$.

2.4 Chapter summary

This chapter opened with a description of known heuristics from the literature for solving the *offline* strip packing problem (§2.1), ranging from the classical heuristics dating back to 1983 to more recent heuristics, developed as late as 2004. The algorithms employ different approaches, such as partitioning the strip into horizontal levels (§2.1.1) or just packing rectangles in the plane (§2.1.2). To improve the performance in terms of obtaining the smallest packing height, several modifications to some of the known heuristics were suggested in §2.2. Finally, a new heuristic was described in §2.3 which also partitions the strip into horizontal levels. However, different techniques are employed to fully utilise space within each level. All algorithms presented in this chapter were applied to the same example strip packing instance in order to illustrate the various packing patterns produced by the algorithms.

Algorithm 11 The size alternating stack (SAS) algorithm

Description: Packing a list of rectangles into a strip of fixed width and infinite height. The list of rectangles is fully specified in advance, before packing commences.

Input: The number of rectangles to be packed n, the dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height H of the packing obtained in the strip.

1: Partition the list of rectangles $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ such that $\mathcal{L}_1$ is a list with $h(L_i) > w(L_i)$ for all $1 \leq i \leq n_1$, while $\mathcal{L}_2$ is a list with $w(L_j) \geq h(L_j)$ for all $1 \leq j \leq n_2$.
2: Order $\mathcal{L}_1$ according to non-increasing height and order $\mathcal{L}_2$ according to non-increasing width. $j \leftarrow 1$, $i \leftarrow 1$, level $\leftarrow 1$
3: **while** $n_1 \neq 0$ or $n_2 \neq 0$ **do**
4: compare $h(L_i)$ with $h(L_j)$ and select the rectangle with greatest height. Pack the selected rectangle on the level
5: **if** tallest rectangle is narrow **then**
6: $h(\text{level}+1) \leftarrow h(\text{level}) + h(L_i)$; **call** PackWide($w$(packedlevel), VerticalSpace)
7: **else** [tallest rectangle is wide]
8: $h(\text{level}+1) \leftarrow h(\text{level}) + h(L_j)$; **call** PackNarrow($w$(packedlevel), VerticalSpace)
9: **end if**
10: level $\leftarrow$ level $+ 1$, $j \leftarrow 1$, $i \leftarrow 1$
11: **end while**

Procedure PackNarrow(w(packedlevel), VerticalSpace)

1: pack first narrow rectangle that fits height-wise and width-wise
2: **while** there is sufficient vertical and horizontal space **do**
3: search $\mathcal{L}_1$ for rectangle whose width is at most the width of the bottom-most narrow rectangle
4: **if** such a rectangle exists **then**
5: stack the rectangle; remove it from $\mathcal{L}_1$
6: **end if**
7: **end while**

Procedure PackWide(w(packedlevel), VerticalSpace)

1: **while** there is sufficient vertical space and $j \leq n_2$ **do**
2: **if** rectangle fits height-wise **then**
3: stack rectangle L_j; remove it from $\mathcal{L}_2$
4: **if** rectangles of unequal widths are stacked **then**
5: region $\mathcal{R}$ is created and narrow rectangles are packed in this region
6: **call** PackNarrow (w(packed$\mathcal{R}$), $h(\mathcal{R})$)
7: **end if**
8: **end if**
9: **end while**

Chapter 3

Heuristics for Online problems

A number of known and new heuristics for *offline* strip packing were described and illustrated in Chapter 2. In this chapter heuristics for *online* packing problems are considered. Thirteen known heuristics from the literature are described in §3.1 and this is followed by six suggested modifications to some of the known heuristics in §3.2. The chapter closes with a description of four entirely new heuristics in §3.3.

3.1 Known Heuristics

As mentioned in §1.2, *online* problems refer to problems where the entire list of rectangles to be packed is not available in advance and as such one rectangle is packed at a time. The next rectangle to be packed becomes available immediately after the current rectangle has been packed, and once a rectangle is packed it may not be moved. Heuristics from the literature for solving problems of this nature are described in this section.

3.1.1 Level algorithms

Level algorithms for *online* problems partition the strip into horizontal levels of height equal to the height of the tallest rectangle packed on any particular level, similar to the level algorithms described in §2.1.1. The algorithms considered in this section are a slight variation to the NFDH, FFDH and BFDH algorithms. However, the pre-ordering condition is not considered since packing takes place in an *online* environment.

3.1.1.1 The next fit level algorithm

The *next fit level* (NFL) algorithm [26] was developed in 1990 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

In this algorithm, rectangles are packed (one at a time and in the order given) on the current level, left justified. The first level corresponds with the bottom of the strip. If there is insufficient horizontal space on the current level to pack the next rectangle, a horizontal line is drawn across the upper edge of the tallest rectangle on the current level

so as to create a new level above the current level. All levels below the current level are never revisited. Coffman [29] performed an average-case performance analysis of the NFL algorithm and found the expected height[1] of the packing to be

$$E[\text{NFL}(\mathcal{L})] \approx (0.3813384945\ldots)n\,, \tag{3.1}$$

asymptotically as $n \to \infty$, where n is the number of rectangles in the list $\mathcal{L}$, where $\text{NFL}(\mathcal{L})$ denotes packing height achieved for the list of rectangles $\mathcal{L}$ by the NFL heuristic, and where $E[\cdot]$ denotes the expected value operator. The algorithm is given in pseudocode form as Algorithm 12.

Example 3.1 *A total packing height of 26 units is obtained by the NFL algorithm when applied to rectangles in Table 1.2, as illustrated in Figure 3.1(a). Packing rectangles in the order given, rectangles L_1 and L_2 are packed on the first level. A new level is created where rectangle L_3 is packed, because there is insufficient space on the first level. Rectangles L_4 and L_5 are packed on the third level whose height is $h(L_5)$, because L_5 is the tallest rectangle on the this level. The packing process is continued in this manner until all rectangles are packed.* □

Algorithm 12 The next fit level (NFL) algorithm

Description: Packing a given list of rectangles into a strip of fixed width and infinite height, without pre-ordering the list.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i)\rangle$ and the strip width W.
Output: The height H and the entire packing.

```
1:  level ← 0; h(level + 1) ← 0; H ← 0; i ← 0
2:  while there is an unpacked rectangle do
3:    i ← i + 1
4:    if there is sufficient space on current level then
5:      pack rectangle left justified
6:      if h(level + 1) < h(L_i) then
7:        h(level + 1) ← h(L_i)
8:      end if
9:    else [there is insufficient space on current level]
10:     open a new level and pack the rectangle
11:     H ← H + h(level + 1); level ← level + 1
12:   end if
13: end while
14: print H and entire packing
```

The NFL algorithm has a worst-case time complexity of $O(n)$ [26]. The for-loop spanning lines 2–12 is executed exactly n times, while all other steps have constant time complexity.

3.1.1.2 The first fit level algorithm

In the *first fit level* (FFL) algorithm [26], also developed in 1990 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

[1] In this probabilistic model, it is assumed that the $2n$ random variables $w(L_1), \ldots, w(L_n), h(L_1), \ldots, h(L_n)$, are independent and drawn from the uniform distribution on [0,1] [29].

rectangles are packed (one by one in the order given) on the *lowest* level into which they fit both height-wise and width-wise; if a rectangle does not fit into any existing level, then a new level is created exactly as in the NFL algorithm and the rectangle in question is packed on that level. The difference between the NFL and FFL algorithms is that in the latter procedure levels lower than the highest level may be revisited, whereas this is disallowed in the former procedure. To the best knowledge of the author the asymptotic performance or average-case analysis of the FFL algorithm has not been analysed. The algorithm is given in pseudocode form as Algorithm 13.

Example 3.2 *A total packing height of 19 units is obtained by the FFL algorithm when applied to rectangles in Table 1.2, as shown in Figure 3.1(b). Rectangles L_1, L_2, L_3, L_4, L_5 are packed in the same manner as in NFL algorithm. However, rectangle L_6 is packed on the lowest level with sufficient space, which is level 2.* ∎

Algorithm 13 The first fit level (FFL) algorithm

Description: Packing a given list of rectangles into a strip of fixed width and infinite height, without pre-ordering the list.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$ and the strip width W.
Output: The height H and the entire packing.

1: level $\leftarrow 0$; $h(\text{level}+1) \leftarrow h(L_1)$; $H \leftarrow h(L_1)$; $i \leftarrow 1$
2: **while** there is an unpacked rectangle **do**
3: $i \leftarrow i+1$; level $\leftarrow 0$
4: search for the lowest level with sufficient space
5: **if** such a level exists **then**
6: pack rectangle left justified
7: **if** this is the top-most level and $h(\text{level}) < h(L_i)$ **then**
8: $h(\text{level}) \leftarrow h(L_i)$
9: **end if**
10: **else** [there is insufficient space in all existing levels]
11: create a new level above the top-most level
12: pack rectangle left justified
13: $h(\text{level}) \leftarrow h(L_i)$; $H \leftarrow H + h(L_i)$
14: **end if**
15: **end while**
16: **print** H and entire packing

The FFL algorithm has an average-case time complexity of $\Omega(\log n)$, where Ω denotes the asymptotic lower bound (described in Appendix A) [26]. It is not linear because it requires the entire packing to be active (*i.e.* previously packed shelves may be revisited) at every step of the packing.

3.1.1.3 The best fit level algorithm

The *best fit level* (BFL) algorithm [26] was also developed in 1990 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

It is similar to the FFL algorithm, except that each rectangle is placed on the lowest level (into which it fits both height-wise and width-wise) with minimum residual horizontal space. The residual horizontal space of a level is the space between the right-most edge of the last rectangle packed on the level and the right-hand boundary of the strip. If none of the existing levels have enough room, a new level is created above the current highest level. The average case analysis or the asymptotic performance of the BFL algorithm was also not analysed. The algorithm is given in pseudocode form as Algorithm 14.

Example 3.3 *The BFL algorithm obtains a total packing height of 19 units when applied to the rectangles in Table 1.2, as depicted in Figure 3.1(b). The packing produced is the same as that produced by the FFL algorithm for this special case.* □

Algorithm 14 The best fit level (BFL) algorithm

Description: Packing a given list of rectangles into a strip of fixed width and infinite height, without pre-ordering the list.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$ and the strip width W.
Output: The height H and the entire packing.

1: level $\leftarrow 0$; $h(\text{level}+1) \leftarrow h(L_1)$; $H \leftarrow h(L_1)$; $i \leftarrow 0$
2: **while** there is an unpacked rectangle **do**
3: $i \leftarrow i+1$; level $\leftarrow 0$
4: search for lowest level with minimum residual horizontal space
5: **if** such a level exists **then**
6: pack rectangle left justified
7: **else** [such a level does not exist]
8: create a new level above the top-most level
9: pack rectangle left justified
10: $h(\text{level}) \leftarrow h(L_i)$; $H \leftarrow H + h(L_i)$
11: **end if**
12: **end while**
13: **print** H and entire packing

Similarly to the FFL algorithm, the BFL algorithm has an average-case time complexity of $\Omega(\log n)$, because the entire packing is available (*i.e.* previously packed rectangles may be revisited) at every step of the packing process [26].

3.1.1.4 The bi-level next fit algorithm

The *bi-level next fit* (BiNFL) algorithm [29] was developed in 2002 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

It is a modification of the NFL algorithm described in §3.1.1.1. As the name suggests, the algorithm packs two levels at a time, referred to as the *lower* and *upper* levels. The height of the *lower* level is determined by the height of the tallest rectangle packed on it.

On the *lower* level, the first rectangle L_i to be packed is placed left justified. If the next rectangle L_{i+1} to be packed fits on the *lower* level, it is placed there, right justified. All other rectangles that follow and fit on the *lower* level are placed right justified, next to

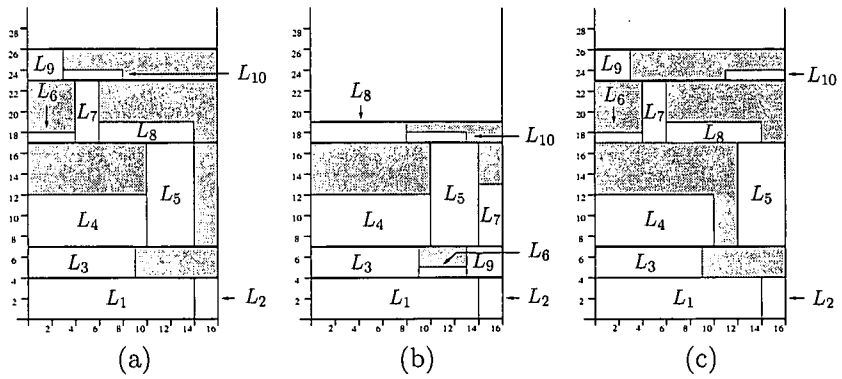

Figure 3.1: *Comparison of packings produced by different algorithms when applied to the list of rectangles in Table 1.2: (a) the NFL algorithm packing, (b) the FFL and BFL algorithm packings and (c) the BiNFL algorithm packing.*

the previous rectangle packed. If there is not enough room for a rectangle to be packed on the *lower* level, packing proceeds on the *upper* level. A horizontal line is drawn along the top edge of the tallest rectangle on the *lower* level and this becomes the lower boundary of the *upper* level.

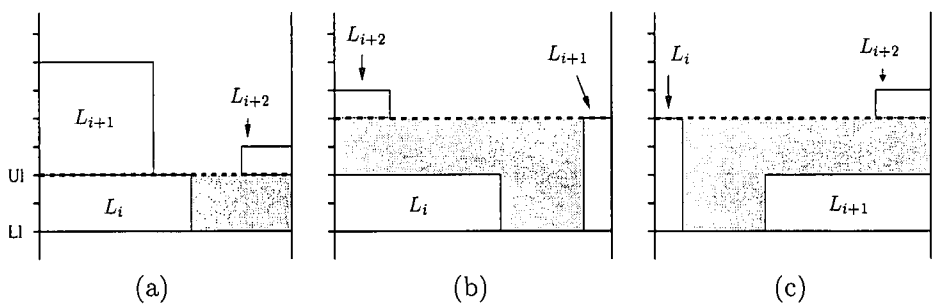

Figure 3.2: *Examples of patterns resulting from a BiNFL packing. In all instances the third rectangle, L_{i+2}, may be shifted down onto the* lower *level. Ul and Ll represent the lower boundaries of the* upper *and* lower *levels respectively.*

On the *upper* level, if rectangle L_{i+1} is the first rectangle to be packed (because it failed to fit on the *lower* level), it is packed left justified on top of L_i since it is the only rectangle on the *lower* level. Subsequent rectangles are packed right justified on this level provided there is sufficient space (see Figure 3.2(a)). If, on the other hand, L_{i+2} is the first rectangle to be packed on the *upper* level, it is packed above the shorter of L_i and L_{i+1} (because these are the only two rectangles on the *lower* level), justified against the same strip boundary as the shorter of rectangles L_i or L_{i+1}, as depicted in Figures 3.2(b) and (c). If a rectangle does not fit on the *upper* level, a new bi-level is created above the top level. Once a new bi-level is created, previous bi-levels are never revisited. The packing process proceeds as discussed for the *lower* and *upper* levels until all rectangles are packed.

At equilibrium[2], the expected height of the BiNFL algorithm is twice the expected height

[2]This occurs when the distributions of the widths of rectangles packed on any particular level, converge at a

of the NFL packing (3.1) [29]. The algorithm is given in pseudocode form as Algorithm 15.

Example 3.4 *A total packing height of 26 units was obtained by the Bi-NFL algorithm for our example instance in Table 1.2, as shown in Figure 3.1(c). Rectangles L_1 and L_2 are packed on the lower level, while rectangle L_3 is packed on the upper level. Rectangle L_4 does not fit on the upper level; hence a new bi-level is created and rectangle L_4 is packed left justified on the lower level. Rectangle L_5 is right justified on this level and rectangle L_6 is packed on the upper level, justified against the left-hand boundary of the strip, because rectangle L_4 is also justified against this strip wall and it is shorter than rectangle L_5.* □

Algorithm 15 The bi-level next fit (BiNFL) Algorithm

Description: Packing a given list of rectangles into a strip of fixed width and infinite height, without pre-ordering the list.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$ and the strip width W.
Output: The height H and the entire packing.

```
1: level ← 1; h(level) ← 0; H ← 0; i ← 1
2: open a new bi-level
3: call pack on lower level
4: print H and entire packing
```

Procedure pack on lower level

```
1: the first rectangle packed is left justified
2: subsequent rectangles that fit are right justified
3: if there is insufficient space then
4:    h(level) is given by the height of tallest rectangle packed
5:    call pack on upper level
6: end if
```

Procedure pack on the upper level

```
1: if L_{i+1} is the first rectangle packed then
2:    left justify on top of L_i
3: else
4:    if L_{i+2} is the first rectangle packed then
5:       pack above min {h(L_i), h(L_{i+1}}
6:       justify according to min {h(L_i), h(L_{i+1}}
7:    else
8:       if L_{i+j} (j > 3) fits on the upper level then
9:          left justify all other rectangles
10:      else [rectangle does not fit]
11:         H ← H + h(lower) + h(upper); open new bi-level
12:      end if
13:   end if
14: end if
```

geometric rate to an equilibrium distribution function [29].

The two procedures *pack on lower level* and *pack on upper level* in Algorithm 15, each has a time complexity of $O(n)$ because each of the n rectangles are packed in the bi-levels. Since these are the two main procedures, the worst-case time complexity of the BiNFL algorithm is therefore $O(n)$.

3.1.2 Shelf algorithms

A study of polynomial time approximation algorithms, called shelf algorithms, in which rectangles are packed in the order specified by means of a given list (queue) without sorting them first was carried out by Baker and Schwarz [6]. Shelf algorithms are primarily used to solve *on-line* packing problems. The first two algorithms described in this section, namely the Next Fit Shelf (NFS) and First Fit Shelf (FFS) algorithms dating from 1983, are modifications of the NFDH and FFDH algorithms[3] without the condition of first sorting the rectangles in order of non-increasing height. About seven years later the Best Fit Shelf (BFS) algorithm was introduced [26]. This algorithm is described next and is a modification of the BFDH algorithm (see §2.1.1.3). In these three algorithms shelf heights are not determined by the height of the first rectangles packed on the shelves (because such rectangles are not necessarily the tallest); additional space (called *free space*) is therefore left above the first rectangle on each shelf, so as to accommodate, to some extent, taller rectangles that may follow. The NFS, FFS and BFS algorithms employ a parameter, $0 < r < 1$, which is a measure of how much free space is allowed in each shelf. These algorithms are consequently referred to as Next Fit Shelf (NFS_r), First Fit Shelf (FFS_r) and Best Fit Shelf (BFS_r) algorithms.

3.1.2.1 The next fit shelf algorithm

The *next fit shelf* (NFS_r) algorithm [6] with parameter $0 < r < 1$ was developed in 1983 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

It is a natural modification of the NFDH algorithm (see §2.1.1.1), the difference being that the rectangles are not sorted in the NFS_r algorithm; they are merely packed in the order given. In this algorithm, a value of r is initially selected for an entire packing. Before packing each rectangle, the smallest value $k \in \mathbb{Z}$ is computed such that $r^{k+1} < h(L_i) \leq r^k$; here r^k is referred to as the *appropriate height* of the shelf to pack rectangle L_i. A rectangle is packed on the highest shelf of appropriate height. If a shelf of appropriate height for rectangle L_{i+1} does not exist, a new shelf of appropriate height is created above the top-most shelf and rectangle L_{i+1} is packed there, left justified. If a shelf of appropriate height exists, but there is insufficient space to accommodate the rectangle, this shelf is closed off and a new shelf of the same (appropriate) height is created. Shelves that have been closed off are never revisited. The asymptotic worst-case performance bound for the NFS_r algorithm,

$$\text{NFS}_r(\mathcal{L}) < \left\lceil \frac{2}{r} + \frac{1}{r(1-r)} \right\rceil \text{OPT}(\mathcal{L}), \tag{3.2}$$

[3]The NFDH and FFDH algorithms were discussed fully in §2.1.1.1 and §2.1.1.2 respectively.

was established in [6] in the limit as $n \to \infty$, where $\text{NFS}_r(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the NFS_r heuristic, and where $\text{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. A pseudocode listing for the NFS_r algorithm is given as Algorithm 16.

Example 3.5 *The* NFS_r *algorithm was applied to the list of rectangles in Table 1.2 and a value of* $r = 0.2$ *is used for illustrative purposes. Computing the shelf height suitable for packing rectangle* L_1*, the real number* $k = \ln 4 / \ln 0.2 \approx -0.861$ *is used as guideline to find the interval* $0.2^{(-1+1)} < h(L_1) \leq 0.2^{-1}$*, with* $k = -1$*. All rectangles whose heights lie within this interval may be packed on shelf height 5. The process is repeated until all rectangles are packed. A total packing height of 46 units is obtained via the* $\text{NFS}_{0.2}$ *algorithm, as shown in Figure 3.3(a).* □

Algorithm 16 The next fit shelf (NFS_r) algorithm

Description: Packing rectangles in a given order into a strip of fixed width and infinite height.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$, the parameter r and the strip width W.
Output: The height H and the entire packing.

1: shelf $\leftarrow 1$; $w(\text{shelf}) \leftarrow 0$; $h(\text{shelf}) \leftarrow r^k$ for some integer k; $i \leftarrow 0$; $H \leftarrow h(\text{shelf})$
2: **while** there is an unpacked rectangle **do**
3: $i \leftarrow i + 1$; compute k
4: **if** there is sufficient space **and** $r^{k+1} < h(L_i) \leq r^k$ **then**
5: pack rectangle to the right of the previously packed rectangle
6: **else** [there is insufficient space]
7: create a new shelf on top of the previous one and pack rectangle L_i on the new shelf.
8: shelf $\leftarrow$ shelf $+ 1$; $H \leftarrow H + r^k$
9: **end if**
10: **end while**
11: **print** H and entire packing.

The NFS_r algorithm is a linear time algorithm. It has a worst-case time complexity of $O(n)$, because the while-loop spanning lines 2–11 is executed exactly $n + 1$ times and all other steps have constant time complexity.

3.1.2.2 The first fit shelf algorithm

The *first fit shelf* (FFS_r) algorithm [6] with parameter $0 < r < 1$ was also developed in 1983 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

It is a modification of the FFDH algorithm (see §2.1.1.2) and it is similar to the NFS_r algorithm, except that a rectangle is placed left justified on the *lowest* shelf of appropriate height instead of on the highest shelf of appropriate height. This means that all shelves remain active (*i.e.* may be revisited) and before a rectangle is packed, the strip is searched from the bottom upwards. If such a shelf does not exist, then a new shelf of appropriate

height is created above the top-most shelf. An analysis of the asymptotic worst-case bound in [6] indicated that

$$\mathrm{FFS}_r(\mathcal{L}) < \left\lceil \frac{1.7}{r} + \frac{1}{r(1-r)} \right\rceil \mathrm{OPT}(\mathcal{L}), \tag{3.3}$$

in the limit as $n \to \infty$, where $\mathrm{FFS}_r(\mathcal{L})$ denotes the packing height achieved for the list of rectangles $\mathcal{L}$ by the FFS_r heuristic and $\mathrm{OPT}(\mathcal{L})$ is the optimal packing height for the list of rectangles $\mathcal{L}$. The algorithm is given in pseudocode form as Algorithm 17.

Example 3.6 *When applied to the rectangles in Table 1.2, a total packing height of 46 units is also obtained by the* $\mathrm{FFS}_{0.2}$ *algorithm, as illustrated in Figure 3.3(b). The same procedure of computing a value of k to obtain the shelf of appropriate height for each rectangle is followed as described in Example 3.5. The only difference is that rectangles are packed in the lowest shelf of appropriate height. Rectangle* L_9 *is packed on the second shelf instead of on the sixth shelf, which is the lowest of appropriate height for rectangle* L_9. □

Algorithm 17 The first fit shelf (FFS_r) algorithm

Description: Packing rectangles in a given order into a strip of fixed width and infinite height.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$, the parameter r and the strip width W.
Output: The height H and the entire packing.

1: shelf $\leftarrow 1$; $h(\text{shelf}) \leftarrow r^k$ for some integer k; $i \leftarrow 0$; $H \leftarrow h(\text{shelf})$; shelfnum $\leftarrow 1$
2: **while** there is an unpacked rectangle **do**
3: $i \leftarrow i + 1$; compute k
4: search for lowest shelf of appropriate height with sufficient space
5: **if** such a shelf exists **then**
6: pack rectangle to the right of the previously packed rectangle
7: **else** [there is insufficient space in all shelves of appropriate height or such a shelf does not exist]
8: create a new shelf above the top-most shelf and pack rectangle L_i on the new shelf.
9: shelf $\leftarrow$ shelfnum $+ 1$; $H \leftarrow H + r^k$; shelfnum $\leftarrow$ shelfnum $+ 1$
10: **end if**
11: **end while**
12: **print** H and entire packing.

The FFS_r algorithm is a nonlinear-time algorithm because it is based on the first-fit rule that all shelves of appropriate height may be revisited. Suppose the rectangles in the list all have the same height. Then they may be packed on shelves with the same height. The problem thus reduces to the first fit one dimensional bin packing problem which has a worst-case time complexity of $O(n \log n)$. If, on the other hand, the rectangles in the list have varying heights so that each requires packing on a shelf with a different height, then the worst-case time complexity becomes $O(n)$. Consequently, the worst-case time complexity of the FFS_r is $O(n \log n)$ [25].

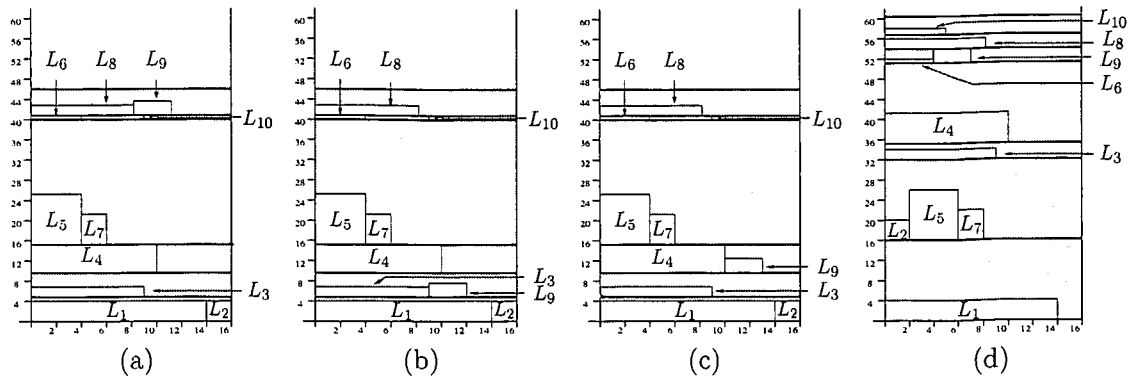

Figure 3.3: *Comparison of packing heights produced by different shelf algorithms when applied to the list of rectangles in Table 1.2: (a) the* $\text{NFS}_{0.2}$ *algorithm packing, (b) the* $\text{FFS}_{0.2}$ *algorithm packing, (c) the* $\text{BFS}_{0.2}$ *algorithm packing and (d) the* $\text{HS}_{4_{0.2}}$ *algorithm packing.*

3.1.2.3 The best fit shelf algorithm

The *best fit shelf* (BFS_r) algorithm [26] with parameter $0 < r < 1$ was developed in 1990 to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

It is a modification of the *best fit decreasing height* (BFDH) algorithm (see §2.1.1.3). The difference between the FFS_r and BFS_r algorithms is that once the parameters r (for the entire packing) and k (for a particular rectangle) have been determined, the latter procedure packs a rectangle on the lowest shelf of appropriate height with minimum residual horizontal space. If a rectangle does not fit on any of the existing shelves of appropriate height, a new shelf is created above the top-most shelf. An average-case analysis of the BFS_r algorithm [26] shows that the expected wasted space[4] is

$$E[W^{\text{BFS}_r}(\mathcal{L}, s)] = \frac{n}{4s} + \Theta\left(\sqrt{ns}\log^{3/4}(n/s)\right) \tag{3.4}$$

in the limit as $n \to \infty$, where $W^{\text{BFS}_r}(\mathcal{L}, s)$ denotes the wasted space created by the BFS_r algorithm when employing s shelf heights for the list of rectangles $\mathcal{L}$, where $E[\cdot]$ denotes the expected value operator, and where Θ denotes the asymptotically tight order notation. A pseudocode listing of the algorithm is given as Algorithm 18.

Example 3.7 *When the* $\text{BFS}_{0.2}$ *algorithm is applied to rectangles in Table 1.2, a total packing height of 46 units is again achieved, as shown in Figure 3.3(c). The computation of k is similar to that in Examples 3.5 and 3.6, with the difference that a rectangle is packed on the shelf of appropriate height with minimum residual horizontal space. Rectangle L_9 is this time packed on the third shelf, because it is of appropriate height and has minimum residual space.* □

The BFS_r algorithm is also a nonlinear-time algorithm [26]. The reasoning is analogous to that for the FFS_r algorithm: if the rectangles in the list all have the same height then

[4] $W^{\text{A}}(\mathcal{L}) = \text{A}(\mathcal{L}) - \sum_{i=1}^{n} w(L_i)h(L_i)$, where $\text{A}(\mathcal{L})$ is the packing height produced by algorithm A when rectangles in $\mathcal{L}$ are packed and $w(L_i), h(L_i)$ are the widths and heights of the rectangles respectively. This is the wasted space created by algorithm A when packing the list of rectangles $\mathcal{L}$.

Algorithm 18 The best fit shelf (BFS_r) algorithm

Description: Packing rectangles in a given order into a strip of fixed width and infinite height.
Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i) \rangle$, the parameter r and the strip width W.
Output: The height H and the entire packing.

1: shelf $\leftarrow$ 1; h(shelf) $\leftarrow r^k$ for some integer k; $i \leftarrow 0$; $H \leftarrow h$(shelf); shelfnum $\leftarrow$ 1
2: **while** there is an unpacked rectangle **do**
3: $i \leftarrow i + 1$; compute k
4: search all shelves (starting with the bottom) for one with appropriate height and has minimum residual horizontal space.
5: **if** such a shelf exists **then**
6: pack rectangle to the right of the previously packed rectangle
7: **else** [there is insufficient space **or** there is no shelf of appropriate height]
8: create a new shelf above the top-most shelf and pack the rectangle L_i on the new shelf.
9: shelf $\leftarrow$ shelfnum + 1; $H \leftarrow H + r^k$; shelfnum $\leftarrow$ shelfnum + 1
10: **end if**
11: **end while**
12: **print** H and entire packing.

the problem becomes a best fit one dimensional packing problem which has worst-case time complexity of $O(n \log n)$. The overall complexity of the BFS_r algorithm is then $O(n \log n)$ [25].

3.1.2.4 The harmonic shelf algorithm

In 1993, the *harmonic shelf* (HS_{M_r}) algorithm [33] was developed to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

Csirik and Woeginger [33] combined the one dimensional bin packing algorithm, called the $harmonic_M$ algorithm, proposed in 1985 by Lee and Lee [72] with the principles of shelf algorithms. The $harmonic_M$ algorithm is used to partition the interval (0,1] non-uniformly into M intervals $I_1, \ldots, I_M$. A reasonable value of M is considered to be in the range $3 \leq M \leq 12$. The intervals are such that $I_p = (1/(p+1), 1/p]$, $1 \leq p < M$ and $I_M = (0, 1/M]$. This harmonic partition allows rectangles to be classified according to the interval into which their widths fit.

The *harmonic shelf* (HS_{M_r}) algorithm does not only aim to pack rectangles of similar heights on the same shelf; over and above this objective the rectangles should also have similar widths. Before rectangle L_i is packed, two decisions have to be made. The first decision is to determine the appropriate shelves onto which a rectangle may be packed in terms of its height, by selecting a value for r and computing a value of k such that $r^{k+1} < h(L_i) \leq r^k$. The second decision is to determine the interval into which the rectangle width fits by computing the value of p such that $1/(p+1) < w(L_i) < 1/p$. If no shelf of appropriate height exists or if there is insufficient horizontal space on all shelves of appropriate height, then a new shelf of the appropriate height is created above the

current top-most shelf. The asymptotic worst-case ratio[5]

$$\mathrm{HS}_{M,r}(\mathcal{L}) \leq (h_\infty + \delta)\,\mathrm{OPT}(\mathcal{L})\,, \tag{3.5}$$

was found for the harmonic shelf algorithm for any $\delta > 0$, in the limit as $n \to \infty$ [33], where $\mathrm{HS}_{M,r}(\mathcal{L})$ is the packing height obtained by the harmonic algorithm when packing rectangles in the list $\mathcal{L}$ and $\mathrm{OPT}(\mathcal{L})$ is the optimal packing height of rectangles in the list $\mathcal{L}$. The algorithm is given in pseudocode form as Algorithm 19.

Example 3.8 *When applied to the rectangles in Table 2.1, whose dimensions have been scaled such that strip width is 1 and rectangle dimensions are in the range (0,1], a total packing height of 60.8 units is obtained via the* $\mathrm{HS}_{4_{0.2}}$ *algorithm, as depicted in Figure 3.3(d). Values of* $M = 4$ *and* $r = 0.2$ *were used in this example for illustrative purposes. Values of* k *are computed as explained in Example 3.5 and this gives the shelf height suitable for packing each rectangle. Starting with the first rectangle, the interval to which the width* $w(L_1)$ *belongs is determined by computing a value of* p *such that* $1/(p+1) < w(L_1) < 1/p$. *Rectangles whose heights are less than 4 and whose widths lie in the interval (0.5,1] for a value of* p *equal to 1 are packed on the first shelf. The procedure is continued until all rectangles are packed.* □

Algorithm 19 The harmonic shelf ($\mathrm{HS}_{M,r}$) algorithm

Description: Packing a given list of rectangles into a strip of fixed width and infinite height, without pre-ordering the list.

Input: The dimensions of the rectangles $\langle w(L_i) \times h(L_i)\rangle$, the parameters M, r and the strip width W.

Output: The height H and the entire packing.

1: shelf $\leftarrow 1$; h(shelf) $\leftarrow r^k$ for some integer k; $i \leftarrow 0$
2: **while** there is an unpacked rectangle **do**
3: $\quad i \leftarrow i+1$; compute k such that $r^{k+1} < h(L_i) \leq r^k$
4: $\quad$ compute the value of p such that $1/p + 1 < w(L_i) \leq 1/p$ where $1 \leq p < M$
5: $\quad$ amongst shelves of height r^k find the one with packing rectangles whose width fits in the interval I_p
6: $\quad$ **if** there is insufficient space **or** no rectangle of height r^k **then**
7: $\quad\quad$ A new shelf of appropriate height is created above the top-most shelf
8: $\quad\quad$ shelf $\leftarrow$ shelf $+ 1$
9: $\quad$ **end if**
10: $\quad$ **if** rectangle L_i is the first on the shelf **then**
11: $\quad\quad$ h(shelf) $\leftarrow r^k$; $H \leftarrow H + h$(shelf)
12: $\quad$ **end if**
13: **end while**
14: **print** H and entire packing

The $\mathrm{HS}_{M,r}$ algorithm is a linear time algorithm, *i.e.* has a worst-case time complexity of $O(n)$. The while-loop spanning lines 1–11 in Algorithm 19 is executed exactly $n+1$ times, while all other steps have constant time complexity.

[5]The number h_∞ is defined as $h_\infty = \sum_{i=1}^{\infty} \frac{1}{t_i - 1} \approx 1.69103$ where $t_1 = 2$, $t_{i+1} = t_i(t_i - 1) + 1$ for $i \geq 1$ [33].

3.1.3 Special case algorithms

The first special case under consideration is that of allowing for the modification of rectangular shapes. Imreh [62] explored the idea of modifying the shapes of the rectangles in such a way that the area of each rectangle remains constant. Applications in which such modifications are permissible include task scheduling where the two dimensions of a rectangle represent resource and time. The width and height of the strip represent the maximal resource and the time used respectively. The objective is obviously to minimise the time taken to complete a task. Lengthening a rectangle amounts to using less resource and more time to complete the task. The rectangles may neither be rotated nor may they overlap. Each rectangle is still packed in an *on-line* fashion, without prior knowledge of further rectangles. The algorithms obeying the modification of shapes are described in §3.1.3.1–3.1.3.3. The second special case considered is that of assuming rectangles arrive from the top of the strip and must reach a suitable location within the strip without being blocked by any of the rectangles already packed—once placed, rectangles may not be moved again. The special case algorithms are referred to as those obeying the *tetris* constraint since the analogy is made to the *tetris* game[6]. The algorithms obeying the *tetris* constraint are described in §3.1.3.4–3.1.3.5.

3.1.3.1 The xS algorithm

The *xS* algorithm [62] was developed in 2001 to solve problems of the form

2D	R	SP	On	MiS	0,0,1,1

.

The algorithm depends on a parameter $x > 1$ and the strip is partitioned into shelves which are rectangular parts of the strip, each of width 1, without loss of generality. Initially a shelf of height $x > 1$ is created and this is called the active shelf. If the next rectangle fits on the active shelf, then it is lengthened to have height x before it is packed. If it does not fit, then a new shelf of height x is created and the rectangle is packed there. This new shelf then becomes the new active shelf. The process is repeated until all rectangles in $\mathcal{L}$ have been packed.

It was shown in [62] that the xS algorithm has asymptotic performance ratio

$$\mathrm{xS}(\mathcal{L}) \leq \left(1 + \frac{1}{x+1}\right) \mathrm{OPT}(\mathcal{L}), \tag{3.6}$$

in the limit as $n \to \infty$, where $\mathrm{xS}(\mathcal{L})$ is the packing height obtained by the xS algorithm when packing rectangles in the list $\mathcal{L}$ and $\mathrm{OPT}(\mathcal{L})$ is the optimal packing height of the rectangles in the list $\mathcal{L}$. The algorithm is given in pseudocode form as Algorithm 20.

Example 3.9 *A value of $x = 2$ is selected for illustrative purposes. When the xS algorithm is applied to the rectangles in Table 2.1, whose dimensions are scaled to fall within the range (0,1] with strip width 1, a total packing height of $2 \times 16 = 32$ units is obtained, as shown in Figure 3.4(a). The first shelf of height 2×16 is created and rectangle L_1, whose area is approximately 0.21×16, is lengthened such that $h(L_1) = 2 \times 16$ and $w(L_1) = (0.21/2) \times 16 \approx 0.109 \times 16$. All other rectangles fit on this shelf.* ◙

[6]The Tetris game was originally developed in 1985-86 by Alexey Pajitnov, Dmitry Pavlovsky and Vadim Gerasimov. It is a very popular game which is a registered trademark of The Tetris Company.

Algorithm 20 The xS Algorithm

Description: Packing a rectangle without prior knowledge of the next rectangle in a list of rectangles to be packed, so as to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and the entire packing.

```
1: create a shelf of height x > 1; let this shelf be the active shelf
2: shelf ← 1; h(shelf) ← x; i ← 1
3: while a rectangle is available for packing do
4:    i ← i + 1; Area ← w(L_i)h(L_i)
5:    lengthen the rectangle to have height x but keeping the area constant
6:    h(L_i) ← x; w(L_i) ← Area/h(L_i)
7:    if there is sufficient space then
8:       pack rectangle into active shelf
9:    else [there is insufficient space]
10:      create new shelf with new height x
11:      shelf ← shelf + 1; h(shelf) ← x
12:      pack rectangle L_i into new shelf
13:   end if
14: end while
15: print H = x × shelf and entire packing
```

The xS algorithm has worst-case time complexity of $O(n)$ [63]. The while-loop spanning lines 3–13 in Algorithm 20 has a time complexity of $O(n)$ and lines 4–12 have a constant time complexity. Lines 1 and 2 also have a constant time complexity.

3.1.3.2 The DS algorithm

The *DS* algorithm [62], also developed in 2001, may be used to solve problems of the form

2D	R	SP	On	MiS	0,0,1,1

.

In this algorithm, the first shelf is assigned the height of the first rectangle packed into it and this is called the active shelf. If it is possible to pack the next rectangle in the active shelf, this is done, after first lengthening it. Otherwise a new shelf which is twice the height of the active shelf is created and this becomes the new active shelf. The process is continued until all rectangles in $\mathcal{L}$ are packed. The DS algorithm has asymptotic performance ratio

$$\mathrm{DS}(\mathcal{L}) \leq 4\,\mathrm{OPT}(\mathcal{L}) \tag{3.7}$$

in the limit as $n \to \infty$, where $\mathrm{DS}(\mathcal{L})$ is the packing height obtained by the DS algorithm when packing rectangles in the list $\mathcal{L}$ and $\mathrm{OPT}(\mathcal{L})$ is the optimal packing height of rectangles in the list $\mathcal{L}$. The algorithm is given in pseudocode form as Algorithm 21.

Example 3.10 *When applied to the rectangles in Table 1.2, a total packing height of 28 units is obtained, as shown in Figure 3.4(b). Rectangles L_1 and L_2 are packed on the first shelf, which has height $h(L_1)$. Rectangle L_3 does not fit on the first shelf; hence a new shelf of height $2 \times h(L_1) = 8$ is created and rectangle L_3 is packed thereafter, lengthening its height to 8. The process continues until all rectangles are packed.* ∎

Algorithm 21 The DS Algorithm

Description: Packing a rectangle without prior knowledge of the next rectangle in a list of rectangles to be packed, so as to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and the entire packing.

1: $i \leftarrow 1$; shelf $\leftarrow 1$; $h(\text{shelf}) \leftarrow h(L_i)$; $H \leftarrow h(\text{shelf})$
2: pack the first rectangle
3: **while** a rectangle is available for packing **do**
4: $i \leftarrow i + 1$; $Area \leftarrow w(L_i)h(L_i)$
5: lengthen the rectangle to have height $h(\text{shelf})$ but keeping the area constant
6: $h(L_i) \leftarrow h(\text{shelf}); w(L_i) \leftarrow Area/h(L_i)$
7: **if** there is sufficient space **then**
8: pack rectangle L_i on the current shelf
9: **else** [there is insufficient space]
10: shelf $\leftarrow$ shelf $+ 1$; $h(\text{shelf}) \leftarrow 2 \times h(\text{shelf} - 1)$; $H \leftarrow H + h(\text{shelf})$
11: go to step 5
12: **end if**
13: **end while**
14: **print** H and entire packing

Similar to the xS algorithm, the DS algorithm also has worst-case time complexity of $O(n)$ [62]. All steps in Algorithm 21 have constant time complexity, except for the while-loop spanning lines 3–12 which is executed $n + 1$ times.

3.1.3.3 The FFS1 algorithm

In 2001 Imreh [62] proposed another algorithm, called the *FFS1* algorithm, for solving problems of the form

2D	R	SP	Aon	MiS	0,0,1,1

that are not entirely *on-line* (*i.e.* some information about the rectangles to be packed is known, such as that the rectangles are ordered according to non-increasing height). In this algorithm, a shelf of height $h(L_1)$ is first created, where $h(L_1)$ is the height of the first rectangle (the tallest one). The next rectangle is packed on the lowest shelf into which it fits, after first having lengthened it. Otherwise, a new shelf of height $h(L_1)$ is created and the rectangle is packed there. This process is repeated until all rectangles in $\mathcal{L}$ are packed. This algorithm was proved to have the absolute performance ratio

$$\text{FFS1}(\mathcal{L}) \leq 2\,\text{OPT}(\mathcal{L}) \tag{3.8}$$

in the limit as $n \to \infty$, where $\text{FFS1}(\mathcal{L})$ is the packing height obtained by the FFS1 algorithm when packing rectangles in the list $\mathcal{L}$ and $\text{OPT}(\mathcal{L})$ is the optimal packing height of rectangles in the list $\mathcal{L}$. The algorithm is given in pseudocode form as Algorithm 22.

Example 3.11 *A total packing height of 20 units is obtained when the FFS1 algorithm is applied to the rectangles in Table 1.2, as shown in Figure 3.4(c). Because the rectangles are already pre-ordered according to non-increasing height, the first shelf of height*

$h(L_5)$ is created and rectangle L_5 is packed there. Rectangles $L_7, L_4, L_2, L_9, L_{10}, L_3, L_6$ are packed consecutively on this shelf, after lengthening their heights to $h(L_5)$. A second shelf of height $h(L_5)$ is created since there is insufficient space to pack the next rectangle. Rectangles L_1 and L_8 are packed on this shelf, after having been lengthened first. □

Algorithm 22 The FFS1 Algorithm

Description: Packing a list of rectangles with prior knowledge about the ordering, to obtain minimum strip height.

Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles which are pre-ordered according to non-increasing height, and the strip width W.

Output: The height H and the entire packing.

1: $i \leftarrow 1$; shelf $\leftarrow 1$; $h(\text{shelf}) \leftarrow h(L_1)$
2: pack L_i into the active shelf
3: **while** a rectangle is available for packing **do**
4: $i \leftarrow i + 1$; $Area \leftarrow w(L_i)h(L_i)$
5: lengthen the rectangle to have height $h(\text{shelf})$ but keeping the area constant
6: $h(L_i) \leftarrow h(\text{shelf})$; $w(L_i) \leftarrow Area/h(L_i)$
7: search for lowest shelf with sufficient space
8: **if** such a shelf exists **then**
9: pack rectangle into the shelf
10: **else** [such a shelf does not exist]
11: create a new shelf above the top-most shelf
12: shelf $\leftarrow$ shelf $+ 1$; $h(\text{shelf}) \leftarrow h(L_1)$
13: pack rectangle into the new shelf
14: **end if**
15: **end while**
16: **print** $H = \text{shelf} \times h(L_1)$ and entire packing

If all the rectangles in the list have the same height, then the problem reduces to first fit one dimensional packing problem with a worst-case time complexity of $O(n \log n)$. However, if the heights of rectangles are different and fit on new shelves each time then the algorithm has a time complexity of $O(n)$. Consequently the overall worst-case time complexity of the FFS1 algorithm is $O(n \log n)$.

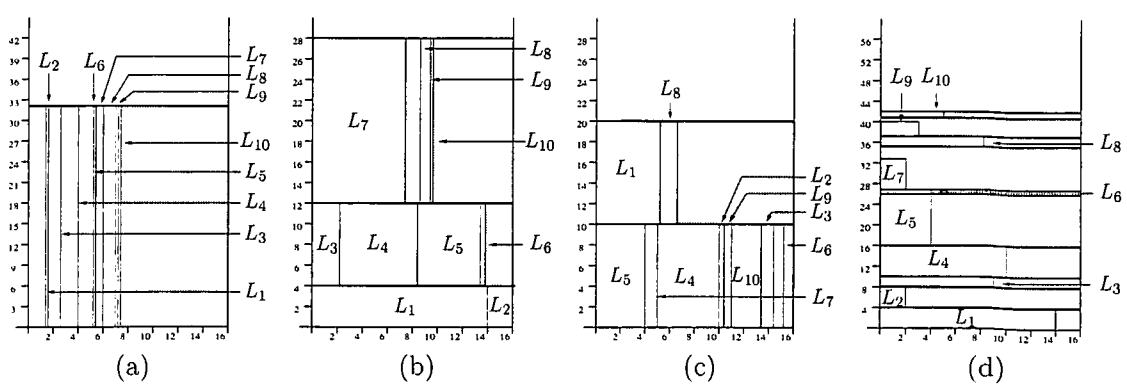

Figure 3.4: *Comparison of packing heights produced by different on-line special case algorithms when applied to the list of rectangles in Table 1.2: (a) the xS algorithm packing, (b) the Ds algorithm packing, (c) the FFS1 algorithm packing and (d) the* $\text{Azar}_{0.25}$ *algorithm packing.*

3.1.3.4 The Azar_Y algorithm

The $Azar_Y$ algorithm [2] was developed in 1996 to solve problems of the form

2D	R	SP	On	MiS	0,1,0,1

.

In this algorithm, the rectangle widths are assumed to be in the range (0,1] and the strip has width 1. However, there is no restriction on the rectangle heights. The Azar_Y algorithm partitions the strip into horizontal levels by means of a real threshold constant $0 < Y < \frac{1}{2}$. Rectangles of particular heights ($2^{j-1} < h(L_i) \leq 2^j$) and widths ($2^{-x-1} < w(L_i) \leq 2^{-x}$) are packed on the same level, referred to as an (x, j) level (where $j \in \mathbb{Z}$ and $x \in \mathbb{N}$).

A rectangle whose width is at least Y is referred to as a *buffer*. When the next rectangle to be packed arrives, it is classified either as a *buffer* or *non-buffer*, depending on its width. A *buffer* may block *non-buffers* still to be packed from reaching a suitable level since previously packed levels may be revisited. If the next rectangle to be packed is a *buffer*, a new level, whose height is equal to the height of the *buffer*, is created above the top-most level and the rectangle is packed there, left justified. This means that *buffers* are packed on their own within levels. If, on the other hand, the rectangle is a *non-buffer*, it is classified as an (x, j) rectangle, for some $j \in \mathbb{Z}$ and some $x \in \mathbb{N}$. The first *non-buffer* rectangle packed on a level determines the height of the level as 2^j and this level becomes an (x, j) level. If a rectangle fits on an (x, j) level and it can reach such a level without being blocked by any of the *buffers* then it is placed on that level. However, if no such level exists, then a new level of height 2^j is created above the top-most level. It was established in [2] that the Azar_Y algorithm has performance ratio

$$\text{Azar}_Y(\mathcal{L}) \leq O\left(\log \frac{1}{\epsilon}\right) \text{OPT}(\mathcal{L}) \tag{3.9}$$

as $n \to \infty$, where ϵ is the minimum width of each rectangle, where $\text{Azar}_Y(\mathcal{L})$ is the packing height obtained by the Azar algorithm when packing rectangles in the list $\mathcal{L}$, and where $\text{OPT}(\mathcal{L})$ is the optimal packing height of rectangles in the list $\mathcal{L}$. A pseudocode listing of the algorithm is given as Algorithm 23.

Example 3.12 *When applied to the rectangles with scaled dimensions in Table 2.1, a total packing height of 42 units is obtained via the* $\text{Azar}_{0.25}$ *algorithm, as depicted in Figure 3.4(d), where the value of* $Y = 0.25$ *was chosen for illustrative purposes. Rectangle* L_1 *is a buffer and it is packed on its own on the first level. By computing the values of* x *and* j*, rectangle* L_2 *is packed on a* $(3, -2)$ *level whose height is* $2^{-2} = 0.25$*. This process is continued until all the rectangles have been packed. Rectangles* $L_1, L_3, L_4, L_5, L_6, L_8$ *are also buffers because their widths are at least 0.25.* ∎

If all the rectangles are buffers, then the time complexity of the Azar_Y is $O(n)$, since each rectangle will be packed on a new level. If none of the rectangles are buffers, then the algorithm reduces to the HS_{M_r} algorithm (since a rectangle is classified according to its width and height) which also has a linear time complexity. This will also apply if there are buffers and rectangles are not blocked. Therefore the overall worst-case time complexity of the Azar_Y algorithm is $O(n)$.

Algorithm 23 The Azary algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.

Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles, the parameter Y and the strip width W.

Output: The height H and the entire packing.

1: $h(\text{level}) \leftarrow 0; w(\text{level}) \leftarrow 0$; level $\leftarrow 0$
2: **while** there is an unpacked rectangle **do**
3: **if** $w(L_i) \geq Y$ **then**
4: level $\leftarrow$ level $+ 1$; $w(\text{level}) \leftarrow w(L_i)$; $h(\text{level}) \leftarrow h(L_i)$; $H \leftarrow H + h(\text{level})$
5: **else** $[w(L_i) < Y]$
6: compute x and j such that $(2^{j-1} < h(L_i) \leq 2^j)$ and $(2^{-x-1} < w(L_i) \leq 2^{-x})$
7: search for the lowest (x, j) level
8: **if** no (x, j) level is available or L_i is blocked **then**
9: level $\leftarrow$ level $+ 1$; $h(\text{level}) \leftarrow 2^j$; $H \leftarrow H + h(\text{level})$
10: $w(\text{level}) \leftarrow w(\text{level}) + w(L_i)$
11: **else** $[(x, j)$ level is available and not blocked$]$
12: $w(\text{level}) \leftarrow w(\text{level}) + w(L_i)$
13: **end if**
14: **end if**
15: **end while**
16: **print** H and entire packing

3.1.3.5 The compression algorithm

The *compression* (CA) algorithm [29] is an extension of the BiNFL algorithm (see §3.1.1.4). It was developed in 2002 to solve problems of the form

2D	R	SP	On	MiS	0,1,0,1

.

The algorithm exploits certain patterns (when only one or two rectangles are packed on the *lower* level) that result from a BiNFL packing. In the CA algorithm, packing on the *lower* level is similar to a BiNFL packing. On the *upper* level, however, if rectangle L_i $(i \geq 3)$ is the first rectangle to be packed, it is justified according to the shorter of the first left justified and first right justified rectangles on the *lower* level, and it is shifted down onto the *lower* level, provided that there is sufficient space (see Figures 3.2(b) and (c)). Such a rectangle that has been shifted down into the *lower* level is referred to as a *compressed rectangle*. If rectangle L_i $(i \geq 3)$ is the second rectangle to be packed (there is one rectangle on each level, each of them left justified), it is right justified and if there is sufficient room on the *lower* level, this rectangle is shifted down onto the *lower* level. Subsequent rectangles that fit on the *lower* level may also be shifted down next to previously compressed rectangles. Packing continues on the *upper* level as in the BiNFL algorithm for rectangles that may not be shifted downwards. A rectangle that fails to fit on the *upper* level is placed in a new bi-level that is created above the top-most level and the previous bi-level is closed off. The asymptotic expected height of a packing produced by the CA algorithm is

$$E[\text{CA}(\mathcal{L})] \approx (0.369\,764\,213\ldots)n \tag{3.10}$$

in the limit as $n \to \infty$ [29], where $E[\cdot]$ is the expected value operator and where $\text{CA}(\mathcal{L})$ denotes packing height achieved for the list of rectangles $\mathcal{L}$ by the CA algorithm. The algorithm is given in pseudocode form as Algorithm 24.

Example 3.13 *For the example instance in Table 1.2, a total packing height of 26 units is obtained via the CA algorithm, as shown in Figure 3.5(a). Rectangles L_1 and L_2 are packed on the lower level, while rectangle L_3 is packed on the upper level of the first bi-level. A new bi-level is created, where rectangle L_4 is left justified on the lower level and rectangle L_5 is right justified on the lower level. Because rectangle L_6 fails to fit on the lower level, it is packed left justified on the upper level, because rectangle L_4 is shorter than rectangle L_5. Rectangle L_6 is then slid downwards onto the lower level such that it rests on top of rectangle L_4. Rectangles L_7 and L_8 are the left justified on the upper level. The process is continued until all rectangles have been packed.* ◻

Algorithm 24 The CA algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and the entire packing.

```
1: i ← 0; H ← 0
2: open new bi-level
3: packing on lower level is similar to BiNFL (§3.1.1.4); H ← H + h(lowerlevel)
4: if there is insufficient space to pack a rectangle on the lower level then
5:    call pack on the upper level
6: end if
7: print H and entire packing
```

Procedure pack on the upper level

```
1: while there is an unpacked rectangle do
2:    if i ≥ 3 and L_i is the first rectangle packed then
3:       justify according to the shorter of the left-most and right-most rectangles
4:       slide rectangle downwards if there is sufficient space; go to 12
5:    else
6:       if i ≥ 3 and L_i is the second rectangle packed then
7:          right justify and go to 12
8:       else [rectangle does not fit]
9:          H ← H + h(upperlevel); open new bi-level
10:      end if
11:   end if
12: end while
```

If there are no rectangles to be compressed, then the CA algorithm reduces to the BiNFL algorithm which has a worst-case time complexity of $O(n)$. Therefore the overall worst-case time complexity of the CA algorithm is $O(n)$. As each rectangle arrives, only a constant amount of time is taken to pack it. In the CA algorithm only two levels are packed at a time, and these are levels with patterns containing 3 or 4 rectangles—this repacking time is bounded by a constant [39].

3.2 Possible improvements to known heuristics

A total of six modifications to some of the heuristics described in §3.1 are proposed in this section.

3.2.1 Level algorithms

This section resumes with the modifications to the level algorithms described in §3.1.1.

3.2.1.1 The modified next fit level algorithm

As the name suggests, the *modified next fit level* (MNFL) algorithm is a newly proposed variation to the NFL algorithm described in §3.1.1.1. In the MNFL algorithm, the first rectangle packed on a level determines the height of that level. If a rectangle is encountered that does not fit onto the current level, the level is closed off (never to be revisited) and a new current level is created above this level. The algorithm is expected to perform poorly if the rectangles are presented in an order in which they tend to increase in height. However, if the rectangles are presented in an order in which they tend to decrease in height, then the algorithm is expected to perform well. The MNFL algorithm differs from the NFL algorithm in that in the latter procedure, level heights are determined by the tallest rectangle packed on a level, while in the former procedure, level heights are determined by the first rectangle packed on the level. When applied to rectangles in Table 1.2, a total packing height of 25 units is obtained via the MNFL algorithm, as shown in Figure 3.5(b).

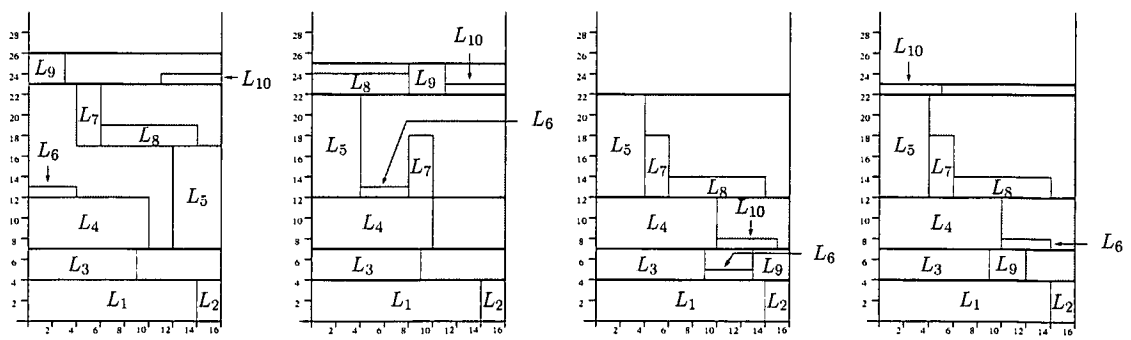

Figure 3.5: *Comparison of packing heights produced by CA, MNFL, MFFL and MBFL algorithms when applied to the list of rectangles in Table 1.2: (a) the CA algorithm packing (b) the MNFL algorithm packing, (c) the MFFL algorithm packing and (d) the MBFL algorithm packing.*

3.2.1.2 The modified first fit level algorithm

In the *modified first fit level* (MFFL) algorithm, the height of each level corresponds to the height of the first rectangle packed on that level. The strip is searched, one level at a time, from the bottom upwards for sufficient space to pack the next rectangle. If a

rectangle fits height-wise on a level and there is enough horizontal space on the level to accommodate the rectangle, then it is placed there, left justified. If a rectangle does not fit on any of the existing levels, a new level is created above the top-most level and the rectangle is packed on that level, left justified. The MFFL and FFL algorithms differ in a manner analogous to the difference between the MNFL and NFL algorithms. A total packing height of 22 units is obtained via the MFFL algorithm when applied to rectangles in Table 1.2, as illustrated in Figure 3.5(c).

3.2.1.3 The modified best fit level algorithm

The *modified best fit level* (MBFL) algorithm is similar to the BFL algorithm, except that in the MBFL algorithm the height of a level is determined by the height of the first rectangle packed on the level, while in the BFL algorithm the height of a level is determined by the tallest rectangle packed on the level. A rectangle is packed on a level with sufficient space and minimum residual horizontal space. If none of the existing levels have enough room to accommodate a rectangle, a new level is created above the top-most level and the rectangle is placed there, left justified. A total packing height of 23 units is obtained via the MBFL algorithm when applied to rectangles in Table 1.2, as illustrated in Figure 3.5(d).

3.2.2 Special case algorithms

Three modifications to the CA algorithm (see §3.1.3.5) are proposed in this section. Downey [39] noted that the CA algorithm is far from optimal, because it only takes into consideration a few patterns (where it may be possible to slide rectangles from the *upper* to the *lower* level). The proposed modifications seek to take more patterns into consideration where it may be possible to slide down rectangles.

3.2.2.1 The compression part fit algorithm

The *compression part fit* (CPF) algorithm is proposed to accommodate more sliding patterns occurring within a *bi-level*. An idea originally introduced by Burke *et al.* [18] of using a linear array whose number of elements is equal to the width of the strip, is employed. Each element of the linear array is used to store the height of the rectangle packed on a particular level at that coordinate of the linear array. However, the drawback of using the linear array approach is that it requires the dimensions of the rectangles and the strip to be integers.

The bi-level stage

Packing on the *lower* level is exactly similar to that in the BiNFL algorithm, except that a linear array is used to represent the various heights of rectangles packed only on the *lower* level. Before any packing takes place on a bi-level, the linear array is filled with zeros (see Figure 3.6(a)). On the *upper* level, the CPF algorithm differs from the BiNFL algorithm in that rectangles are always packed left justified. A *vertical space* on the *lower* level is defined as the space between the lower boundary of the *upper* level and the upper edge

Algorithm 25 The compression part fit (CPF) algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and entire packing.

1: $i \leftarrow 0$; $H \leftarrow 0$
2: open a new bi-level
3: packing on lower level is similar to BiNFL (§3.1.1.4); $H \leftarrow H + h$(lowerlevel)
4: **if** there is insufficient space to pack a rectangle on the lower level **then**
5: **call** pack on the upper level
6: **end if**
7: **print** H and entire packing

Procedure pack on the upper level

1: **while** there is an unpacked rectangle **do**
2: **if** $h(L_i) >$ VerticalSpace and $w(L_i) \leq$ HorizontalSpace **then**
3: shift rectangle L_i downwards; go to 11
4: **else**
5: **if** $h(L_i) \leq$ VerticalSpace and $W - w$(level) $\geq w(L_i)$ **then**
6: left justify; go to 11
7: **else** [rectangle does not fit]
8: $H \leftarrow H + h$(upperlevel); open a new bi-level
9: **end if**
10: **end if**
11: **end while**

of rectangles packed on the *lower* level (or sometimes the lower boundary of the *lower* level) at each x–coordinate of the linear array. Three *vertical spaces* of heights $2, 4$ and 3 units are indicated by dashed vertical arrows in Figure 3.6(b) at x–coordinates $1, 6$ and 8 respectively. A *horizontal space* on the *lower* level on the other hand, is defined as the space between the left edge of a rectangle being considered for shifting downwards and the nearest left edge of a rectangle packed on the *lower* level at a height given in the linear array at the x–coordinate corresponding to the left edge of the rectangle. A *horizontal space* of 7 units is shown in Figure 3.6(c) by the horizontal dotted arrows computed from the coordinates $x = 2$ to $x = 8$ at a height of 3 (given in the lower linear array, at $x = 2$ which corresponds to the left edge of the sixth rectangle).

The compression stage

A rectangle on the *upper* level is said to be *covered* by a single value (resp. multiple values) of the *vertical space*, if the value of the *vertical space* along the entire width of the rectangle remains constant (resp. changes). The number of values covering a rectangle is determined by the number of times the value of the *vertical space* changes (*i.e.* if the value of the *vertical space* changes four times along the width of the rectangle then the rectangle is said to be covered by 4 values). For a rectangle to be slid downwards onto the *lower* level, two conditions must be satisfied.

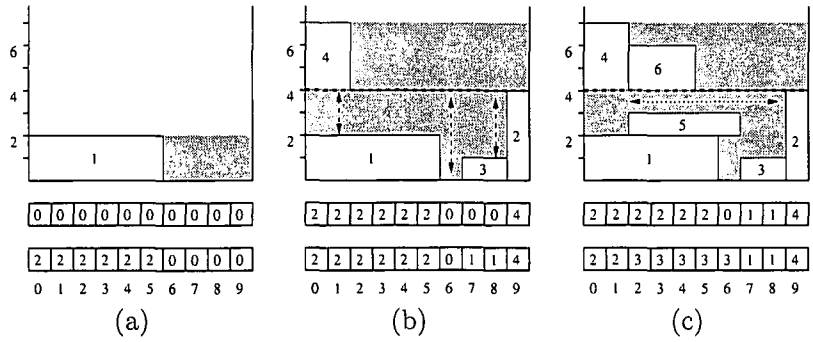

Figure 3.6: *Examples of how the elements of the linear array are filled when a new bi-level is created. (a) The upper linear array with elements of zeros represents a new bi-level with no rectangles packed. The lower linear array stores the heights of the rectangles packed on the* lower *level from x–coordinates $x = 0$ to $x = 5$. (b) The upper linear array stores the heights of the first and second rectangles packed. The lower linear array stores height of the third rectangle and the* vertical space *is indicated by the dashed arrows at certain x–coordinates of the linear array. (c) The fifth rectangle has been shifted onto the* lower *level by the CFF algorithm. The* horizontal space *is indicated by the horizontal dotted arrow.*

1. The height of the rectangle must exceed the value of the *vertical space*. The width of a rectangle may be covered by a single value (Figure 3.7(a)) or multiple values of the *vertical space* (Figure 3.7(b)). If more than one value of the *vertical space* covers the entire width of the rectangle, the height of the rectangle must exceed the shortest value of the *vertical space*.

2. The width of the rectangle must not exceed the width of the *horizontal space*.

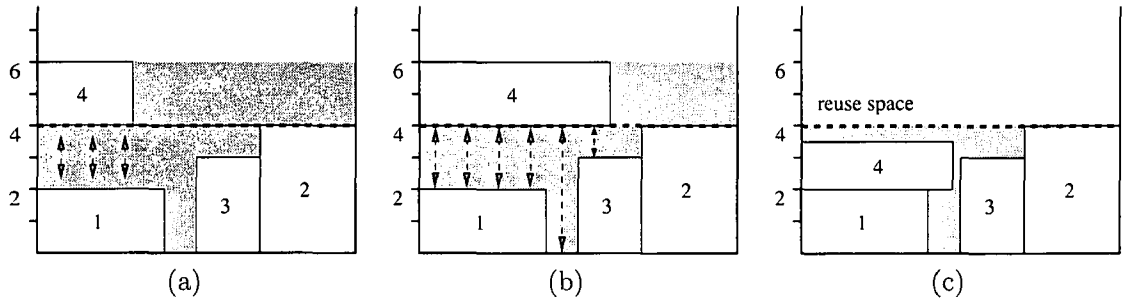

Figure 3.7: *Examples of how the width of a rectangle is* covered *by: (a) one value of the vertical space, or (b) multiple values of the vertical space. In (c), after sliding a rectangle downwards onto the lower level, the space it was supposed to occupy on the upper level may be reused to pack other rectangles.*

Provided that the two conditions above are satisfied, the rectangle in question is slid down so that its lower edge rests on the top edge of the rectangle on the lower level or on the lower level itself. If multiple values of the *vertical space* cover the width of the rectangle, the bottom edge of the shifted rectangle should coincide with the top edge of the rectangle corresponding to the shortest *vertical space*. The algorithm is expected to perform well

if the tallest rectangle on most *upper* levels may be shifted onto the corresponding *lower* levels. The algorithm is given in pseudocode form as Algorithm 25.

Example 3.14 *A total packing height of 22 units is obtained when CPF algorithm is applied to the rectangles in Table 1.2, as shown in Figure 3.8(a). The first bi-level is packed in a similar fashion to Example 3.13. The only difference occurs in the second bi-level, where rectangle L_6 remains on the upper level, while rectangle L_7 is slid downwards onto the lower level because its height exceeds the value of the* vertical space *(both conditions are satisfied). This process is continued until all rectangles have been packed.* □

The while-loop spanning lines 2–13 in Algorithm 25 is executed exactly $n + 1$ times. Packing on the lower level and updating the linear array have a constant time complexity. Similar to the CA algorithm, this repacking time on the upper level in the CPF algorithm is also bounded by a constant. Therefore the CPF algorithm has a worst-case time complexity of $O(n)$.

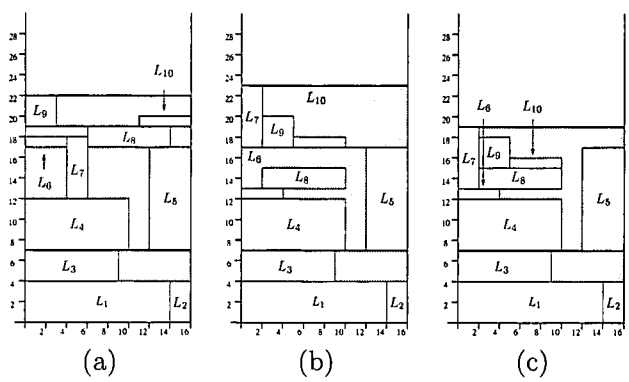

Figure 3.8: *Packings produced by CPF, CFF and CC algorithms for the example instance of the strip packing problem in Table 1.2: (a) the CPF algorithm packing, (b) the CFF algorithm packing and (c) the CC algorithm packing.*

3.2.2.2 The compression full fit algorithm

The steps of another newly proposed modification to the CA algorithm, called the *compression full fit* (CFF) algorithm and CPF algorithm described in the previous section are similar in all aspects, except for condition 1 of the compression stage. In the CFF algorithm, a rectangle is slid down onto the *lower* level provided that its height does not exceed the *vertical space* covering the entire width of the rectangle. The advantage of doing this is that the residual *vertical space* (this is the *vertical space* remaining after a rectangle has been slid downwards) may again be considered when packing the next rectangle. In Figure 3.6(c), before rectangle 5 was slid down, there was a *vertical space* and *horizontal space* of 2 and 7 units respectively at $x = 2$. After rectangle 5 was slid down onto the *lower* level, a *vertical space* of 1 unit resulted. If rectangle 6 had a height of 1 unit, then it would also be slid onto the *lower* level.

The idea in the CFF algorithm is to increase the probability of packing more rectangles on the *upper* level by utilising the space left after compression of a rectangle onto the

Algorithm 26 The compression full fit (CFF) algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and entire packing.

1: $i \leftarrow 0$; $H \leftarrow 0$
2: open new bi-level
3: packing on lower level is similar to BiNFL (§3.1.1.4); $H \leftarrow H + h$(lowerlevel)
4: **if** there is insufficient space to pack a rectangle on the lower level **then**
5: **call** pack on the upper level
6: **end if**
7: **print** H and entire packing

Procedure pack on the upper level

1: **while** there is an unpacked rectangle **do**
2: **if** $h(L_i) \leq$ VerticalSpace and $w(L_i) \leq$ HorizontalSpace **then**
3: slide rectangle L_i downwards; go to 11
4: **else**
5: **if** $h(L_i) \leq$ VerticalSpace and $W - w$(level) $\geq w(L_i)$ **then**
6: left justify; go to 11
7: **else** [rectangle does not fit]
8: $H \leftarrow H + h$(upperlevel); open new bi-level
9: **end if**
10: **end if**
11: **end while**

lower level. Once a rectangle is slid onto the lower level, the space it was supposed to occupy on the upper level may be reused to pack other rectangles. This is illustrated in Figure 3.7(c) where rectangle 4 has been slid onto the *lower* level and the space it was originally meant to occupy on the *upper* level may be used to pack other rectangles. The algorithm is expected to perform better if the tallest rectangle on the *upper* level may be compressed onto the *lower* level and if more rectangles fit onto the *upper* level. The latter implies an increased chance of creating fewer levels, hence leading to a decrease in the overall strip height. The algorithm is given in pseudocode form as Algorithm 26.

Example 3.15 *When the CFF algorithm is applied to the example instance in Table 1.2, a total packing height of 23 units is obtained, as illustrated in Figure 3.8(b). The packing on the first bi-level is similar to that in Example 3.13. On the upper level of the second bi-level, rectangle* L_6 *is slid downwards because its height is less than that of the* vertical space. *Rectangle* L_7 *is packed on the space that was supposed to be occupied by rectangle* L_6 *on the upper level. Rectangle* L_8 *is then also slid downwards from the upper level, because its height is less than that of the* vertical space. *This process is continued until all rectangles have been packed.* ∎

The while-loop spanning lines 2–13 in Algorithm 26 is executed exactly $n + 1$ times. If none of the rectangles may be slid down to the lower level, the CPF algorithm reduces to the BiNFL algorithm which has a time complexity of $O(n)$. The compression stage is also bounded by a constant and packing on the lower level and updating the linear array

have a constant time complexity. Therefore the CPF algorithm has a worst-case time complexity of $O(n)$.

3.2.2.3 The compression combo algorithm

The *compression combo* (CC) algorithm is a combination of the first conditions of the compression stage of both the CPF and CFF algorithms. In the CC algorithm, any rectangle may be slid down onto the *lower* level regardless of whether it fully or partially fits on the *lower* level, as long as the second condition, namely that the width of the rectangle to be slid downwards is at most the width of the horizontal space, is satisfied. The algorithm is given in pseudocode form as Algorithm 27.

Example 3.16 *When the CC algorithm is applied to rectangles in Table 1.2, a total packing height of 19 units is obtained, as illustrated in Figure 3.8(c). The first bi-level is packed similarly to that in Example 3.13. On the second bi-level, however, all the rectangles are slid downwards from the upper level to the lower level, provided there is sufficient space.* □

Algorithm 27 The compression combo (CC) algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and the entire packing.

1: $i \leftarrow 0$; $H \leftarrow 0$
2: open new bi-level
3: packing on lower level is similar to BiNFL (§3.1.1.4); $H \leftarrow H + h$(lowerlevel)
4: **if** there is insufficient space to pack a rectangle on the lower level **then**
5: **call** pack on the upper level
6: **end if**
7: **print** H and entire packing

Procedure pack on the upper level

1: **while** there is an unpacked rectangle **do**
2: **if** $w(L_i) \leq$ HorizontalSpace **then**
3: slide rectangle L_i downwards; go to 11
4: **else**
5: **if** $w(L_i) >$ HorizontalSpace and $W - w$(level) $\geq w(L_i)$ **then**
6: left justify, go to 11
7: **else** [rectangle does not fit]
8: $H \leftarrow H + h$(upperlevel); open new bi-level
9: **end if**
10: **end if**
11: **end while**

Analogous to the analysis of the CPF and CFF algorithms, the CC algorithm also has a worst-case time complexity of $O(n)$. The compression stage is bounded by a constant and the while-loop spanning lines 2–13 is executed exactly $n + 1$ times.

3.3 Five new on-line strip packing heuristics

In this section a total of five new heuristics for on-line packing problems are introduced. In the proposed algorithms, rectangles are not classified according to either their height or width, but rather according to the properties of the rectangles already packed.

Algorithm 28 The shelf deviation (SDev) and shelf difference (SDiff) algorithms

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.

Input: The dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.

Output: The height H and the entire packing

1: $h(type_{1,1}) \leftarrow h(L_1)$; $H \leftarrow h(type_{1,1})$; $w(type_{1,1}) \leftarrow W - w(L_1)$
2: $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$; NumTypes $\leftarrow 1$; NumShelfType$_1 \leftarrow 1$
3: **while** there is a rectangle to be packed **do**
4: $i \leftarrow i + 1$ (going to the next rectangle)
5: **while** $j \leq$ NumTypes **or** rectangle is not packed **do**
6: $k \leftarrow 1$
7: **if** $(j = 1$ **and** $0 < h(L_i) \leq h(type_{1,k}))$ **or** $(j > 1$ **and** $h(type_{j-1,k}) < h(L_i) \leq h(type_{j,k}))$ **then**
8: **while** $k \leq$ NumShelfType$_j$ **do**
9: **if** $w(type_{j,k}) \geq w(L_i)$ **then**
10: pack rectangle
11: **else** $[w(type_{j,k}) < w(L_i)]$
12: $k \leftarrow k + 1$ (move on to the next shelf of the same type)
13: **end if**
14: **end while**
15: **if** $k >$ NumShelfType$_j$ **then**
16: NumShelfType$_j \leftarrow$ NumShelfType$_j + 1$ (increase the number of shelves of this particular type)
17: $w(type_{j,k}) \leftarrow W - w(L_i)$
18: $H \leftarrow H + h(type_{j,k})$
19: **end if**
20: **else** $[h(L_i) > h(type_{j,k})]$
21: $j \leftarrow j + 1$ (move on to the next type)
22: **end if**
23: **end while**
24: **if** $j >$ NumTypes **then**
25: NumTypes $\leftarrow$ NumTypes $+ 1$, $k \leftarrow 1$
26: $proportion \leftarrow stdev(h(L_1), \ldots, h(L_i))$ **SDev algorithm**
27: $proportion \leftarrow (h(L_i) - h(type_{j-1,k}))$ **SDiff algorithm**
28: $h(type_{j,k}) \leftarrow proportion + h(L_i)$; $H \leftarrow H + h(type_{j,k})$
29: **end if**
30: **end while**
31: **print** H and entire packing

3.3.1 Shelf algorithms

Four new shelf algorithms are introduced in this section. The algorithms highlight three different methods of creating the *free space* in each shelf to cater for volatility in heights of rectangles to be packed later.

3.3.1.1 The shelf deviation algorithm

The newly proposed *shelf deviation* (SDev) algorithm was developed to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

In this algorithm the notion of a shelf *type* refers to a collection of shelves of equal height and the objective is to increase these fixed heights as more *types* are created, so as to accommodate variations in the heights of future rectangles to be packed (in the sense that the more shelf *types* created, the larger the fixed shelf heights per type). A $type_1$ shelf only accommodates rectangles of height $0 < h(L_i) \leq h(L_1)$ where L_1 is the first rectangle to be packed (the height of the first rectangle determines the height of the first shelf *type*). A rectangle whose height fits within this range is referred to as a $type_1$ rectangle. The heights of subsequent shelves of $type_j$ ($j \geq 2$) equals the height of the first rectangle packed on the shelf together with a certain proportion, referred to as the *shelf height increase proportion*. This proportion is computed as the *standard deviation* (stdev) of the rectangle heights already packed on all shelves, *i.e.* $h(type_j) = h(L_{i+1}) + stdev(h(L_1), \ldots, h(L_{i+1}))$. In general, $type_j$ shelves can accommodate rectangles of height $h(type_{j-1}) < h(L_i) \leq h(type_j)$, where $j \geq 2$.

Rectangles are classified according to the shelf *type* to which they belong and are packed onto the lowest shelf of that *type* with enough room. New shelf *types* are created above the top-most shelf each time the next rectangle has height exceeding the height of all existing shelf *types*. It is not necessary for two consecutive shelves to be of the same *type*, the shelf *types* may be interspersed as long as rectangles are placed onto appropriate shelf *types*. If there is insufficient horizontal space to accommodate rectangle L_{i+1}, a new shelf of the appropriate *type* is created above the top-most shelf. The pseudocode of this algorithm is given as Algorithm 28.

Example 3.17 *When applied to the rectangles in Table 1.2, a total packing height of 30 units is obtained via the SDev algorithm, as shown in Figure 3.9(a). Rectangles L_1 and L_2 are packed on the first shelf (of height 4), because they have the same height and this is a $type_1$ shelf. Rectangle L_3 does not fit on the first shelf but it is a $type_1$ rectangle since its height is less than 4, therefore the second shelf created is also a $type_1$ shelf of height 4 and rectangle L_3 is packed there. Rectangle L_4 is not a $type_1$ rectangle; hence the standard deviation of heights of rectangles L_1, L_2, L_3, L_4 is computed as 0.861 and $type_2$ shelf of height $h(L_4) + 0.861 = 5.861$ is created above the top-most shelf. This process is continued until all rectangles have been packed.* □

A discussion of the algorithmic complexity of the SDev algorithm is postponed to the end of the following subsection.

3.3.1.2 The shelf difference algorithm

The *shelf difference* (SDiff) algorithm was also developed to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

The SDiff algorithm differs from the SDev algorithm only in the way the proportion of increasing the fixed shelf heights of the different *types* is computed. In the SDiff algorithm, a $type_1$ shelf is still determined by the height of the first rectangle packed. For a subsequent shelf of $type_j$ ($j \geq 2$), instead of computing the standard deviation, the shelf height increase is taken as the difference between the height of the rectangle to be packed and the previous shelf height added to the height of the rectangle to be packed, *i.e.* $h(type_j) = (h(L_{i+1}) - h(type_{j-1})) + h(L_{i+1})$. A pseudocode listing for this algorithm is also given as Algorithm 28.

Example 3.18 *A total packing height of 32 units is obtained when the SDiff algorithm is applied to the example instance in Table 1.2, as shown in Figure 3.9(b). Shelves 1 and 2 are* $type_1$ *shelves of height 4 and are packed in a similar fashion as explained in Example 3.17. The height of rectangle* L_4 *exceeds 4; therefore a new shelf type of height* $5 + (5 - 4) = 6$ *is created and rectangle* L_4 *is packed there, left justified. This procedure is continued until all rectangles have been packed.* ◙

Suppose the rectangles in the list all have the same height, then this means only one shelf type is created. The algorithm reduces to the one dimensional first fit bin packing algorithm which has worst-case time complexity of $O(n \log n)$. If, on the other hand, all the rectangles have varying heights, then each rectangle will be packed on a new shelf type with a time complexity of $O(n)$. Consequently, the worst-case time complexity of the SDev and SDiff algorithms is $O(n \log n)$.

3.3.1.3 The shelf average algorithm

The *shelf average* (SAve) algorithm is another new shelf algorithm designed to use the history of rectangles packed, namely the average height of rectangles already packed, to create *free space* within each shelf. It was also developed to solve problems of the form

2D	R	SP	On	MiS	0,0,0,1

.

Unlike in the SDev and SDiff algorithms where the first rectangle packed on the first shelf determines the height of the first shelf created, in the SAve algorithm the height of the first shelf is determined by the tallest rectangle packed on the shelf. A subsequent rectangle is then packed on the lowest shelf where it fits both height-wise and width-wise. If a rectangle fits height-wise, but not width-wise, on the lowest shelf, a new shelf of the same height as the height of the lowest admissible shelf, is created above the top-most shelf. If, on the other hand a rectangle does not fit both height-wise and width-wise in any of the existing shelves, a new shelf height is computed as the average height of rectangles already packed along with that of the rectangle to be packed. If the height of the rectangle to be packed exceeds the shelf height computed, then the shelf height is increased by an integral factor of the standard deviation ($1 \times stdev, 2 \times stdev, 3 \times stdev$) of the rectangle heights already packed until a point is reached where the shelf height is

sufficient to accommodate the rectangle. The algorithm is given in pseudocode form as Algorithm 29.

Example 3.19 *When the SAve algorithm is applied to the rectangles in Table 1.2, a total packing height of* 24.38 *units is obtained, as shown in Figure 3.9(c). Rectangles* L_1 *and* L_2 *are packed on the first shelf (of height* 4*), since both rectangles packed have equal height (*4 *units). Rectangle* L_3 *fits on the first shelf height-wise, but not width-wise; therefore a second shelf (of height* 4 *units) is created. Rectangle* L_4 *does not fit on the existing shelves; therefore the average height of rectangles* L_1, L_2, L_3, L_4 *is computed as* 4*. But* $h(L_4) > 4$*; hence the standard deviation (stdev) of the heights of the four rectangles is computed as* 0.816*. Adding the standard deviation to the average height* $(1 \times stdev + 4 = 4.82)$*, the result is still less than* $h(L_4) = 5$*. The factor of the standard deviation is increased* $(2 \times stdev + 4 = 5.63)$ *and this becomes the new shelf height since* $h(L_4) \leq 5.63$*. Similar computations are carried out when packing rectangle* L_5*. Rectangle* L_6 *is packed on the lowest shelf where it fits both height-wise and width-wise which is the second shelf. The process is continued until all rectangles have been packed.* □

Algorithm 29 The shelf average (SAve) and shelf average2 (SAve2) algorithms

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.
Output: The height H and the entire packing.

1: $i \leftarrow 0$; $H \leftarrow 0$; shelf $\leftarrow 0$; ShelfNum $\leftarrow 1$
2: Pack the first shelf with rectangles until there are no more rectangles or there is insufficient room to pack a rectangle; $i \leftarrow i - 1$
3: **while** there is an unpacked rectangle **do**
4: $i \leftarrow i + 1$; shelf $\leftarrow 0$; pack $\leftarrow False$; factor $\leftarrow 1$
5: **while** shelf $+ 1 \leq$ ShelfNum **or** pack $\leftarrow False$ **do**
6: shelf $\leftarrow$ shelf $+ 1$
7: **if** $w(L_i) \leq W - w$(shelf) **and** $h(L_i) \leq h$(shelf) **then**
8: w(shelf) $\leftarrow w$(shelf) $+ w(L_i)$; pack $\leftarrow True$
9: **else** $[w(L_i) > W - w$(shelf) **and** $h(L_i) \leq h$(shelf)$]$
10: shelf $\leftarrow$ ShelfNum $+ 1$; ShelfNum $\leftarrow$ ShelfNum $+ 1$; $H \leftarrow H + h$(shelf); w(shelf) $\leftarrow w$(shelf) $+ w(L_i)$; pack $\leftarrow True$ **[SAve algorithm]**
11: **end if**
12: **end while**
13: **if** pack $\leftarrow False$ **then**
14: compute average height (AH); factor $\leftarrow 1$
15: shelf $\leftarrow$ ShelfNum $+ 1$; h(shelf) $\leftarrow$ AH; ShelfNum $\leftarrow$ ShelfNum $+ 1$
16: **while** $h(L_i) > h$(shelf) **do**
17: h(shelf) $\leftarrow$ AH $+$ factor $\times$ *stdev*
18: factor $\leftarrow$ factor $+ 1$
19: **end while**
20: $H \leftarrow H + h$(shelf); w(shelf) $\leftarrow w$(shelf) $+ w(L_i)$; pack $\leftarrow True$
21: **end if**
22: **end while**
23: **print** H and entire packing

A discussion of the algorithmic complexity of the SAve algorithm is postponed to the end of the following subsection.

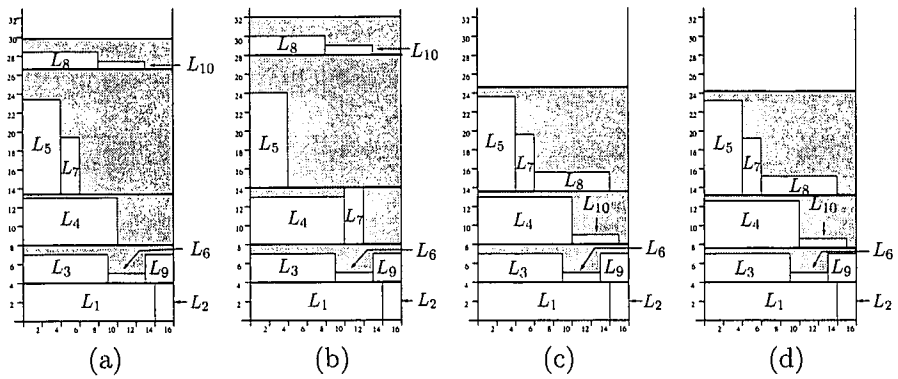

Figure 3.9: *Comparison of packing heights produced by the new on-line heuristics described in §3.3, when applied to the rectangles in Table 1.2: (a) the SDev algorithm packing, (b) the SDiff algorithm packing, (c) the SAve algorithm packing and (d) the SAve2 algorithm packing.*

3.3.1.4 The shelf average2 algorithm

The *shelf average2* (SAve2) algorithm is similar to the SAve algorithm, except the condition of checking whether a rectangle fits height-wise and not widthwise is not taken into consideration when creating a new shelf. Analogously to the SAve algorithm, the SAve2 algorithm also solves problems of the form

2D	R	SP	On	MiS	0,0,0,1

approximately. The height of the first shelf is still determined by the height of the tallest rectangle packed. If a rectangle does not fit on any of the existing shelves, then a new shelf is computed in a similar fashion to the SAve algorithm. The pseudocode listing of the algorithm is also given in Algorithm 29.

Example 3.20 *A total packing height of* 24.04 *units is obtained when the SAve2 algorithm is applied to the rectangles in Table 1.2, as shown in Figure 3.9(d). Rectangles* L_1 *and* L_2 *are packed on the first shelf and since they have equal heights, the height of the first shelf is* 4 *units. Rectangle* L_3 *does not fit on the first shelf, therefore the average height of rectangles* L_1, L_2, L_3 *is computed as* 3.66 *and this becomes the height of the second level, since* $h(L_3) < 3.66$*. Rectangles* L_4 *and* L_5 *are packed in exactly the same fashion as explained in Example 3.19. The process is continued until all rectangles have been packed.* □

The while loop spanning lines 2–19 of Algorithm 29 is executed exactly $n + 1$ times and therefore has a time complexity of $O(n)$. An analogous reasoning to the SDev and SDiff complexity analysis is used for the SAve and SAve2 algorithms. If the rectangles in the list have the same height, then the algorithm reduces to the one dimensional first fit bin packing algorithm, which has a worst-case time complexity of $O(n \log n)$. Hence the overall worst-case time complexity of both the SAve and SAve2 algorithms is $O(n \log n)$.

3.3.2 The online fit algorithm

The *online fit* (OF) algorithm was developed to solve problems of the form

2D	R	SP	On	MiS	0,0,0,0

.

It is a plane algorithm that uses the idea originally introduced by Burke *et al.* [18] (see §2.1.2.2) of using a linear array with number of elements equal to the width of the strip. Each entry represents the height of the rectangles packed at the x–coordinate of the array. There are three key concepts used in locating a space to pack a rectangle, namely the use of *empty areas*, *available widths* and *packing at the top*, and these are described below in order of preference.

Empty areas

A rectangle may be packed in an empty area and this is defined as the area formed when some rectangle L_i is placed on top of another rectangle L_j and $w(L_i) > w(L_j)$, as illustrated in Figure 3.10(a), the empty area then has width $w(L_i) - w(L_j)$ and height $h(L_j)$. An empty area may be fully or partially filled and a list of all active empty areas is maintained because any of the rectangles still to be packed may fit into one of the empty areas. A rectangle is packed into the lowest empty area with sufficient space. Once an empty area is filled completely, it becomes inactive and it is removed from the list. When an empty area is partially filled (see Figure 3.10(b)), two new empty areas are created each time and the list of empty areas is updated.

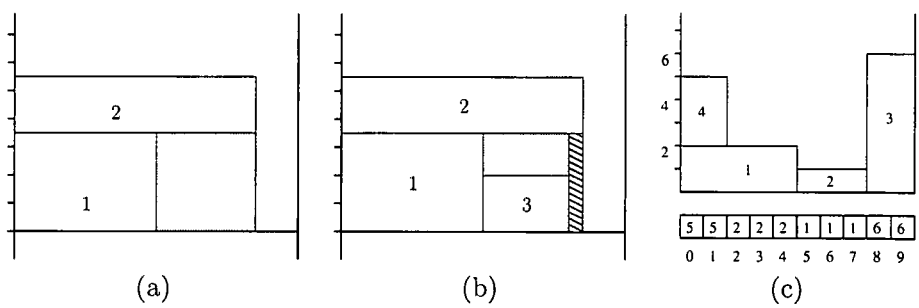

Figure 3.10: *Illustration of some key elements in the OF algorithm; (a) formation on an empty area, indicated by the shaded region, (b) a partially filled empty area generates two new empty areas, indicated by the different shadings, and (c) computation of possible widths from the linear array.*

Finding available widths

If a rectangle does not fit into any of the empty areas the linear array is searched for the lowest sufficient space. It is first searched for similar consecutive entries (they give the available width for packing) and if none of them have enough room, then the array is searched for consecutive decreasing entries. For example, in Figure 3.10(c), a rectangle of width 7 units may be packed from $x = 0$ to $x = 6$, since the entries in the array are decreasing from values of 5 to 1. The number of consecutive decreasing entries gives the

maximum width that may be accommodated. Packing a rectangle in this way guarantees the formation of empty areas.

Packing at the top

If both of the two options mentioned above fail, then a rectangle is placed at the top of the packing at a height given by the largest entry in the linear array, $x = j$ (say). Provided that there is enough room, if the space from $x = 0$ to $x = j$ is smaller (larger, respectively) than that from $x = j$ to the right-hand boundary of the strip, the rectangle is placed left justified from $x = 0$ ($x = j$, respectively). Packing at the top also guarantees the formation of empty areas.

The array elements are updated once a rectangle has been packed and the largest entry in the linear array gives the total height of the packing. The algorithm is given in pseudocode form as Algorithm 30.

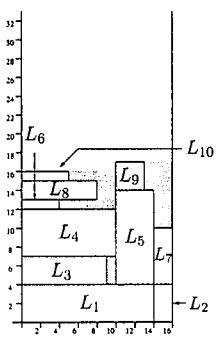

Figure 3.11: *Packing height produced by the OF algorithm when applied to the rectangles in Table 1.2.*

Example 3.21 *When the OF algorithm is applied to the rectangles in Table 1.2, a total packing height of 17 units is obtained, as shown in Figure 3.11. Rectangle L_1 is packed left justified and the array elements $x = 0$ to $x = 13$ are updated from 0 to 4. Since there are no empty areas formed, the linear array is searched for the lowest available width with enough room to pack the next rectangle. Rectangle L_2 is packed from $x = 14$ to $x = 15$, since the lowest available width of 2 units is sufficient and array elements are updated accordingly. Rectangle L_3 is packed left justified from $x = 0$ to $x = 9$, because the entire width of the strip is available for packing. The width of rectangle L_4 exceeds the available width given by similar consecutive entries in the linear array. However, there is sufficient space when considering decreasing consecutive entries and, as such, rectangle L_4 is packed on top of rectangle L_3. Since $w(L_4) > w(L_3)$, the first empty area of width 1 unit and height 3 units is created. Rectangle L_5 does not fit into the empty area; therefore the linear array is searched for equal consecutive entries and rectangle L_5 is packed from $x = 10$ to $x = 13$. This process is continued until all rectangles have been packed.* ∎

The while-loop spanning lines 2–12 in Algorithm 30 is executed exactly $n + 1$ times and thus has time complexity $O(n)$. The list of empty areas maintained varies but does not

Algorithm 30 The online fit (OF) algorithm

Description: Packing a list of rectangles without prior knowledge about the ordering, to obtain minimum strip height.
Input: The list $\mathcal{L}$ of dimensions $\langle w(L_i) \times h(L_i) \rangle$ of the rectangles and the strip width W.
Output: The height H and entire packing.

```
1: i ← 0; H ← 0
2: Create a linear array whose number of elements are equal to W
3: while there is an unpacked rectangle do
4:    i ← i + 1
5:    Search the list of empty areas for sufficient space to pack L_i
6:    if w(EmptyArea) ≥ w(L_i) and h(EmptyArea) ≥ h(L_i) then
7:       pack rectangle in empty area
8:    else
9:       if rectangle does not fit in any of the empty areas then
10:         search the linear array for available packing width
11:      else [there is insufficient space in linear array]
12:         pack rectangle on top left justified from the index leading to smaller space
13:      end if
14:   end if
15: end while
16: H ← highest entry in the linear array
17: print H and entire packing
```

exceed n. Searching the linear array has a constant time complexity. Therefore the overall worst-case time complexity of the OF algorithm is $O(n)$.

3.4 Chapter summary

In this chapter thirteen heuristics from the literature for solving *on-line* packing problems (approximately) were described in §3.1, grouped into level (§3.1.1) and shelf (§3.1.2) algorithms. Heuristics for the two special cases of allowing for modification of rectangular shapes, as well as obeying the *tetris* constraint were also described (§3.1.3). Six possible modifications to some of these heuristics were additionally proposed in §3.2. Finally, five new heuristics were introduced in §3.3, the first four (namely the SDev, SDiff, SAve and SAve2 algorithms) being shelf algorithms, while the fifth (the OF algorithm) is a plane algorithm. In the SDev, SDiff and SAve algorithms different methods were utilised for determining the proportion of increase of shelf heights. The OF algorithm on the other hand, packs rectangles in the plane by following specific rules that seek to find the lowest available location with sufficient space to pack a rectangle. Analogously to the previous chapter, the algorithms were applied to a single set of rectangles to demonstrate the various packing patterns produced by the algorithms.

Chapter 4

Exact Algorithms

An exact algorithm for optimisation problems, as defined by Murty [95], is an algorithm that is guaranteed to find an optimal solution if one exists, within a reasonable time. However, real world applications usually lead to large scale models and exact approaches, such as the branch-and-bound, whose time requirement typically grows exponentially with the size of the problem instance may lead to high computation costs. In §4.1 an integer linear programming (ILP) model capable of solving the strip packing problem exactly under the additional constraint that rectangles may only be packed on levels is described. This is followed in §4.2 by a description of an exact algorithm employing a branch and bound technique to determine an optimal packing height without the restriction of packing on levels.

4.1 The Lodi Algorithm

Lodi *et al.* [80] developed a mathematical model for the two dimensional bin packing and strip packing problems. However, the main focus here will be on the latter, since the former falls outside the scope of this dissertation. It was developed for strip packing problems of the form

2D	R	SP	Off	MiS	0,0,0,1

.

In the model, the rectangles are packed on levels (as defined in §1.3) and, without loss of generality, all input data are assumed to be positive integers. The model formulation is based on the rectangles being sorted according to non-increasing height ($h(L_1) \geq h(L_2) \geq \ldots \geq h(L_n)$); the first, leftmost rectangle placed on a level is the tallest and this automatically translates into the first level created being the tallest level in the strip. The first rectangle placed on a level is said to *initialize* the level and it is important because it contributes to the total height of the packing. For a set of n rectangles to be packed, it is assumed that potentially n levels may be created, each associated with rectangle L_i initialising it and having height $h(L_i)$. The variable

$$y_i = \begin{cases} 1 & \text{if rectangle } L_i \text{ initializes level } i \\ 0 & \text{otherwise} \end{cases}$$

is used to indicate such a rectangle ($i = 1, \ldots, n$). Due to the sorting of rectangles, any rectangle L_j packed on an initialised level must have $j > i$, where L_i is the first rectangle

packed on that level. This property is represented by the decision variables

$$x_{ij} = \begin{cases} 1 & \text{if rectangle } L_j \text{ is placed on level } i \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1 \ldots, n-1; j > i$. The objective is to

$$\begin{aligned} \text{minimise} \quad & \sum_{i=1}^{n} h(L_i) y_i, \\ \text{subject to} \quad & \sum_{i=1}^{j-1} x_{ij} + y_j = 1, && j = 1, \ldots, n, \\ & \sum_{j=i+1}^{n} w(L_j) x_{ij} \leq (W - w(L_i)) y_i, && i = 1, \ldots, n-1, \\ & y_i \in \{0, 1\}, && i = 1, \ldots, n, \\ & x_{ij} \in \{0, 1\}, && i = 1, \ldots, n-1;\ j > 1. \end{aligned} \tag{4.1}$$

The objective function measures the total packing height by adding the heights of the rectangles initialising each level. Constraint 1 ensures that rectangle L_i is packed exactly once, either by initialing a level or as a subsequent rectangle in an initialised level. This is followed by the width constraint which ensures that packing is confined to within the strip.

This mathematical model is flexible in that it can incorporate other constraints such as allowing rectangles to be rotated by 90°. This may be achieved by introducing for each rectangle L_j $(j = 1, \ldots, n)$ its rotated version L_{n+j} with $w(L_{n+j}) = h(L_j)$ and $h(L_{n+j}) = w(L_j)$. A total of $2n$ rectangles is created which are then sorted and renumbered according to non-increasing height. Between a rectangle L_j and its rotated version L_{ρ_j}, exactly one rectangle must be packed—this may be achieved by modifying the first constraint such that

$$\sum_{i=1}^{j-1} x_{ij} + y_j + \sum_{i=1}^{\rho_j - 1} x_{i\rho_j} + y_{\rho_j} = 1, \quad j = 1, \ldots, 2n;\ j < \rho_j, \tag{4.2}$$

and in the other constraints to replace n by $2n$.

Another constraint that may be added to the model is when there is a limit σ to the maximum number of rectangles that may be packed per level. To incorporate this limit the additional constraint

$$y_i + \sum_{j=i+1}^{n} x_{ij} \leq \sigma, \quad i = 1, \ldots, n - \sigma \tag{4.3}$$

may be added. The model was tested against known and improved lower bounds. By introducing LP relaxations, Lodi *et al.* [80] were able to determine lower bounds (LB)

that dominate the known *area bounds* (LB_a)

$$LB_a = \left\lceil \frac{\sum_{j=1}^{n} w_j h_j}{W} \right\rceil . \tag{4.4}$$

The first improved lower bound is referred to as the *continuous bound* (LB_c) and it is obtained by relaxing the integrality constraints such that the third and fourth constraint sets in (4.1) become $\{y_i, x_{ij}\} \in [0, 1]$. Suppose Z_c is a solution to this LP relaxation, then $LB_c = \lceil Z_c \rceil$ and $LB_c \geq LB_a$. For a proof of this inequality the reader is referred to [80, page 6].

The second relaxation introduced involves cutting rectangles vertically into slices of integer width resulting in better *combinatorial bounds* (LB_{cut}). The algorithm employed to perform this relaxation is very simple and involves sorting the rectangles according to non-increasing height. The first level is initialised with rectangle L_1 and subsequent rectangles are packed until a rectangle L_i is encountered which does not fit on the level. Such a rectangle is then split into 2 segments with the first segment having the width equal to the residual horizontal space ($\zeta = W - \sum_{j=1}^{i-1} w(L_j)$) on the level while the second segment has width equal to $w(L_i) - \zeta$ and this initialises the second level. The steps are repeated until all rectangles are placed on s levels created. If H_i represents the height of the i^{th} level such that $H_i \geq H_{i+1}$ for all $i = 1, \ldots, s-1$, then

$$LB_{cut} = \sum_{i=1}^{s} H_i \text{ and } LB_{cut} \geq LB_c. \tag{4.5}$$

The reader is referred to [80, pages 7–8] for a proof of (4.5). The efficiency of the model was tested against these lower bounds using benchmark data generated in [14] and [85] with a time-out limit of 300 CPU seconds. Results indicated that in practice the model is very effective for level packing problems.

Example 4.1 *The rectangles in Table 1.2 are ordered according to non-increasing height* $\{L_5, L_7, L_4, L_2, L_1, L_9, L_3, L_8, L_{10}, L_6\}$. *The area bound is given by* $LB_a = \lceil 223/16 \rceil = 14$ *while, after relaxing the constraints, the continuous bound is given by* $LB_c = \lceil 17.22 \rceil = 18$. *In determining* LB_{cut}, *rectangles* L_5, L_7, L_4 *are fully packed on the first level while rectangles* L_2 *and* L_1 *are also fully packed on the second level, since the sum of their widths is equal to the strip width. Rectangles* L_9 *and* L_3 *are packed on the third level with a residual horizontal space of 4 units. Rectangle* L_8 *is then cut vertically and half of its slice is packed on the third level while the other half initialised the fourth level, since it has a width of 8 units. The remaining rectangles* L_{10} *and* L_6 *are also packed on this level. The resulting packing height is* $LB_{cut} = 10 + 4 + 3 + 2 = 19$. *When applied to these rectangles, the Lodi algorithm achieves a total packing height of 19 units, as shown in Figure 4.1(a).* ∎

4.2 The Martello Algorithm

Martello *et al.* [86] considered solving a packing problem of the form

2D	R	SP	Off	MiSH	0,0,0,0

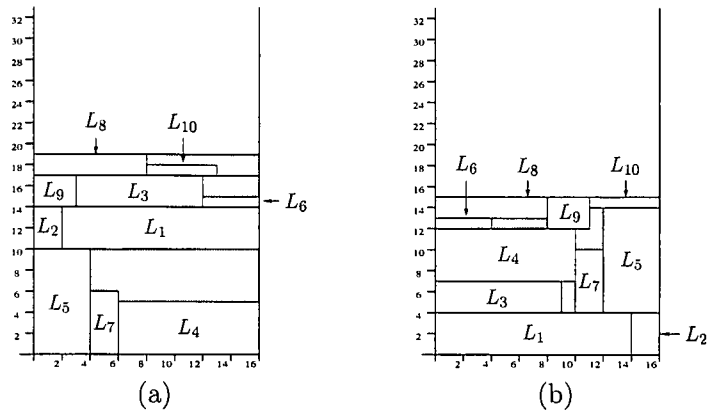

Figure 4.1: *Packings achieved by exact algorithms when applied to the rectangles in Table 1.2: (a) the Lodi algorithm and (b) the Martello algorithm.*

using an exact approach. They introduced a new relaxation producing good lower bounds and which may be solved as a 1D bin packing problem. A branch-and-bound algorithm may be employed with some heuristics being applied either at the root node or at the descendant nodes to improve the solution. Without loss of generality, it is again assumed that all input data are positive integers.

1CBP relaxation

Consider a list $\mathcal{L} = \{L_1, \ldots, L_n\}$ of n rectangles to be packed into a 2D strip of width W. The proposed relaxation technique involves "cutting" each rectangle $L_j \in \mathcal{L}$ into $h(L_j)$ unit height slices of width $w(L_j)$. The relaxed instance may be solved as a 1D bin packing problem with bin capacity W. Suppose $\mathcal{B}_1, \mathcal{B}_2, \ldots$ are 1D bins in the resulting optimal solution, then the corresponding solution to the relaxed instance packs rectangles in bin $\mathcal{B}_i$ at height $i-1$ in the strip. To improve the quality of the relaxation, it is imposed that for each rectangle L_j, the $h(L_j)$ unit height slices derived from it, be packed into $h(L_j)$ contiguous one–dimensional bins. This is referred to as the *one dimension contiguous bin-packing* (1CBP) problem. A feasible solution to the 1CBP only specifies the bin where the first unit height slice of rectangle L_j is packed. This is because the remaining slices will be packed in the subsequent $h(L_j) - 1$ bins.

Branch and bound algorithm for 1CBP

A branch-and-bound algorithm developed for the 1CBP generates at most n levels. The bins are numbered such that bin 1 corresponds to packing at the bottom of the strip. Let C_i and z denote the current content of bin i and current solution value (*i.e.* the number of initialised bins) respectively. At the root node of the branching tree for the 1CBP, an attempt is made to determine an optimal packing for a subset of rectangles, thereby reducing the size of the problem instance (since there will be fewer rectangles in the branch-and-bound algorithm procedure). The rectangles are first sorted according to non–increasing width and the reduction is attempted only if $w(L_1) > W/2$.

Let $G = \{L_j : w(L_j) = w(L_1)\}$ and $S = \{L_j : w(L_j) \leq W - w(L_1)\}$, then rectangles in G cannot be packed side by side. Only rectangles in S may be packed alongside the rectangles in G. An optimal packing for the original instance exists, provided that a feasible packing of all the rectangles of $G \cup S$ into a strip of height $\sum_{L_j \in G} h(L_j)$ exists, and so the items of $G \cup S$ may be removed. The existence of the required packing is heuristically tested as follows: if $\sum_{L_j \in S} w(L_j)h(L_j) \leq \sum_{L_j \in G}(W - w(L_j))h(L_j)$, the rectangles of G are packed one above the other, left justified and starting from this partial solution various heuristics (to be discussed later) are applied to the 1CBP instance brought about by the rectangles of S. If no solution of height $\sum_{L_j \in G} h(L_j)$ is found, then the first rectangle L_f such that $w(L_1) > w(L_f) > W/2$ (if any) is found. The rectangles $\{L_j : w(L_j) = w(L_f)\}$ are added to G and those of $\{L_j : W - w(L_f) \geq w(L_j) > W - w(L_1)$ are added to S.

Within the branch-and-bound procedure, a number of heuristics are employed, as mentioned above. They may be used either at the root node to obtain an initial solution or at the descendant nodes to improve the current solution. Some of the heuristics for 1CBP suggested include the (1) First Fit (FF), (2) Best Fit (BF) and (3) Worst Fit (WF) algorithms. The $h(L_j)$ slices of rectangle L_j are packed into $h(L_j)$ consecutive bins. If more than one set of consecutive bins exists then different rules in selecting the appropriate sets are used depending on the heuristic employed. If no set of consecutive bins exists, then assuming that bin i is the largest bin number for which $C_i + w(L_j) > W$, a new set of bins $i + 1, \ldots, i + h(L_j)$ is generated. The rules of the heuristics are as follows:

FF: Select the first set of bins;

BF: Select the set of bins already containing the maximum total contents;

WF: Select the set of bins already containing the minimum total contents.

However, additional rules are required, since this case deals with packing rectangular slices into a set of bins as opposed to the usual problem of packing one rectangle into a bin at a time. There are at most $n!$ ways of sorting the rectangles, each of them bringing out a potentially different solution. Martello *et al.* [86] tested a number of sorting criteria and the best ones are reportedly:

- decreasing height, breaking ties by decreasing width;
- decreasing area, breaking ties by decreasing width;
- decreasing area, breaking ties by decreasing height;
- decreasing height over width ratio, breaking ties by decreasing height.

This results in 12 heuristics which are all executed at the root node and the one producing the best solution is executed at each decision node.

The branch decision tree is searched according to a depth-first strategy. At level L_j, the first slice (bottom) of rectangle L_j is considered first. Rectangle L_j is assigned, in turn, to each bin of a subset Q_j for which: (1) a feasible packing of all the slices of rectangle L_j is allowed, and (2) a partial solution of value less than the incumbent is produced *i.e.* $Q_j \subseteq \{k \leq z - h(L_j) : C_i + w(L_j) \leq W$ for $i = k, \ldots, k + h(L_j) - 1\}$. The second condition is illustrated in Example 4.2.

Example 4.2 *Assuming that $z = 8$, $h(L_j) = 5$ and the slices of rectangle L_j fits into all the bins, packing the rectangle into bin 4 does not lead to an improvement of the current solution since the five slices would be packed into bins 4, 5, 6, 7, 8. In this way z is not improved. The first slice should be packed into any one of bins 1, 2 or 3 ($3 \leq 8 - 5$) as these will lead to an improvement of the current solution.* □

Two methods are suggested for reducing the number of descendants of the current node, *i.e.* to obtain a small value of $|Q_j|$. Let $y(L_i)$ be the bin where, for each rectangle L_i $(i = 1, \ldots, h(L_{j-1}))$ already packed, the last (top) slice of rectangle L_i is currently packed. In Q_j only bins k such that k can be obtained as a combination of one value from $\{1\} \cup \{y(L_i) + 1 : 1 \leq i < j\}$ and any number of values from $\{h(L_i) : i > j\}$ are considered. A further reduction of the number of nodes may be achieved by eliminating branches that lead to equivalent patterns, *i.e.* if $h(L_j) = h(L_{j-1})$ and $w(L_j) = w(L_{j-1})$, then all bins below the bin currently accommodating the bottom slice of rectangle L_{j-1} (*i.e.* all bins k with $k \leq y(L_{j-1}) - h(L_{j-1})$) are removed.

In any branch-and-bound algorithm, a bound is required that indicates the cost of the solution in a given subset. Martello *et al.* [86] used a lower bound LB_2 which is stated without proof as: Let α be any integer in [1, W/2], and define

$$\mathcal{L}_1 = \{L_j \in \mathcal{L} : w(L_j) > W - \alpha\}, \tag{4.6}$$

$$\mathcal{L}_2 = \{L_j \in \mathcal{L} : W - \alpha \geq w(L_j) > W/2\} \text{ and} \tag{4.7}$$

$$\mathcal{L}_3 = \{L_j \in \mathcal{L} : W/2 \geq w(L_j) \geq \alpha\}. \tag{4.8}$$

Also, let

$$LB(\alpha) = \sum_{L_j \in \mathcal{L}_1 \cup \mathcal{L}_2} h(L_j) + \max\left(0, \left\lceil \frac{\sum_{L_j \in \mathcal{L}_3} w(L_j)h(L_j) - (\sum_{L_j \in \mathcal{L}_2} (W - w(L_j))h(L_j))}{W} \right\rceil\right). \tag{4.9}$$

Then

$$LB_2 = \max_{1 \leq \alpha \leq W/2}\{LB(\alpha)\} \tag{4.10}$$

is a valid lower bound for 2D strip packing.

At each decision node, to determine whether the node should be explored any further to possibly bring about an improvement in the current solution, the lower bound LB_2 is computed as follows. Let L_j be the last assigned rectangle and let $\overline{z}$ be the highest bin number containing a slice. For $v = 1, \ldots, W$, let $\overline{h}_v = |\{k \leq \overline{z} : C_k = v\}|$ be the number of bins containing the current content of value v. A dummy rectangle of width v and height $\overline{h}_v$ is introduced for each $\overline{h}_v > 0$. The unassigned rectangles L_q $(q > j)$ along with the dummy rectangles bring about a 2D strip packing instance which allows for the computation of LB_2. If $LB_2 > z$, then the node is fathomed (*i.e.* the node is not explored any further).

If the branch-and-bound procedure fails to determine an optimal solution within a fixed time limit, a looser relaxation of the 2D strip packing is employed. Each rectangle L_j

is "cut" into $\lfloor h(L_j)/2 \rfloor$ slices of height 2 (disregarding the top unit height slice if $h(L_j)$ is odd) or into $\lfloor h(L_j)/3 \rfloor$ slices of height 3 and so on, until a solvable 1CBP instance is produced. The steps followed to solve the 1CBP are shown in Example 4.3.

	L_1	L_2	L_3	L_4
Width	4	3	2	1
Height	2	2	3	4

Table 4.1: *Rectangle dimensions used in Example 4.3*

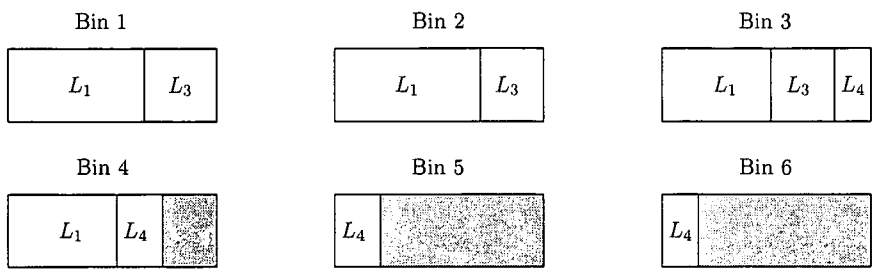

Figure 4.2: *The First Fit heuristic applied to the 1CBP producing an initial solution over 6 bins.*

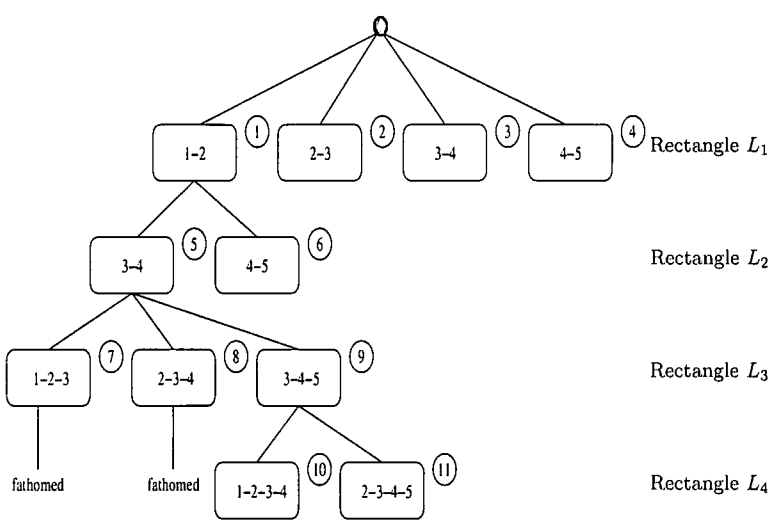

Figure 4.3: *Branch decision tree for the exact approach towards solving the 1CBP which has generated 4 levels.*

Example 4.3 *Consider the list $\mathcal{L}$ of rectangles with dimensions listed in Table 4.1 and suppose a packing of the rectangles into a strip of width $W = 6$ is required.*

1. *Reduction of the problem instance.*

Because $w(L_1) > W/2$, *a reduction may be attempted with the sets* $G = \{L_1\}$ and $S = \{L_3, L_4\}$. *Hence* $\sum_{L_j \in S} w(L_j)h(L_j) = 10 > \sum_{L_j \in B}(W - w(L_j))h(L_j) = 4$. *It is therefore not possible to find an optimal solution for rectangles in* $G \cup S$.

2. *Applying the First Fit heuristic at the root node.*

   The first fit heuristic is used at the root node to obtain an initial solution with an objective function value $z = 6$ *to the 1CBP, as shown in Figure 4.2. The slices are packed into the first set of contiguous bins with sufficient space.*

3. *Branch Decision Tree.*

   The branch decision tree is depicted in Figure 4.3. At the beginning, all bins are empty and rectangle L_1 *may be assigned to any pairs of bins. Four nodes are initialised, each representing a feasible allocation of rectangle* L_1 *to a set of one dimensional bins. It is clear that packing rectangle* L_1 *into bins 5–6 does not lead to an improvement of the current solution.*

4. *Calculating the lower bound* LB_2 *at each decision node.*

   Node 1: The current content of bins 1 and 2 is $C_1 = 4, C_2 = 4$. *The highest bin containing a slice is bin* $\overline{z} = 2$. *For* $v = 1, \ldots, W$, *the number of bins* k *for which* $C_k = v$ *is* $\overline{h}_4 = 2$. *A dummy rectangle* L_{v1}, *with dimensions* $w(L_{v1}) = 4$ and $h(L_{v1}) = 2$ *is created and* LB_2 *is computed on the set of unassigned rectangles* L_2, L_3, L_4 *along with the dummy rectangle* L_{v1}.

   Let $\alpha = 1$. *Then*

$$\begin{aligned} \mathcal{L}_1 &= \emptyset, \\ \mathcal{L}_2 &= L_{v1}, \\ \mathcal{L}_1 &= L_2, L_3, L_4 \text{ and} \\ LB(1) &= 2 + \max\left(0, \left\lceil \tfrac{(6+6+4)-(4)}{6} \right\rceil\right) = 4. \end{aligned} \tag{4.11}$$

   Let $\alpha = 2$. *Then*

$$\begin{aligned} \mathcal{L}_1 &= \emptyset, \\ \mathcal{L}_2 &= L_{v1}, \\ \mathcal{L}_1 &= L_2, L_3 \text{ and} \\ LB(2) &= 2 + \max\left(0, \left\lceil \tfrac{(6+6)-(4)}{6} \right\rceil\right) = 4. \end{aligned} \tag{4.12}$$

   Let $\alpha = 3$. *Then*

$$\begin{aligned} \mathcal{L}_1 &= L_{v1}, \\ \mathcal{L}_2 &= L_2, \\ \mathcal{L}_1 &= \emptyset \text{ and} \\ LB(3) &= 4 + \max\left(0, \left\lceil \tfrac{0-(6)}{6} \right\rceil\right) = 4. \end{aligned} \tag{4.13}$$

   Hence $LB_2 = 4$ *and since it is less than* z, *this node may be partitioned into nodes that improve the current solution. Nodes 2, 3 and 4 also yield* LB_2 *values which are less than* z. *The branching process therefore starts at node 1.*

5. *Assigning rectangle L_2.*

   First, the subset Q_2 is defined as $Q_2 \subseteq \{k \leq z - h(L_2) : C_k + w(L_2) \leq W\}$. Hence $Q_2 \subseteq \{3, 4\}$, because $C_1 + w(L_2) > W$ and $C_2 + w(L_2) > W$. Node 1 has two descendant nodes, assigning rectangle L_2 either to bins 3-4 (node 5) or to bins 4-5 (node 6). At node 5, the lower bound is given by $LB_2 = 4$. Hence this node is branched further.

6. *Assigning rectangle L_3.*

   Node 5 has three descendant nodes assigning rectangle L_3 either to bins 1-2-3 (node 7) or to bins 2-3-4 (node 8) or to bins 3-4-5 (node 9). The lower bound is given by $LB_2 = 2$ at node 7. Hence this node is branched further.

7. *Assigning rectangle L_4.*

   At Node 7, the current content of the bins is as follows: $C_1 = 6, C_2 = 6, C_3 = 5, C_4 = 2$. Hence $Q_4 \subseteq \{1, 2\}$, meaning that rectangle L_4 may be packed into bins 1-2-3-4 (node 10) or into bins 2-3-4-5 (node 11). However, none of these packings is feasible, because bins 1 and 2 have already been packed to capacity. This node is therefore fathomed and back-tracking to node 5 occurs. There are two further descendant nodes (nodes 8 and 9) which may be explored.

   Considering node 8, a similar situation to node 7 arises, whereby bin 2 is fully packed—hence a feasible packing of rectangle L_4 is not possible and this node is also fathomed. Back-tracking to node 5 therefore occurs. Now only node 9 has not been visited before—hence it is explored next. The current content of bins at this node is as follows: $C_1 = 4, C_2 = 4, C_3 = 5, C_4 = 5, C_5 = 2$. Therefore a feasible packing of rectangle L_4 is possible, producing two descendant nodes.

The branch-and-bound algorithm for the 1CBP in this example has produced four levels and there are two optimal solutions produced either by nodes 1, 5, 9, 10 or nodes 1, 5, 9, 11, each yielding a solution in 5 bins to the 1CBP, as shown in Figure 4.4. ∎

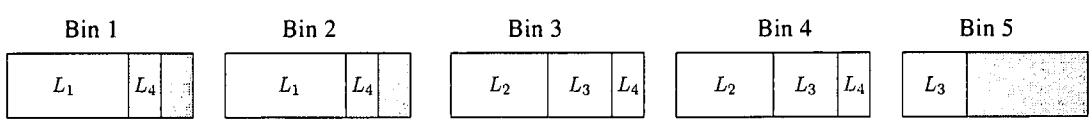

Figure 4.4: *Solution to the 1CBP in Example 4.3 utilising 5 bins.*

Branch decision tree for the 2D strip packing problem

In the branch-and-bound procedure for the 2D strip packing problem, each node represents a feasible placement of a subset $I \subset \mathcal{L}$ of rectangles into the strip. The rectangles placed into the strip define an "envelope", which is a line joining the outer-most, top edges of rectangles packed into the strip. The envelope separates the strip into two regions where it may or may not be possible to place incoming rectangles. "Corner points" are defined as the points where the slope of the envelope changes from vertical to horizontal. Nodes in the branching tree are generated by placing the bottom-left corner of each

incoming rectangle at these corner points, in all its admissible positions. For illustration purposes, the branching tree for the 2D strip packing problem is applied to the rectangles in Table 4.1.

The same reduction procedure as described for the 1CBP is applied at the root node, but with a different set of heuristics. The heuristics may also be implemented at the descendant nodes to improve the current solution. Martello *et al.* [86] implemented (1) the Next-Fit Decreasing Height (NFDH) algorithm, (2) the First Fit Decreasing Height (FFDH) algorithm, (3) the Best Fit Decreasing Height (BFDH) algorithm, (4) AlgorithmJOIN, (5) the Bottom-Left (BL) algorithm and (6) AlgorithmBUILD. The first four heuristics initially require the rectangles to be sorted according to non-increasing height and they are packed into the strip such that they form levels. The NFDH algorithm, the FFDH algorithm, the BFDH algorithm and AlgorithmJOIN were discussed fully in Chapter 2. In the BL algorithm, each rectangle is packed in the lowest possible position, left-justified (see Baker, *et al.* [4]). AlgorithmBUILD is used to combine the slices from a solution to the 1CBP—thus generating a solution to the original strip packing instance.

To avoid a situation of generating identical nodes, Martello *et al.* [86] suggested that for each potential node corresponding to the placement of rectangle L_j at corner point (say), envelopes generated by rectangle sets $\{L_j\} \cup (I \setminus \{L_i\})$[1] be determined for all $L_i \in I$, where I represents an instance of rectangles already packed. If there exists a rectangle L_i currently placed in a corner point d (say), for which the envelope corresponding to $\{L_j\} \cup (I \setminus \{L_i\})$ includes d among its corner points, then the current potential node would produce a pattern identical to that produced by exchanging the order in which rectangles L_i and L_j are considered in the branching process. The node is generated only if $j > i$.

Example 4.4 *(Continuation of Example 4.3) The solution comprising five bins obtained in Example 4.3 for the 1CBP is used as the overall lower bound for 2D strip packing. If, after placement of rectangle L_4 at any node, a strip height equal to 5 is obtained, then that is an optimal solution. At the root node, there is only one corner point (0,0), where any of the four rectangles may be placed—hence there are four descendant nodes; rectangle L_1 at (0,0) (node 1), rectangle L_2 at (0,0) (node 2), rectangle L_3 at (0,0) (node 3) and rectangle L_4 at (0,0) (node 4). After a node is created by branching, the lower bound of the total strip height (H) associated with the node is computed. Placing rectangle L_1 at (0,0) with lower bound $H \geq 2$, two corner points (4,0) and (0,2) are generated. Figure 4.5 shows the expansion of node 1, where the black dots indicate the corner points and the broken line represents the envelope.*

1. *Placing rectangle L_2.*

   At level 2, there are six descendant nodes formed by placing rectangles L_2, L_3, L_4 at each of the two corner points. Rectangle L_2 may be placed at (4,0) (node 5) or (0,2) (node 6). Node 5 is fathomed, since there is not enough space to pack rectangle L_2. Therefore, back-tracking to node 1, there are still five nodes to be explored. Moving on to node 6, after placing rectangle L_2 at corner point (0,2), the three corner points (4,0), (3,2) and (0,2) are generated.

[1] The notation $\{L_j\} \cup (I \setminus \{L_i\})$, as used in [86], means that rectangle L_i has not yet been packed [92].

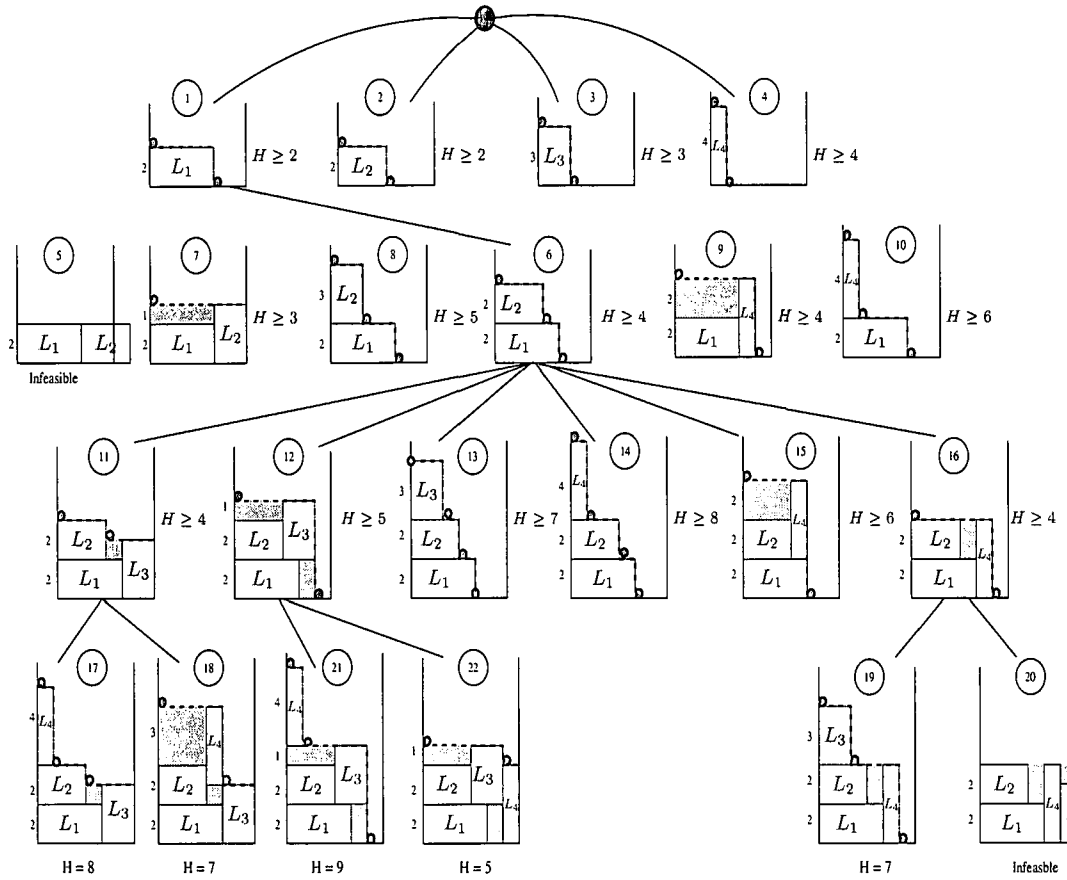

Figure 4.5: *Branch decision tree for the 2D strip packing problem where the black dots and dashed lines denote the corner points and the envelope respectively.*

2. *Placing rectangles L_3 and L_4.*

   Because there are three corner points and two rectangles to be packed, level 3 has six descendant nodes. Rectangle L_3 may be placed either at corner point (4,0) (node 11) or at (3,2) (node 12) or at (0,2) (node 13).

   Node 11: Branching on node 11, because it has the smallest bound on the strip height, two corner points (0,4) and (3,3) are generated resulting in two descendant nodes. Rectangle L_4 may be placed at (0,4) (node 17) resulting in strip height of 8 or at (3,3) (node 18) resulting in strip height of 7. Node 18 is a feasible packing and may be considered a candidate solution for which $H = 7$ and node 17 is eliminated. Back-tracking to node 6, the next node to be branched on is node 16.

   Node 16: There are two corner points (0,4) (node 19) and (5,0) (node 20) where rectangle L_3 may be placed, resulting in two descendant nodes. Node 19 is associated with a strip height of 7, while node 20 is fathomed, because rectangle L_3 cannot fit into the strip if placed at this corner point. So far the candidate solution has not been improved.

   Node 12: The next node to be branched on is node 12, since it produces a lower bound of 5 on the total strip height. Two corner points (0,5) (node 21) and (5,0)

(node 22) lead to two descendant nodes where rectangle L_4 can be placed. Node 21 results in a total strip height of 9 and since this is greater than that associated with the candidate solution, this node is eliminated. Back-tracking to node 12, there is still node 22 which has not been visited. Node 22 results in a strip height of 5.

The branching process continues in this manner until the candidate solution may not be improved—thus resulting in an optimal solution. In this example, an optimal packing height of 5 units was obtained. □

In the above example, a node with descendant nodes, such as nodes 1, 6 and 12 shown in Figure 4.5, is referred to as a *branch node*, whilst all the lines linking two nodes are referred to as *branches*. When testing the Martello algorithm on the benchmark data described in §1.6, the results will be presented in Chapter 5 and will be based on this terminology.

4.3 Chapter Summary

In this chapter two exact algorithms producing respectively guillotineable and non-guillotineable packings were described. The Lodi algorithm is a level algorithm that seeks to find the best combination of rectangles to fully fit a level [80]. The Martello algorithm, on the other hand, is a plane algorithm that determines the optimal solution by attempting to pack rectangles, in turn, at all admissible positions [86]. The two algorithms were tested on some of the benchmark data sets described in §1.6 and the results are presented later in Chapter 5.

Chapter 5

Comparison of offline algorithmic results

All the offline algorithms described in Chapter 2 were implemented using Visual Basic 6.0 [89] and a comparison of the experimental results achieved by means of the algorithms is provided in this chapter. Each algorithm was applied to the 542 benchmark data sets described in §1.6. The criteria used when comparing and ranking the algorithm performances were: the total *packing height* achieved in each test case, the *frequency* with which an algorithm achieves the smallest packing height over all test instances and the *execution time*. Ideally, the best heuristic would be able to attain the smallest packing height in the shortest possible time. However, there is typically a trade-off between total average packing height and execution time, as will be shown.

This chapter is structured such that, initially, each original heuristic in §2.1.1–§2.1.2 is compared with its proposed improvements and finally all algorithms are compared in terms of their efficiency and performance. Standard statistical analyses, such as the student's t-test, ANalyses Of VAriance (ANOVA) and chi-squared tests are carried out to test for statistically significant differences between the mean packing heights obtained by the different classes of algorithms and the frequencies with which algorithms achieved smallest packing heights. All statistical analysis results presented in this dissertation were carried out at a 5% level of significance. These statistical tools are fully described in Appendix C.

5.1 Comparison of offline level algorithms

Three basic types of analyses were carried out during the comparison of algorithmic results. Firstly, the student's t-test was used when comparing average packing heights of two algorithms while an ANOVA was used when comparing average packing heights of three or more algorithms. The results of these tests indicate whether there is any significant difference between the mean packing heights obtained by the algorithms in question. The results of these tests are shown in a table (such as Table 5.1), indicating the average packing heights obtained over the 542 data sets for each algorithm and two values, F_{value} and $F_{critical}$, in each case. Here, F_{value} represents the fraction of variance between the packing heights obtained by the algorithms and the variance of packing

heights within each algorithm, while $F_{critical}$ is the associated test statistic obtained from an F-distribution table. If the calculated F_{value} exceeds the tabulated value of $F_{critical}$, then there are significant differences between the mean packing heights obtained by the algorithms in question. The second test is the chi-squared test and this was used to test whether there are significant differences between the frequencies with which each algorithm obtained the smallest packing height. The results are also represented in tabular form (such as Table 5.2), which gives the frequencies obtained by each algorithm and respective χ^2_{df} and $\chi_{critical}$ values, where df denotes the number of degrees of freedom. If the calculated value of χ^2_{df} exceeds the tabulated value of $\chi_{critical}$, then there are significant differences between the frequencies with which the smallest packing height was obtained by the algorithms. Thirdly, a so-called aspect ratio test is applied, whereby the abilities of the algorithms to obtain good solutions are linked to average aspect ratios of the data sets. The first 24 columns of Table 5.3 represent a summary of the packing heights obtained by the level algorithms (see Chapter 2) over the 542 data sets.

5.1.1 Comparison of algorithms in the next fit class

The mean packing heights obtained by the NFDH, NFDHIW and NFDHDW algorithms (see §2.1.1.1, §2.2.1 and §2.2.2 respectively) over the 542 benchmark data sets are shown in Table 5.1. These values were used to assess and compare the relative performances amongst the three algorithms. The results of an ANOVA on the *next fit* class show that there is no significant difference between the mean packing heights obtained by the algorithms in the class at a 5% level of significance. This result is not surprising, given the similar natures of the algorithms in question.

NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	F_{val}	F_{crit}
169.634	169.946	169.662							0.0004	3.0013
			161.728	161.592	161.854				0.0003	3.0013
						161.524	161.427	161.633	0.0002	3.0013

Table 5.1: *Summary of results from the analysis of variance for the next fit, first fit and best fit classes of algorithms.*

Turning from the analysis of average packing heights to the comparison of the frequency with which the smallest packing height was obtained by the three algorithms, the results in Figure 5.1 were obtained. The first three bars in the figure represent the frequencies for the *next fit* class of algorithms. The heights of the unshaded bars in the figure represent the number of times each algorithm obtained the smallest packing height achieved by any algorithm and the heights of the shaded bars represent the number of times each algorithm was the only procedure to obtain the smallest packing height (*i.e.* the number of times the algorithm obtained the smallest packing height uniquely). The NFDHIW algorithm was able to obtain the smallest packing height more often than the other two algorithms, as may be seen in the figure.

NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	χ^2_{df}	$\chi crit$
387	381	412							1.375	5.990
			447	475	397				**7.102**	**5.990**
						446	474	406	5.285	5.990

Table 5.2: *Summary of results from the chi-squared test for the next fit, first fit and best fit classes of algorithms. The bold faced values indicate significant differences between the frequencies in that particular class.*

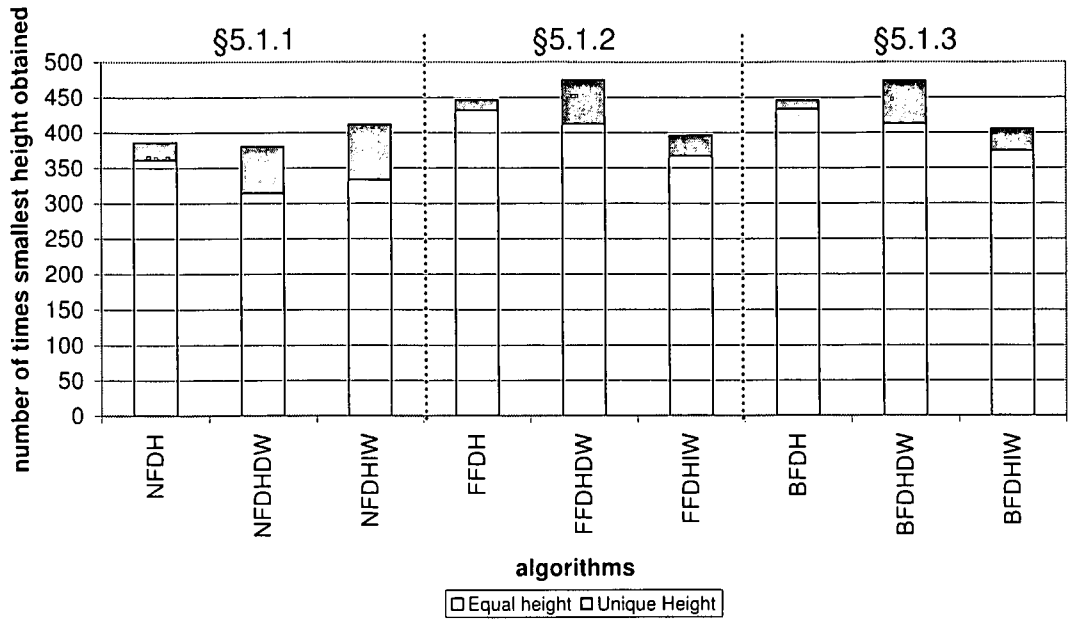

Figure 5.1: *The number of times the algorithms in the next fit, first fit and best fit classes obtained the smallest packing height.*

It is of interest to establish statistically whether there are significant differences between these frequencies. The chi-squared test was employed at a 5% level of significance in this regard and the results are shown in row 1 of Table 5.2. Because $\chi_{critical}(5.990) > \chi^2_2(1.375)$, it is concluded that statistically there appears to be no difference between the frequencies. Hence, no algorithm in the *next fit* class is statistically superior to another in terms of the frequency with which it achieved the best results, at a 5% level of significance. Because the three algorithms in the *next fit* class can never perform better than their counterparts in the *first fit* and *best fit* classes, these algorithms were excluded from all further comparisons and analyses.

5.1.2 Comparison of algorithms in the first fit class

Analogous to the analysis carried out in §5.1.1, a comparison of the relative performances of the FFDH, FFDHIW and FFDHDW algorithms (see §2.1.1.2, §2.2.3 and §2.2.4 respectively) was also based on the packing heights obtained by each algorithm. Results from an ANOVA indicate that there is no significant difference between the mean packing heights obtained by the three algorithms over all 542 benchmark data sets at a 5% significance level. This may be seen from row 2 of Table 5.1 in which the computed F_{value} (0.0003) is less than $F_{critical}$ (3.0013). Hence, in terms of the overall quality of the solution obtained, there is no statistical difference between the algorithms in the *first fit* class—all of the algorithms are expected to perform approximately equally effectively.

Moving on to the frequency analysis, it is evident from row 2 of Table 5.2 that the FFDHDW algorithm outperformed the other two algorithms in terms of the frequency with which it obtained the smallest packing height amongst the three first fit algorithms.

Author	Data Sets	# of Rectangles	OPT	NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	SF	SF_{mod}	KP	SAS	FCNR	$FCNR_{mod}$	FCNRNG	Alg.JOIN	Alg.JOIN01	Alg.JOIN00	$Alg.JOIN_{mod}$	$Alg.JOIN01_{mod}$	$Alg.JOIN00_{mod}$	Slentor	$Slentor_{mod}$	Burke	BurkeUP	BurkeDown
Bensley [9, 10]	U_1	10	-	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1086	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016	1016
	U_2	20	-	1564	1564	1564	1349	1349	1349	1349	1349	1349	1382	1735	1347	1499	1349	1349	1349	1554	1364	1552	1414	1364	1552	1630	1600	1259	1259	1259
	U_3	30	-	2101	2112	2101	1873	1873	1873	1810	1810	1810	2446	3008	1899	2077	1810	1810	1810	2023	1901	2023	2023	1901	2023	2537	2479	1803	1803	1803
	U_4	50	-	3996	4121	3995	3216	3216	3216	3216	3216	3216	4043	5445	3159	3396	3216	3216	3216	3571	3376	3826	3503	3376	3602	4321	4321	3056	3056	3056
	V_1	10	-	25	25	25	25	25	25	25	25	25	25	25	25	27	25	25	25	25	25	25	25	25	25	25	23	23	23	23
	V_2	17	-	33	33	33	33	33	33	33	33	33	33	33	36	33	33	33	33	33	33	33	33	33	33	34	34	32	32	32
	V_3	21	-	34	37	34	34	31	34	34	34	34	34	34	35	31	34	34	34	34	34	34	34	34	34	33	32	29	29	29
	V_4	7	-	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	22	22	20	20	20
	V_5	14	-	46	46	46	46	37	46	46	37	46	46	46	37	37	46	46	46	37	37	37	46	46	47	46	46	37	37	37
	V_6	15	-	38	38	42	38	38	40	38	38	40	38	38	38	35	36	36	36	40	40	40	38	38	38	39	39	35	35	35
	V_7	8	-	21	21	21	21	21	21	21	21	21	22	21	21	20	21	20	21	21	21	21	21	21	21	20	20	20	20	20
	V_8	13	-	44	44	44	38	44	38	38	44	38	44	44	38	38	38	38	38	38	38	38	44	44	44	42	42	38	38	38
	V_9	18	-	66	65	66	65	65	66	65	65	65	65	65	65	60	64	64	64	66	66	70	66	66	70	66	65	55	55	55
	V_{10}	13	-	85	85	85	85	85	85	85	85	85	85	85	93	85	85	85	85	85	85	85	86	85	86	83	83	81	81	81
	V_{11}	15	-	69	69	69	69	69	69	69	69	69	69	69	69	75	63	63	63	69	69	69	69	69	69	63	60	57	57	57
	V_{12}	22	-	109	104	109	104	104	104	104	104	104	104	104	102	91	96	96	96	104	104	104	104	104	104	96	96	87	87	87
Burke *et al.* [18]	B_1	10	40	48	48	48	46	46	46	46	46	46	48	48	46	60	46	46	46	46	46	46	46	46	46	60	46	48	48	48
	B_2	20	50	71	71	71	65	65	65	65	65	67	67	71	61	65	61	61	61	71	71	71	65	65	65	70	67	57	57	57
	B_3	30	50	68	68	68	68	68	68	68	68	68	76	76	68	63	62	57	62	68	68	68	68	68	68	65	65	53	53	53
	B_4	40	80	126	130	130	126	126	126	126	126	126	140	140	126	103	93	93	93	127	126	127	126	126	131	117	98	129	129	129
	B_5	50	100	132	127	132	119	119	125	119	119	125	120	121	122	115	111	109	109	126	126	126	126	126	126	142	131	107	107	107
	B_6	60	100	117	119	114	109	109	110	109	109	110	131	133	109	110	105	105	105	110	110	111	109	109	109	131	131	107	107	107
	B_7	70	100	164	165	164	160	160	160	160	160	160	161	138	163	120	124	122	122	161	161	161	160	160	160	138	132	116	116	116
	B_8	80	80	112	112	112	108	108	108	108	108	108	115	103	115	104	92	86	92	109	108	111	108	108	108	101	101	97	97	97
	B_9	100	150	201	209	193	181	181	181	177	177	177	190	186	177	177	162	158	162	179	179	179	178	178	178	191	183	154	154	154
	B_{10}	200	150	228	203	229	210	191	212	190	191	192	217	190	192	158	182	164	182	191	191	191	193	193	193	200	176	163	163	163
	B_{11}	300	150	195	194	185	169	169	171	169	169	171	170	173	168	159	155	156	156	171	171	171	172	172	172	181	181	153	153	153
	B_{12}	500	300	384	389	381	372	372	372	372	372	372	375	347	372	341	343	336	341	374	373	376	371	371	378	386	369	305	305	305
Christofides [22]	G_1	16	-	28	31	28	28	28	28	28	28	28	28	28	28	25	25	25	25	31	31	31	28	28	28	28	28	27	27	27
	G_2	23	-	83	83	83	83	83	83	83	83	83	83	82	84	75	73	73	73	83	83	83	83	83	84	81	81	78	78	78
	G_3	62	-	937	937	937	744	728	744	744	728	744	796	977	728	744	744	744	744	930	907	930	870	847	870	987	972	671	671	671

Table 5.3: *Summary of mean packing heights obtained by offline algorithms over the 542 data sets described in §1.6. The shaded columns represent packing heights obtained by the new heuristics and proposed modifications as part of the contribution by the author. The algorithms to the left of the double vertical line are level algorithms while the algorithms to the right of the double line are plane algorithms.*

Author	Data Sets	# of Rectangles	OPT	NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	SF	SF_{mod}	KP	SAS	FCNR	FCNRNG	$FCNR_{mod}$	AlgJOIN	$AlgJOIN_{mod}$	AlgJOIN01	$AlgJOIN01_{mod}$	AlgJOIN00	$AlgJOIN00_{mod}$	Slentor	$Slentor_{mod}$	Burke	BurkeUP	BurkeDown
Hopper *et al.* [57, 58]	C1	16/17	20	27.7	29.0	28.0	27.3	27.3	28.0	27.3	27.3	27.3	29.3	29.7	27.7	25.7	22.0	21.7	22.0	28.0	28.0	28.0	27.3	27.3	27.3	26.0	25.7	23.3	23.3	23.3
	C2	25	15	18.0	19.0	18.7	18.0	17.3	18.3	18.0	17.3	18.3	19.0	18.3	17.0	19.0	17.0	16.7	17.3	18.0	18.0	18.0	18.7	18.7	18.7	19.3	19.0	17.3	17.3	17.3
	C3	28/29	30	40.3	39.7	40.3	38.0	38.0	38.3	38.0	38.0	38.3	39.7	40.0	38.3	36.3	35.3	35.3	35.3	37.7	37.7	37.7	37.3	37.3	37.3	38.7	38.7	34.0	34.0	34.0
	C4	49	60	83.3	82.0	82.3	77.0	76.0	77.7	76.3	76.0	76.3	81.0	77.3	76.3	68.0	69.0	67.7	67.7	78.7	78.7	79.0	78.3	78.3	81.0	78.0	78.0	67.0	67.0	67.0
	C5	72/73	90	113.3	117.7	114.3	104.7	104.7	105.0	104.7	104.7	105.0	105.7	107.3	104.7	102.3	98.3	97.0	98.3	106.3	106.3	106.7	106.0	106.0	106.3	120.7	113.3	95.7	95.7	95.7
	C6	97	120	151.0	152.3	150.7	140.7	140.0	141.7	140.7	140.0	141.0	144.3	143.7	142.0	135.3	128.7	126.7	128.3	143.0	143.0	143.3	142.3	142.3	144.0	156.0	149.3	130.3	130.3	130.3
	C7	196/197	240	293.7	293.0	291.3	272.7	272.7	273.3	272.3	272.3	273.3	277.3	278.7	273.7	266.0	252.7	251.7	252.0	276.7	275.7	278.0	275.3	274.3	281.3	304.0	295.7	253.7	253.7	253.7
	H1	17	200	289.8	289.8	288.4	275	275.0	275	271.8	271.8	271.8	276.4	286.2	278.2	259.4	242.6	241.4	242.6	276	275	280.8	275	275	277	280.4	260	252.6	252.6	252.6
	H2	25	200	279.6	278.2	279.6	266.2	266.2	266.2	266.2	266.2	266.2	282.6	267.4	269.6	269.4	244.8	237	244.6	270.2	269.4	275.4	269.6	269	270.2	276.6	266	230.2	230.2	230.2
	H3	29	200	294.6	295	295.4	273.6	273.6	273.6	273.6	273.6	273.6	281.8	268.2	273.2	259.6	246	238.4	243.6	281.6	273	278.4	274.6	272.8	276.8	272.6	261.6	254	254	254
	H4	49	200	267.8	270.2	266.2	257.2	257.4	257.4	257.2	257.4	257.4	261.4	251.2	262	244.8	237.6	223.4	233.6	261	259.4	271	258.2	257.4	262.6	263	256.2	248.2	248.2	248.2
	H5	73	200	270.6	270.4	270.8	256.6	256.4	256.6	256.4	256.4	256.4	264	250.8	260.8	238	227.8	220.2	225.4	263.2	260.4	266.4	259.6	256.6	260.6	260.6	256.8	240.2	240.2	240.2
	H6	97	200	265.8	267.6	266	256.4	256.2	256.4	256.2	256.2	256.4	261.4	248.6	259.2	235.4	230	223	228.4	258.8	257.8	259.6	257	256.4	259.6	258	249.2	242.4	242.4	242.4
	H7	197	200	256	259.8	257	248.2	248.0	248.6	248.2	248	248.6	254.2	246.6	250.8	229.4	221.2	216	219	250.6	249.4	251.6	248.8	248.6	251	249	249	224.4	224.4	224.4
	T1	17	200	317.4	317.4	316	293.0	293.0	293.0	293.0	293.0	293.0	297.2	295.4	293.0	282.4	274.4	266.6	273.8	293.4	293.0	293.6	293.4	293	293.4	288.2	265.4	259	259	259
	T2	25	200	301.8	298.4	301.6	281.6	281.6	281.6	281.6	281.6	281.6	289.6	273.6	287.8	268.8	263.2	252.2	258.6	287.4	286.2	289.8	286.2	286.2	291.4	277.8	264.6	251.0	251.0	251.0
	T3	29	200	299.4	296	299.4	281.4	281.6	281.4	281.2	281.4	281.2	287	272.6	284	269.2	251.8	242.4	251.8	290.8	281.8	287.8	282.4	282.2	283	271.4	250.8	254.6	254.6	254.6
	T4	49	200	263.4	266.6	263.4	255	255.2	255.4	255	255.2	255.4	260	254	258.8	247.2	235	223.4	230.8	259.4	257.8	269.4	256.8	255.4	261	261.2	252.2	246.2	246.2	246.2
	T5	73	200	267.8	265.4	267.2	254.2	253.8	254.2	253.8	253.8	253.8	260.8	251.2	257.6	234.8	228.2	219	225.2	259	256	263	256	254.2	258.2	264.6	260.2	239.8	239.8	239.8
	T6	97	200	276	276.4	274.8	263.4	263.4	263.4	263.4	263.2	263.4	265.6	248	266.8	233.6	233.2	224.8	231.4	265.4	264.4	266.8	264.4	263.8	266.8	264.2	255.4	242.2	242.2	242.2
	T7	199	200	262	262.4	264	254.8	254.6	255.2	254.8	254.6	255.2	262.4	240.2	255.8	224.6	226	220.8	222.6	258.2	257.4	259.2	255.6	255.6	257.8	243.4	243.4	216.8	216.8	216.8
Munford *et al.* [94]	nice.25	25	100	133.8	134.0	134.7	130.6	130.6	130.7	130.6	130.6	130.7	138.3	139.1	132.3	131.1	123.1	120.4	122.3	134.3	132.0	136.1	132.0	131.5	133.5	133.5	132.8	120.8	122.7	120.5
	nice.50	50	100	125.6	125.9	125.7	122.2	121.9	122.1	122.2	121.9	122.1	129.1	129.5	122.2	125.4	118.1	115.8	117.7	124.9	123.2	126.9	123.4	122.2	125.4	125.5	125.5	116.6	116.7	116.1
	nice.100	100	100	120.6	120.4	120.5	117.8	117.7	117.6	117.8	117.7	117.6	122.4	122.2	117.8	120.0	113.2	111.2	112.6	120.1	118.3	121.1	118.5	117.8	121.0	119.2	119.2	113.8	114.4	112.9
	nice.200	200	100	115.1	115.0	115.2	113.2	113.1	113.3	113.2	113.1	113.3	117.2	116.5	112.9	116.1	110.3	108.4	109.6	115.5	114.0	115.7	114.4	113.2	117.1	113.9	113.9	110.7	112.5	109.1
	nice.500	500	100	109.9	109.9	109.8	108.2	108.2	108.4	108.2	108.2	108.4	111.8	111.7	108.2	112.8	106.5	105.4	106.2	110.1	108.9	110.4	109.9	108.2	112.6	108.5	108.5	107.6	110.2	105.4
	path.25	25	100	151.3	151.1	151.3	147.8	147.8	148.1	147.8	147.8	148.0	169.2	161.1	153.9	149.0	137.5	128.1	134.5	148.7	148.1	149.5	148.3	147.9	148.5	142.5	134.1	140.0	143.9	136.4
	path.50	50	100	156.3	156.4	156.3	149.3	149.2	149.6	149.3	149.2	149.6	168.7	155.0	158.7	138.7	137.2	124.1	132.5	150.9	150.5	151.8	150.3	149.7	151.3	139.3	133.7	141.9	142.2	141.7
	path.100	100	100	154.7	154.9	154.9	149.6	149.6	149.7	149.7	149.6	149.7	161.6	147.0	154.5	131.6	140.2	123.7	132.1	150.9	150.1	151.8	150.1	149.7	150.7	137.9	134.0	142.3	143.5	138.8
	path.200	200	100	152.5	152.4	152.1	147.8	147.8	147.8	147.8	147.8	147.8	153.0	135.6	150.7	125.3	140.8	122.5	129.4	149.2	148.1	149.6	148.2	147.8	148.9	134.3	131.5	134.2	137.3	132.6
	path.500	500	100	144.8	144.7	144.7	142.1	142.0	142.1	142.1	142.0	142.1	144.5	131.4	143.0	119.8	137.7	121.9	127.2	143.4	142.4	143.4	142.4	142.1	143.5	125.4	125.1	129.1	125.4	118.9

Table 5.3 (continued) : *Summary of mean packing heights obtained by offline algorithms over the 542 data sets described in §1.6. The shaded columns represent packing heights obtained by the new heuristics and proposed modifications as part of the contribution by the author. The algorithms to the left of the double vertical line are level algorithms while the algorithms to the right of the double line are plane algorithms.*

The reason for this observation is that in the FFDHDW algorithm, wider rectangles are packed first among rectangles of equal height, and since existing levels are always searched for sufficient space, the smaller rectangles may fit into any of the levels. This decreases the rate at which new levels are created, which in turn results in smaller packing heights.

The chi-squared test was used to determine whether there are significant differences between the frequencies in obtaining the smallest packing height by any of the *first fit* class of algorithms. In row 2 of Table 5.2, $\chi_2^2(7.102) > \chi_{critical}(5.990)$ at a 5% level of significance with two degrees of freedom, implying that there are significant differences between the frequencies. In terms of this outcome, the FFDHDW algorithm takes first preference, because it achieves the highest frequency. The chi-squared test was separately carried out using a Yates correction [36] (because there is now only one degree of freedom) to ascertain whether the frequencies given by the FFDH and FFDHIW algorithms were statistically distinguishable. The results of this test indicate that they are not distinguishable with $\chi_1^2(2.84) < \chi_{critical}(3.84)$, hence the FFDH and the FFDHIW algorithms are jointly ranked second within the first fit class of algorithms at a 5% level of significance.

Finally, the number of data sets for which each algorithm was able to achieve the smallest packing height is shown against the ratio of the standard deviations of the aspect ratio ($stdevAR$) of the data set with respect to the mean aspect ratio ($meanAR$) in Figure 5.2. For a fixed ratio of $stdevAR/meanAR = 2$, for example, as indicated in the figure by the horizontal dashed line, the number of data sets with ratio less than or equal to 2 for which the FFDHIW, FFDH and FFDHDW algorithms were able to obtain the smallest packing height are given by 376, 415 and 464 respectively (indicated by the vertical dashed lines). For data sets with little variation in the rectangle aspect ratios, the algorithms in the *first fit* class seem indistinguishable in terms of the frequency with which they are able to obtain the smallest packing height, as may be seen towards the left of Figure 5.2. However, as the variation in the rectangle aspect ratios increases, the algorithms become distinguishable in terms of the above mentioned frequency, as may be seen towards the right of the figure.

A further investigation with respect to the aspect ratios of the data sets was carried out to determine for what ratio of $stdevAR/meanAR$ in a data set, each algorithm was able to obtain the smallest packing height. This was done by considering the ratios of $stdevAR/meanAR$ for which each algorithm was able to attain the smallest packing height, and performing a chi-squared test for a range of values of the $stdevAR/meanAR$ up to a point where $\chi_{df}^2 \approx \chi_{critical}$ at a 5% significance level. Such a ratio becomes a threshold value in the sense that, for any data set with a $stdevAR/meanAR$ ratio beyond the threshold value, the FFDHDW algorithm is the preferred choice (because for a fixed standard deviation above this threshold, it obtained the smallest height in a larger number of data sets than the other two algorithms, as may be seen in Figure 5.2), while for ratios below the threshold, any one of the three algorithms may be used. In this class, the threshold value is given by a ratio of 0.677.

In the first fit class, the results from ANOVA indicated that there is no significant difference between solution qualities. On the other hand, the results of the chi-squared indicated that there is indeed a significant difference between the three algorithms in terms of the frequency with which smallest packing heights are achieved. Based on these results, only the FFDHDW algorithm will be considered for further comparisons since it

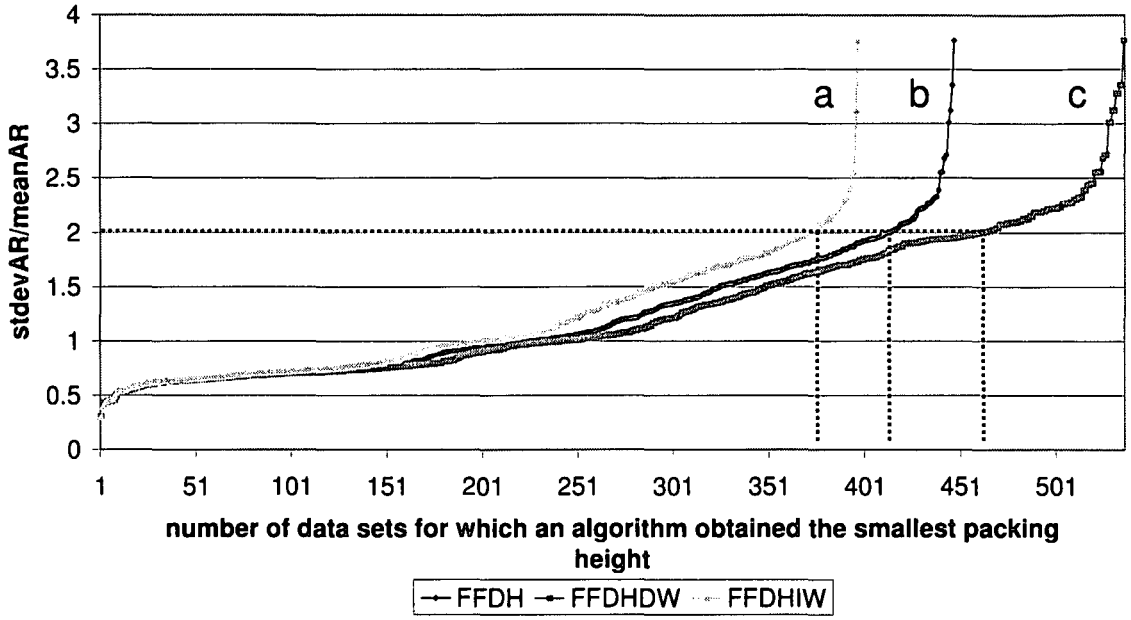

Figure 5.2: *Analysis of the aspect ratio variation of data sets for the* first fit *class of algorithms: a – FFDHIW, b – FFDH and c – FFDHDW.*

achieves the highest frequency.

5.1.3 Comparison of algorithms in the best fit class

The results of an ANOVA, when comparing the mean packing heights obtained by the algorithms in the *best fit* class (see §2.1.1.3, §2.2.5 and §2.2.6 respectively), indicate that there is no difference between the mean packing heights achieved by the algorithms at a 5% level of significance. This may be seen from row 3 of Table 5.1 where $F_{value}(0.0002) < F_{critical}(3.0013)$. Hence, in terms of solution quality, no algorithm in this class is superior with respect to another.

Based on the results of the chi-squared test (row 3 of Table 5.2), there are also no significant differences between the frequencies with which the smallest packing height was obtained by the three algorithms, because $\chi^2_{df}(5.285) < \chi_{critical}(5.990)$.

The aspect ratio analysis was not carried out for the *best fit* class of algorithms, because the results of the chi-squared test indicated that the algorithms are not distinguishable in terms of the frequencies with which the smallest packing height was obtained by these algorithms—hence there was no threshold value.

All three algorithms in the best fit class will therefore be considered during further comparisons, since there is no significant difference between the algorithms in terms of either the mean packing heights obtained or the frequencies with which the smallest packing height was obtained.

5.1.4 Comparison of AlgorithmJOIN and AlgorithmJOIN$_{mod}$

AlgorithmJOIN, described in §2.1.1.6, uses a given percentage γ to distinguish between rectangles that may be joined to form "super" rectangles. It was decided that the experimental runs should be carried out with three values of γ, namely, 1, 5 and 9, so as to determine the effect of the value of γ on the solution quality and the frequency with which the smallest packing height is achieved. AlgorithmJOIN$_{mod}$ (see §2.2.8) was also tested with these three values.

AlgJOIN01	AlgJOIN05	AlgJOIN09	AlgJOIN01$_{mod}$	AlgJOIN05$_{mod}$	AlgJOIN09$_{mod}$	F_{val}	F_{crit}
163.236	165.528	167.117				0.056	3.001
			162.564	163.789	165.720	0.039	3.001

Table 5.4: *Summary of results from the anova for AlgorithmJOIN and AlgorithmJOIN$_{mod}$ with $\gamma = (1, 5, 9)$. The bold faced values indicate the smallest average packing height.*

The results obtained from an ANOVA are shown in Table 5.4. The second and third rows represent the mean packing heights obtained by AlgorithmJOIN and AlgorithmJOIN$_{mod}$ respectively. In each instance the results indicate that there is no significant difference between the mean packing heights obtained over the 542 data sets.

In terms of the frequency of obtaining the smallest packing height, the chi-squared tests revealed that there are significant differences, as shown in Table 5.5, with AlgorithmJOIN01 and AlgorithmJOIN01$_{mod}$ achieving the highest frequencies. These two algorithms were further tested against each other in an attempt at determining whether the modified algorithm leads to an improvement in either solution quality or frequency with which smallest packing heights are achieved.

AlgJOIN01	AlgJOIN05	AlgJOIN09	AlgJOIN01$_{mod}$	AlgJOIN05$_{mod}$	AlgJOIN09$_{mod}$	χ^2_{df}	χ_{crit}
496	161	102				**356.97**	**5.99**
			520	247	151	**239.55**	**5.99**

Table 5.5: *Summary of results from the chi-squared test for AlgorithmJOIN and AlgorithmJOIN$_{mod}$ using $\gamma = (1, 5, 9)$. The bold faced values indicate highest frequency and that there is a significant difference between the frequencies in each algorithm.*

When the mean packing heights obtained by algorithmJOIN01 and algorithmJOIN01$_{mod}$ were tested by means of a student's t-test, the results in row 2 Table 5.6 were obtained. These results indicate that there are no significant differences between the packing heights obtained. However, when testing the frequency with which the smallest packing height is obtained, the chi-squared test indicated that there are significant differences, with algorithmJOIN$_{mod}$ achieving the highest frequency of 488, as shown in Table 5.7 (see first segment of Figure 5.3).

Based on the observation that algorithmJOIN01$_{mod}$ achieves a better frequency with which the smallest packing height is achieved, the algorithm was selected for consideration during further comparisons. AlgorithmJOIN01$_{mod}$ is expected to perform well when data sets comprise rectangles with almost similar heights—therefore creating "super" rectangles composed of more than just two rectangles.

AlgJOIN01	AlgJOIN01$_{mod}$	SF	SF$_{mod}$	FCNR	FCNR$_{mod}$	Sleator	Sleator$_{mod}$	t_{stat}	t_{crit}
163.236	**162.564**							0.132	1.962
		173.123	171.144					0.061	1.962
				152.157	**148.940**			0.307	1.962
						163.238	**159.450**	0.274	1.962

Table 5.6: *Summary of results from the student's t-test for AlgorithmJOIN01, the SF algorithm, the FCNR algorithm and the Sleator algorithm along with their respective modifications. The bold faced values indicate smallest packing height per group.*

5.1.5 Comparison of SF and SF$_{mod}$ algorithms

Because only two algorithms, namely the SF and SF$_{mod}$ algorithms (see §2.1.1.5 and §2.2.7 respectively) are compared in this section, the student's t-test was employed as a tool to facilitate the comparison of mean packing heights. The mean packing heights obtained by each algorithm over all the 542 benchmark data sets are shown in Table 5.6. The results of the t-test indicate that there is no significant difference between the mean packing heights obtained.

The algorithms were also compared in terms of the number of times each algorithm obtained the smallest packing height and the results are shown in Table 5.7 (see also second segment of Figure 5.3). Clearly, there is a significant difference between the frequencies obtained, with the SF$_{mod}$ algorithm having the highest frequency.

AlgJOIN01	AlgJOIN01$_{mod}$	SF	SF$_{mod}$	FCNR	FCNR$_{mod}$	Sleator	Sleator$_{mod}$	χ^2_{df}	χ_{crit}
226	**488**							**27.53**	**3.84**
		263	**399**					**95.41**	**3.84**
				268	**539**			**90.34**	**3.84**
						392	**525**	**19.00**	**3.84**

Table 5.7: *Summary of results from the chi-squared for AlgorithmJOIN01, SF, FCNR and Sleator algorithms along with their respective modifications. The bold faced values indicate highest frequency and that there is a significant differences between frequencies for each group.*

If the list of rectangles to be packed comprises of wide rectangles which do not fit into the regions $\mathcal{R}$, the SF$_{mod}$ algorithm was observed to perform poorly compared to the original SF algorithm. This is because in the SF algorithm, the regions $\mathcal{R}$ are fixed—therefore the rectangles may be packed above this region, thereby utilising the entire width of the strip. However, in the SF$_{mod}$ algorithm, since the regions $\mathcal{R}$ are not fixed, these parts of the strip will not be utilised for any packing, since in this scenario none of the rectangles fit into this region. This means that only part of the strip and not the entire width of the strip is considered for packing. Clearly, under such a scenario, the SF algorithm would achieve a better performance.

5.1.6 Comparison of FCNR and FCNR$_{mod}$ algorithms

As described in §2.1.1.7, the floor-ceiling algorithm may be implemented such that it produces guillotine or non-guillotine patterns. The two versions along with the possible modification were implemented with the non-guillotine algorithm denoted as FCNRNG. As expected the FCNRNG algorithm on average outperformed the FCNR algorithm, since it does not leave any spaces on the ceilings of levels (see Tables 5.3). However, the performance of the FCNR algorithm against the proposed FCNR$_{mod}$ algorithm is of interest.

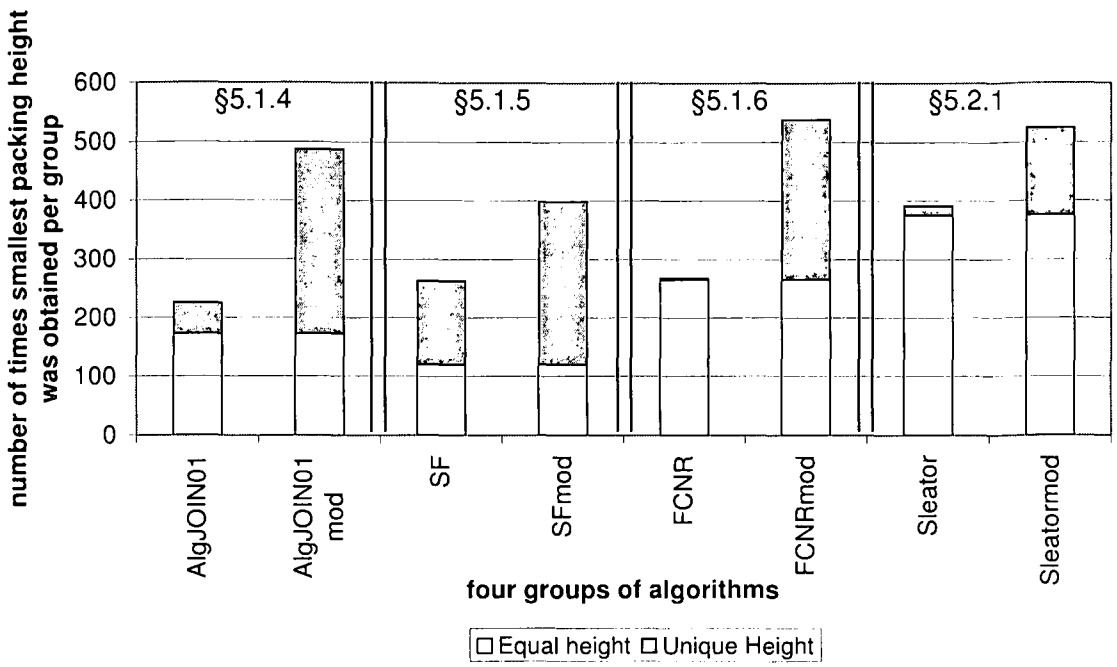

Figure 5.3: *Summary of frequency with which smallest packing heights were obtained by four groups of algorithms. The solid vertical lines separate the four groups to indicate that they are not compared against each other. Comparison is only between algorithms in the same group.*

Since there are two algorithms, the student's t-test was used to compare the mean packing heights obtained by the FCNR and FCNR$_{mod}$ algorithms over the 542 data sets. As shown in Table 5.6, the FCNR$_{mod}$ achieves a smaller packing height, but statistically there is no significant difference between the two mean packing heights achieved.

In terms of the frequency with which the algorithms obtained the smallest packing height, results of the chi-squared test shown in Table 5.7 (see also third segment of Figure 5.3) indicate that there is a significant difference. The FCNR$_{mod}$ algorithm achieved the highest frequency and it is on this basis that the algorithm will be used for further comparisons. This was an expected result, because the FCNR$_{mod}$ algorithm searches all the gaps created on the ceilings of levels for sufficient space before searching the floors of the levels, still maintaining the guillotine pattern.

5.1.7 Comparison of the resulting 9 level algorithms

In this section, all the best performing level algorithms are compared against each other along with the KP01 (see §2.1.1.4) and the SAS (see §2.3) algorithms. It is acknowledged that the SAS and FCNR$_{mod}$ algorithms have an unfair advantage over the other algorithms, since rectangles are not only packed on the floors of levels in these algorithms. The SF$_{mod}$ algorithm differs from the other algorithms in that it allows the *wide* rectangles already packed to be shifted around, but the results in Tables 5.3 indicate that this advantage over the other algorithms does not render the algorithm superior with respect to the other procedures. It is also worth noting that none of the heuristics were able to achieve the optimal heights for those data sets where optimal solutions are known.

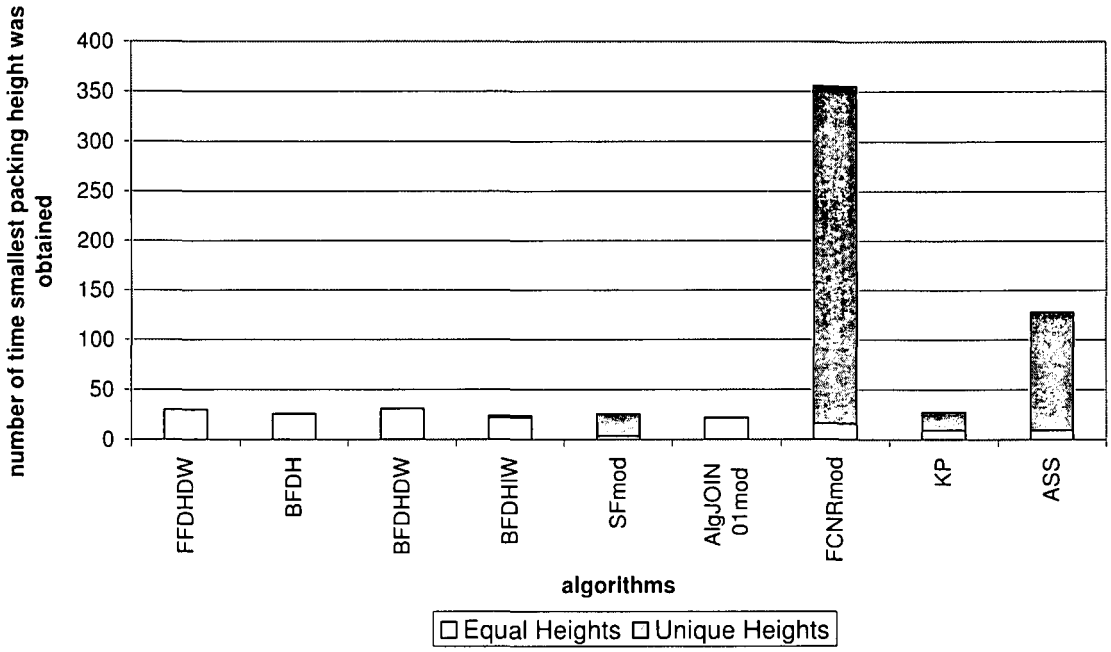

Figure 5.4: *The number of times each of the resulting 9 level algorithms obtained the smallest packing height.*

The first section of Table 5.8 represents the results from an ANOVA indicating that there is no difference between the mean strip heights obtained by the nine algorithms at a 5% level of significance. The second section contains the results from the chi-squared test on the number of times each algorithm obtained the smallest packing height and the results indicate that there is a significant difference between these frequencies. Based on the results from the frequency analysis, the FCNR$_{mod}$ algorithm is superior with respect to the algorithms, followed by the newly proposed SAS algorithm, as shown graphically in Figure 5.4. This was an expected result because both these algorithms attempt to utilise each level fully, thereby leading to reduced total packing heights (the FCNR$_{mod}$ algorithm by utilising both the floors and ceilings of levels, and the SAS algorithm by stacking rectangles vertically within a level)—hence these algorithms have an advantage over the other (classical) level algorithms, leading to performance superiority. An analysis of the set of nine level algorithms with respect to the aspect ratios of the data sets is shown in Figures 5.5 and 5.6—the latter figure is an enlargement of the section indicated by a rectangular shape A in the former figure. A threshold value of 0.588 was obtained for this set of algorithms and the FCNR$_{mod}$ algorithm outperforms the other eight algorithms for $stdevAR/meanAR$ ratios above this value.

The SAS algorithm was observed to perform well as the coefficient of variation increases.

FFDHDW	BFDH	BFDHDW	BFDHIW	SF$_{mod}$	AlgJOIN01$_{mod}$	FCNR$_{mod}$	KP	SAS	F_{val}	F_{crit}
161.592	161.524	161.427	161.633	171.144	162.564	148.940	164.305	156.080	0.540	1.940
FFDHDW	BFDH	BFDHDW	BFDHIW	SF$_{mod}$	AlgJOIN01$_{mod}$	FCNR$_{mod}$	KP	SAS	χ^2_{df}	χ_{crit}
30	26	31	24	26	22	356	28	128	1316.47	15.51

Table 5.8: *Results from the ANOVA and chi-squared test for the resulting 9 level algorithms.*

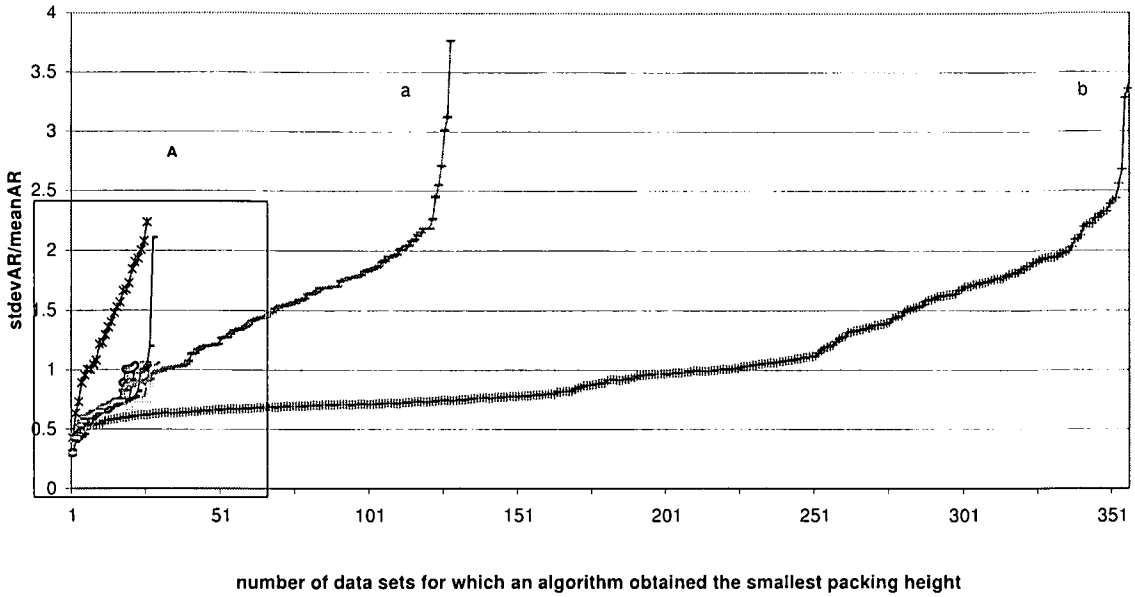

Figure 5.5: *Analysis of the aspect ratio variation of data sets for the resulting nine level algorithms: a – SAS, b – FCNR$_{mod}$, rectangular region A showing aspect ratios of the remaining seven algorithms.*

For small variations it does not do well compared to the other classical algorithms such as the FFDH algorithm. This was seen by the number of data sets for which it was able to obtain the smallest packing height when carrying out the aspect ratio analysis.

As mentioned earlier, the FCNR and SAS algorithms have an advantage over the other level algorithms, by not only packing on the floors of levels, but by also packing on the ceilings or by stacking the rectangles on top of one another. If, in certain applications, it is not possible to pack rectangles on the ceilings of levels or if it is not possible to stack rectangles on top of one another, then it would be interesting to investigate which of the remaining 7 level algorithms are best in terms of the frequency with which smallest packing height is obtained. This investigation was then carried out with the FFDHDW, BFDH, BFDHDW, BFDHIW, SF$_{mod}$, AlgJOIN01$_{mod}$ and KP01 algorithms. The mean packing heights obtained by the algorithms over the 542 data sets were compared and the results of the ANOVA indicate that there is no significant difference between the mean packing heights at 5% level of signifance. The frequencies with which each of these 7 algorithms obtained the smallest packing height are shown in Figure 5.7.

However, the results of the chi-squared test, indicated that there are significant differences between the frequencies displayed in Figure 5.7 with the BFDHDW algorithm achieving the highest frequency.

The aspect ratio analysis was then carried out on this instance with the results shown in Figure 5.8. The threshold value was computed as 0.602 and for any data set with mean aspect ratio above this threshold value, the BFDHDW algorithm is recommended for use.

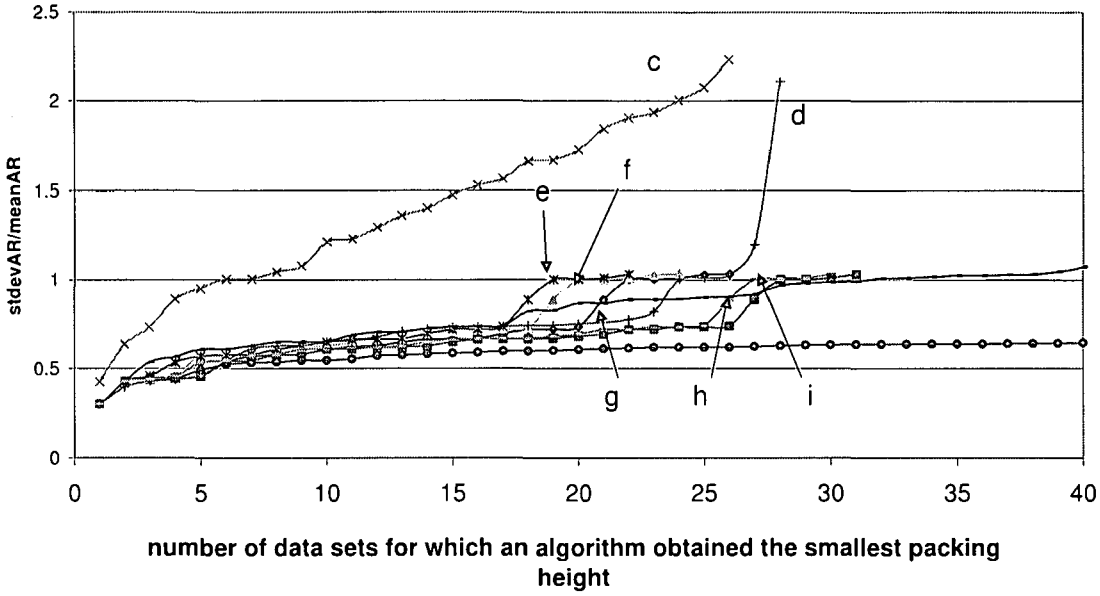

Figure 5.6: *Analysis of the aspect ratio variation of data sets for the resulting nine level algorithms. Enlargement of rectangular region A in Figure 5.5: c – SF_{mod}, d – KP01, e – $AlgJOIN01_{mod}$, f – BFDHIW, g – BFDH, h – FFDHDW, i – BFDHDW.*

5.2 Comparison of plane algorithms

The same criteria used when comparing level algorithms were also employed when comparing plane algorithms. Similar statistical tools as described in §5.1 were also employed to facilitate the investigations in this section. The mean packing heights obtained by the plane algorithms over all 542 data sets are shown in the last five columns of Table 5.3.

5.2.1 Comparison of Sleator and $Sleator_{mod}$ algorithms

The plane algorithms Sleator (§2.1.2.1) and $Sleator_{mod}$ (§2.2.10) were also compared in terms of the mean packing heights obtained over the 542 benchmark data sets. The results of the student's t-test shown in row 4 Table 5.6, indicate that there is no significant difference between the mean packing heights obtained by these two algorithms.

The results of the chi-squared test indicate that there are significant differences between the frequencies with which the smallest packing is obtained by each algorithm, with the $Sleator_{mod}$ algorithm achieving the highest frequency. Based on this outcome, only the $Sleator_{mod}$ algorithm will be considered during further comparisons with another plane algorithm.

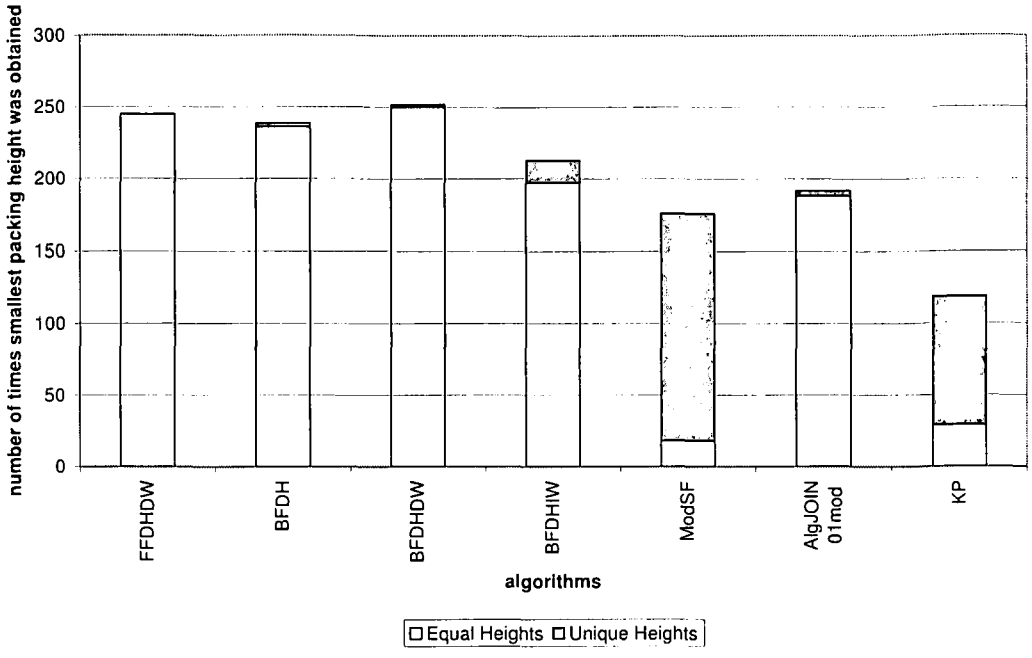

Figure 5.7: *The number of times each of the remaining seven algorithms obtained the smallest packing height.*

5.2.2 Comparison of rounding techniques in the Burke algorithm

As mentioned in §2.1.2.2, the Burke algorithm uses a linear array; therefore requiring both the strip width and the dimensions of the rectangles to be positive integers. It was suggested by Whitwell [118] that it would be interesting to investigate the effects of different ways of rounding the dimensions when dealing with floating point data. As a result three methods of rounding were tested, namely, *rounding up*, *rounding to the nearest integer* and *rounding down*. Before rounding, a similar strategy employed by Burke *et al.* [18] of first multiplying the floating point data by 10 was also used in order to ensure that non-zero dimensions are obtained. As expected, when rounding down,

Nearest	UP	Down	F_{val}	F_{crit}
151.605	152.760	**150.245**	0.031	3.001
Nearest	UP	Down	χ^2_{df}	χ_{crit}
224	194	**379**	**74.216**	**5.990**

Table 5.9: *Summary of results from an ANOVA and chi-squared when three ways of rounding floating point data are employed in the Burke algorithm. The bold faced values, indicate the smallest mean packing height and highest frequency with which the smallest packing height is achieved for results from ANOVA and chi-squared test respectively.*

the mean packing height obtained over all 542 data sets is the smallest, even though the results of the ANOVA indicate that there are no significant differences between the mean packing heights, as shown in the second row of Table 5.9. The results of the chi-squared test indicate that there are significant differences between the frequency with which the

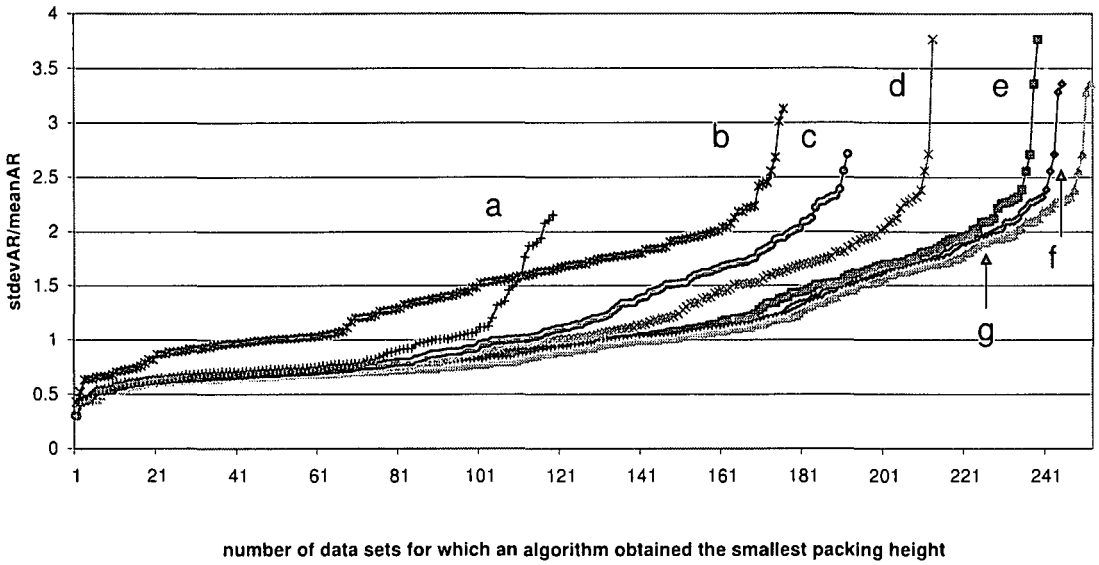

Figure 5.8: *Analysis of the aspect ratio of data sets for the remaining 7 algorithms: a – KP01, b – SF_{mod}, c – $AlgJOIN01_{mod}$, d – BFDHIW, e – BFDH, f – FFDHDW, g – BFDHDW.*

smallest packing height is obtained by the algorithms with rounding down having the highest frequency (fourth row of Table 5.9).

It is worth noting, however, that rounding down and rounding to the nearest integer may not necessarily result in a feasible packing, since some of the original rectangles may not fit in the resulting packing pattern. However, when rounding to the nearest integer, the characteristics of the data are maintained. On the other hand, rounding up wastes space, but creates a feasible packing for the original rectangles.

5.2.3 Comparison of $Sleator_{mod}$ and $Burke_{Down}$ algorithms

When comparing the mean packing heights achieved by the $Sleator_{mod}$ (159.450) and $Burke_{Down}$ (150.245) algorithms, the results of the t-test indicate that there is no significant difference at 5% level of significance.

In terms of the frequency with which the smallest packing heights were achieved, the results of the chi-squared test revealed that the algorithms are distinguishable at a 5% level of significance with values of 155 and 391 achieved by the $Sleator_{mod}$ and $Burke_{Down}$ algorithms respectively. For the plane algorithms, the $Burke_{Down}$ algorithm is superior, as expected, in terms of the frequency with which the smallest packing height is achieved. However, as mentioned earlier, the disadvantage of this algorithm is that it does not lead to a feasible packing of the original instance.

Author	Data Sets	# of Rectangles	NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	SF	SF_{mod}	SAS	FCNR	FCNRNG	$FCNR_{mod}$	AlgJOIN01	AlgJOIN05	AlgJOIN09	$AlgJOIN01_{mod}$	$AlgJOIN05_{mod}$	$AlgJOIN09_{mod}$	Sleator	$Sleator_{mod}$	Burke	BurkeUP	BurkeDown
Beasley [9, 10]	U_1	10	16	32	32	31	31	31	31	31	32	31	31	32	31	32	32	31	31	31	31	31	16	31	47	32	32	47
	U_2	20	31	31	31	32	31	32	31	31	31	31	31	31	32	32	32	31	31	31	31	31	31	31	31	31	31	47
	U_3	30	31	32	31	32	32	32	31	31	31	31	31	31	32	31	32	32	31	31	32	31	32	31	32	31	47	32
	U_4	50	31	32	32	31	31	31	31	32	32	47	47	31	47	31	31	47	32	31	32	47	32	31	47	47	46	47
	V_1	10	31	31	32	31	32	32	31	31	32	31	31	31	31	31	31	31	32	31	31	31	31	15	31	32	47	31
	V_2	17	31	32	31	32	16	32	31	32	16	47	31	31	31	31	31	31	31	31	31	31	31	31	31	32	47	31
	V_3	21	15	32	32	31	31	31	31	32	32	31	31	31	31	31	47	31	32	31	32	31	31	31	31	31	47	31
	V_4	7	31	31	32	31	32	31	31	31	32	32	31	32	31	31	31	31	16	32	31	31	31	31	31	31	31	32
	V_5	14	31	16	31	32	32	31	31	31	31	32	32	32	31	32	31	31	31	31	31	31	31	15	31	31	47	32
	V_6	15	31	32	31	31	31	31	31	32	32	31	32	31	32	31	31	31	31	31	31	31	31	31	32	31	32	31
	V_7	8	31	31	32	31	32	31	15	31	32	32	31	32	31	31	31	31	31	32	31	31	31	31	31	31	32	31
	V_8	13	31	31	31	31	32	31	31	31	31	32	32	32	31	32	16	31	31	32	31	31	31	31	31	31	46	31
	V_9	18	31	32	31	31	16	31	31	16	31	31	32	31	32	32	31	32	31	31	31	32	31	32	31	32	47	31
	V_{10}	13	31	31	31	31	32	31	31	31	16	31	31	31	32	31	31	32	31	32	31	32	16	32	31	32	31	31
	V_{11}	15	31	31	31	31	32	31	31	31	31	32	31	31	31	31	32	31	31	32	31	31	31	31	31	31	31	31
	V_{12}	22	31	31	31	31	31	31	31	31	31	31	31	31	47	47	32	31	31	31	31	32	31	32	31	31	31	32
Burke *et al.* [18]	B_1	10	16	31	31	31	16	31	31	15	31	31	31	31	32	31	31	32	31	31	31	32	32	32	32	31	31	32
	B_2	20	31	31	31	31	32	31	31	31	15	31	31	31	31	31	47	31	31	16	31	31	32	31	32	31	32	31
	B_3	30	31	31	31	31	31	31	31	31	31	31	47	31	31	31	32	31	31	31	31	32	31	47	62	47	32	47
	B_4	40	32	31	31	31	31	31	31	31	31	31	32	47	31	31	31	31	31	31	32	32	31	47	31	46	47	31
	B_5	50	32	31	47	47	31	31	31	31	31	31	32	46	31	47	47	31	32	31	32	31	31	47	31	47	47	47
	B_6	60	31	31	47	31	31	32	32	31	31	32	31	31	47	46	47	31	32	31	31	31	31	47	32	46	62	62
	B_7	70	47	32	46	31	31	32	32	32	31	32	31	31	31	31	32	31	31	47	31	47	31	31	47	62	47	63
	B_8	80	47	47	47	31	31	31	31	32	32	47	47	47	47	47	47	47	47	46	31	47	46	32	47	62	63	62
	B_9	100	47	46	47	47	32	31	31	31	32	31	32	47	47	47	47	47	47	31	31	47	32	47	47	47	62	63	62
	B_{10}	200	31	31	31	47	47	47	47	47	47	46	63	78	63	62	62	47	47	47	47	47	47	78	63	141	141	140
	B_{11}	300	62	62	62	47	47	47	47	46	47	63	63	94	78	78	78	63	62	63	47	47	63	110	110	235	234	250
	B_{12}	500	62	79	62	78	63	78	62	63	63	78	109	172	110	125	109	94	78	78	78	78	78	219	203	563	547	547
Christofides [22]	G_1	16	32	31	31	31	31	31	31	31	31	32	32	31	31	31	31	31	31	32	31	31	31	31	31	31	31	31
	G_2	23	31	31	31	32	31	32	31	31	31	32	32	31	31	31	31	32	31	31	47	32	31	32	31	31	31	31
	G_3	62	31	32	31	32	47	31	32	31	31	31	31	32	47	31	31	32	31	31	47	32	47	32	31	47	47	62

Table 5.10: *Summary of mean execution times (in milliseconds) obtained by the offline algorithms over the 542 data sets. The shaded columns represent new heuristics and proposed modifications as part of the contribution by the author. The algorithms to the left of the double vertical line are level algorithms, while the algorithms to the right of the double vertical line are plane algorithms. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rectangles	NFDH	NFDHDW	NFDHIW	FFDH	FFDHDW	FFDHIW	BFDH	BFDHDW	BFDHIW	SF	SF_{mod}	SAS	FCNR	FCNRNG	$FCNR_{mod}$	AlgJOIN01	AlgJOIN05	AlgJOIN09	$AlgJOIN01_{mod}$	$AlgJOIN05_{mod}$	$AlgJOIN09_{mod}$	Slentor	$Slentor_{mod}$	Burke	BurkeUP	BurkeDown
Hopper *et al.* [58, 57]	C1	16/17	31.3	31.0	31.0	31.0	26.7	31.3	31.0	26.0	31.3	31.0	36.7	36.3	31.3	31.0	36.7	31.3	31.0	31.0	31.3	31.3	31.7	31.0	31.3	31.0	31.0	31.3
	C2	25	31.3	31.0	31.0	26.0	31.3	26.0	31.3	31.0	31.0	31.3	31.0	31.7	31.3	31.7	31.7	31.0	31.7	31.0	31.7	31.0	31.3	31.0	31.7	41.7	41.7	36.7
	C3	28/29	31.7	26.0	31.0	31.7	31.3	31.0	31.7	31.0	31.0	31.3	31.0	31.0	36.3	31.0	31.0	31.0	36.7	31.3	31.0	31.0	31.0	31.7	31.0	36.0	41.3	36.7
	C4	49	31.3	31.3	31.7	36.3	31.7	31.0	36.3	31.3	31.7	31.0	31.3	37.0	41.7	36.3	37.0	46.7	31.0	36.3	31.7	36.3	31.7	47.0	47.0	47.0	47.0	46.3
	C5	72/73	37.0	31.0	41.7	31.7	46.7	31.7	31.0	31.3	31.7	41.3	36.3	41.7	41.3	41.7	41.7	36.3	36.7	36.7	31.0	42.0	31.0	36.7	41.7	57.3	57.3	57.3
	C6	97	31.0	41.7	31.3	31.0	41.7	36.3	37.0	36.3	42.0	36.3	46.7	47.0	41.7	41.7	47.0	36.3	41.3	36.7	36.0	36.3	41.3	41.7	41.7	62.3	73.0	68.3
	C7	196/197	41.7	47.0	47.0	47.0	47.0	42.0	41.3	46.7	47.0	47.0	57.3	67.3	62.3	63.0	63.0	47.0	46.7	46.3	47.0	52.3	47.0	63.0	67.7	135.0	125.0	130.3
	H1	17	31.0	28.2	31.0	28.6	28.6	31.0	27.8	31.0	31.4	31.0	34.6	34.4	31.2	31.6	31.6	31.2	31.0	31.6	31.0	31.2	31.6	31.0	31.0	34.4	34.0	40.8
	H2	25	31.0	31.0	31.0	31.2	31.0	31.4	27.8	31.0	31.0	31.2	31.2	31.4	34.2	31.0	31.2	31.0	34.6	31.0	31.4	31.4	34.2	31.4	31.4	37.2	34.4	37.6
	H3	29	31.2	31.0	31.4	31.0	34.4	31.4	31.2	31.0	31.0	34.6	34.4	31.2	31.4	31.4	34.4	31.4	31.2	31.4	34.4	31.2	31.4	31.0	34.4	40.6	40.4	37.4
	H4	49	31.4	31.4	31.0	31.0	37.6	31.2	34.4	31.4	34.6	43.8	34.2	34.6	37.2	34.4	44.0	31.0	31.4	31.0	31.2	31.4	34.0	31.4	37.2	47.0	46.6	47.0
	H5	73	31.4	31.0	31.4	31.4	31.0	31.4	37.2	31.4	47.0	37.2	43.6	40.6	37.4	44.0	43.6	37.8	34.2	34.4	37.8	34.2	37.8	37.4	40.6	56.2	53.2	56.2
	H6	97	31.0	34.6	34.4	37.4	37.4	37.8	37.4	37.6	41.0	43.6	50.0	43.6	40.6	46.6	43.8	40.6	40.8	37.4	41.0	37.4	34.4	40.8	37.6	69.2	72.2	69.2
	H7	197	43.6	43.8	47.0	46.8	47.0	40.8	40.6	46.8	44.0	47.0	59.4	68.8	59.2	62.6	59.6	62.6	53.4	47.0	46.4	50.0	49.6	62.2	68.4	137.4	134.4	137.4
	T1	17	31.2	27.8	31.0	31.0	34.6	28.0	31.6	31.8	28.6	34.4	31.4	31.6	31.2	31.4	31.2	31.2	31.4	31.0	31.0	31.0	31.0	31.4	31.4	31.4	40.4	34.6
	T2	25	28.4	31.0	27.8	31.4	31.4	34.2	31.4	31.2	31.0	34.6	31.4	31.0	31.4	40.6	40.6	31.4	31.0	31.8	31.0	31.4	31.6	31.2	31.4	41.0	37.2	37.6
	T3	29	31.2	31.2	31.4	31.4	31.0	31.6	31.0	31.0	31.4	31.0	31.4	37.8	31.4	31.4	31.0	31.0	31.4	31.0	31.4	31.4	34.2	31.0	31.4	34.2	43.8	40.4
	T4	49	31.0	31.4	34.2	34.2	34.4	34.2	34.4	31.4	34.2	31.4	37.4	37.6	37.4	37.6	37.8	31.4	31.0	31.4	34.0	31.2	40.6	34.6	34.0	50.2	47.0	50.0
	T5	73	46.8	31.4	46.8	31.4	31.2	46.8	31.2	47.0	31.4	31.2	40.6	40.4	40.6	40.8	40.6	37.4	37.8	34.2	37.2	31.2	37.4	37.8	40.6	56.2	53.0	59.6
	T6	97	31.4	34.2	34.4	37.2	37.2	37.8	40.4	31.4	40.4	37.2	43.6	44.0	50.0	43.4	43.8	34.0	37.4	37.4	37.2	40.6	40.6	40.6	46.6	66.0	62.8	65.2
	T7	199	46.6	46.6	43.6	43.8	46.8	43.8	43.6	43.8	43.6	47.0	53.0	75.0	62.8	62.2	59.2	50.0	46.6	56.0	47.0	47.0	56.0	65.2	68.6	137.2	134.2	134.2
Mumford *et al.* [94]	nice.25	25	30.0	31.0	30.3	31.9	30.3	31.6	31.0	31.0	30.7	31.9	32.5	32.8	32.8	34.4	32.2	31.6	31.3	31.8	31.3	31.2	32.5	30.9	32.5	40.6	39.3	41.0
	nice.50	50	32.5	32.8	31.9	32.5	32.8	32.5	32.5	32.5	33.1	38.8	37.0	39.0	37.1	37.7	36.7	34.7	35.4	33.7	33.8	33.7	36.0	31.8	36.9	51.3	50.9	51.9
	nice.100	100	36.4	36.9	36.4	37.2	36.3	36.6	36.1	36.8	36.5	39.7	43.4	47.9	46.2	45.9	46.0	40.9	40.7	39.4	39.4	38.4	38.8	39.4	45.7	76.9	75.3	75.3
	nice.200	200	43.8	45.3	45.4	43.7	44.4	43.7	45.0	45.3	44.7	50.7	58.7	89.7	65.6	63.7	63.8	47.3	49.7	50.2	51.6	48.7	47.6	58.8	76.3	149.1	145.9	147.5
	nice.500	500	68.4	69.2	68.4	70.4	72.3	69.2	65.3	68.4	66.8	82.9	103.2	300.1	135.9	120.2	131.3	79.6	81.3	81.1	82.8	78.0	78.3	159.3	250.0	537.8	534.6	543.4
	path.25	25	30.3	30.6	30.3	30.0	30.6	29.6	30.9	31.2	31.2	31.9	33.1	32.8	32.5	33.8	33.1	31.8	31.6	31.9	31.6	32.3	32.2	30.7	32.5	40.3	38.8	40.0
	path.50	50	33.1	32.8	32.8	33.1	33.1	32.9	32.2	32.2	32.8	33.4	36.6	37.1	37.4	37.6	37.0	37.8	32.5	34.1	34.7	31.2	32.2	35.3	37.2	50.6	48.7	50.0
	path.100	100	36.5	35.9	36.8	35.3	36.7	36.9	36.5	36.7	36.3	40.3	43.7	47.8	44.4	45.0	44.4	40.6	38.7	39.4	37.8	37.8	38.5	40.3	46.9	73.8	73.5	73.7
	path.200	200	44.4	44.7	45.2	45.0	45.3	44.1	43.1	45.0	44.4	50.6	57.8	80.6	63.5	62.3	60.8	48.7	49.1	47.6	48.9	48.2	48.2	59.3	76.9	142.8	138.4	142.8
	path.500	500	67.2	70.8	66.8	70.3	67.7	69.1	69.0	71.6	68.9	84.5	96.9	267.2	126.6	118.8	118.9	81.2	81.1	79.6	81.2	78.3	79.7	171.9	267.4	500.0	492.4	503.2

Table 5.10 (continued) : *Summary of mean execution times (in milliseconds) obtained by the offline algorithms over the 542 data sets. The shaded columns represent new heuristics and proposed modifications as part of the contribution by the author. The algorithms to the left of the double vertical line are level algorithms, while the algorithms to the right of the double vertical line are plane algorithms. A function with resolution of 10ms was used to measure execution time.*

5.3 Comparison of execution times

Another important element to investigate is the running time of each algorithm. This is because execution time may dictate one's choice of algorithms, if there is an urgency with respect to obtaining the solutions. The running time is affected by many factors, some of which include the input data and the computer system used to implement the algorithm [103]. All the algorithms mentioned in this dissertation were implemented on a computer with a 2.00 GHz processor and 224 MB of RAM.

The KP01 algorithm was excluded from the running time investigation since exhibiting running times of more than 3 hours for large data sets (with 500 rectangles)—so clearly, in terms on running times, it is outperformed by all other algorithms. Each stage of the KP01 algorithm requires solving a knapsack problem and it was implemented such that MatLab [88] is called from Visual Basic to solve the knapsack problem and this is time consuming.

The results of running times of the algorithms are shown in milli seconds in Tables 5.10. The first part of this table, shown on page 112, represents running times obtained per single instance of the data sets, while the second part of the table on the following page represents average times computed over three instances for the C_i categories, five instances for the N_i and T_i categories and finally fifty instances for the *nice* and *path* data sets. It may be seen that the larger the number of rectangles, the longer the running times of the various algorithms, as expected.

The first fit and best fit classes require longer running times on average compared to the next fit class, because in the former class all levels are searched for sufficient space and this is time consuming. This result is not immediately evident in our results because a function with resolution of 10ms was used to measure execution times.

The suggested SF_{mod} algorithm, which was observed to achieve a higher frequency in obtaining the smallest packing height, requires longer running times on average than the SF algorithm. This is again an expected result, because the SF_{mod} algorithm additionally searches the wide rectangles for sufficient horizontal space in establishing the regions $\mathcal{R}$.

It is inconclusive which of AlgorithmJOIN and $\text{AlgorithmJOIN}_{mod}$ with the three values of γ requires longer running times. The same observation may be drawn for FCNR or FCNR_{mod} algorithms—it is not clear which algorithm has shorter running times. The SAS algorithm is observed to require longer running times when solving instances with a large number of rectangles compared to all other level algorithms. This is not a surprising result, because at every iteration in the SAS algorithm, the list of rectangles is scanned to find suitable rectangles to stack—if these lists are large, then significantly longer running times are required.

Amongst the plane algorithms, the Sleator_{mod} algorithm requires, on average, longer running times than the Sleator algorithm. This is because the stacked rectangles at the bottom of the strip are searched for sufficient horizontal space to pack the tallest unpacked rectangle and this additional step may lead to reduced packing heights but it affects the running times negatively. When considering the performance of the Burke algorithm, with respect to the floating point data by Mumford-Valenzuela *et al.* [94], in the second part of Table 5.10 (because this is where the effects of rounding are observed), rounding up appears to result in shorter running times compared to the other two methods of

rounding.

The results from the execution times confirm what was stated earlier in the introduction to this chapter, namely that there is a trade-off between solution qualities and execution times. The algorithms that yield good solutions (bad solutions, respectively) require longer running times (shorter running times, respectively). It is up to the user to decide, depending on the application, whether solutions quality or execution time is the more important consideration.

5.4 Results of exact algorithms

The exact algorithms described in Chapter 4 were also implemented in Visual Basic 6.0 and tested on the 542 benchmark instances described in §1.6.

Data sets	# of Rec	wa/W	OPT	G/NG	z	Nodes	Branch time	Iterations
C_1^1	16	0.2344	20	NG	25	8 681	683.513	169 220
C_1^2	17	0.2824	20	NG	28	3 662	452.350	77 765
C_1^3	16	0.2750	20	NG	28	6 354	563.090	87 350
G_1	16	0.3250	-	G	28	1 461	185.156	38 179
U_1	10	0.5840	-	G	1 016	141	6.590	277
V_1	10	0.5000	-	NG	25	121	5.909	592
V_2	17	0.4647	-	NG	33	5 434	583.769	88 944
V_4	7	0.2143	-	NG	23	59	1.292	163
V_5	14	0.2857	-	NG	37	6 830	387.007	70 332
V_6	15	0.4400	-	NG	38	23 088	2 672.3	500 201
V_7	8	0.2313	-	NG	21	5	0.781	52
V_8	13	0.2269	-	NG	38	264	19.959	3 742
V_{10}	13	0.3872	-	NG	85	8 270	356.202	98 682
V_{11}	15	0.3222	-	NG	69	14 204	749.828	241 066
B_1	10	0.2750	40	G	46	11	1.142	107
T_1^1	17	0.2671	200	G	300	14 680	1 054	105 760
T_1^3	17	0.2076	200	G	Time out	28 550	7 200	409 594

Table 5.11: *Results when the Lodi algorithm is applied to benchmark data. The column labeled wa/W is the ratio of the average width over the strip width for the particular data set. The column labeled G or NG represents guillotineable or non-guillotineable data sets respectively.*

The Lodi algorithm

The Lodi algorithm, as described in §4.1, requires solving a binary integer linear programming (ILP) problem to select rectangles whose combined width is at most the width of the strip—thus forming a level. Since Visual Basic 6.0 does not have built-in functions to solve ILPs, it was decided to use MatLab 7 [88] for this purpose, since it is possible to use it as a server. This means that it is possible to call MatLab from Visual Basic, solve

the ILP in MatLab and then make the output available to Visual Basic. MatLab has an optimisation toolbox which is able to solve binary ILPs by employing a function *bintprog*. The output of such a function is the objective function value z (the packing height), a list of binary variables x_{ij} (indicating that rectangle L_j should be packed on level i), the number of nodes explored, the number of iterations required and the time expended by branch and bound procedure.

Although widely available software packages, such as MatLab and Microsoft Excel, have built-in functions for solving optimisation problems, Lodi [75] recommended that optimisation oriented software packages should rather be used. This is due to the fact that there were some difficulties encountered in formulating the problem in MatLab format. Unfortunately, the author did not have access to such packages due to financial constraints—therefore MatLab was used instead.

The Lodi algorithm was also tested on the 542 benchmark data sets described in §1.6 and the results are shown in Table 5.11. Only instances where an optimal solution was reached within the imposed time limit are shown with the exception of T_1^3, which is included for illustration purposes. The time limit was set at 7200 seconds and from the results, T_1^3 could not be solved within this time limit. This may be a surprising result, because T_1^3 and T_1^1 were generated in a similar manner and both have the same number of rectangles, and yet T_1^1 could be solved within the time limit. The difference between the two data sets is the ratio of average width of rectangles to the strip width (wa/W). The ratio indicates the number of rectangles that may fit on one level. T_1^1 has a higher value of this ratio than T_1^3, which indicates that in the latter more rectangles may fit on a level—hence more nodes will be generated and the time required for the branch and bound procedure will be longer. Another significant attribute of instances with the same number of rectangles is whether the data sets are guillotineable or not. The instances V_1 and C_1^2 have the same number of rectangles as T_1^1, but since the former two are non–guillotineable data sets, the branch and bound was able to reach a solution within a shorter time period. Another aspect that plays an important role in this analysis is whether the data sets have perfect packings or not. Those with perfect packings have a complex combinatorial structure and are not easy to solve. The non–guillotineable data sets with 15 and 13 rectangles do not have perfect packings but have low average width ratios and therefore smaller numbers of nodes are generated. However, for non–guillotineable data sets with perfect packings and low average width ratio, a large number of nodes is expected.

Contact with MatLab consultants [99] was established to find out why it was taking so long to solve even small instances of 18 rectangles. It is acknowledged that the number of nodes explored by the branch and bound algorithm grows exponentially with the size of the problem instance, but still the expectation was that the results would be achieved within the stipulated time limit. The consultants were still investigating this matter at the time of submission of this dissertation.

When comparing these results to the packing heights obtained by the offline level algorithms that only pack rectangles on the floor, it is observed that the heuristics are able to obtain similar packing heights than the exact algorithm within very short time periods. The heuristics were also able to solve all the instances including those with large lists of 500 rectangles within a few seconds. This is, of course, the reason why heuristics are often preferable to exact algorithms.

Data sets	# of Rec	OPT	Optimal height	Reduction height	z	Branch nodes	Branches	All nodes
U_1	10	-	1 016	167	849	1 356 933	2 294 090	2 445 138
V_1	10	-	23	4	19	178 793	566 944	683 644
V_4	7	-	20	0	20	10722	46 183	47 743
V_7	8	-	20	0	20	221 771	1 135 054	1 224 814
V_{10}	13	-	80	3	77	212 849 280	894 322 149	956 538 849
B_1	10	**40**	**40**	16	24	2 386 470	9 239 638	11 010 128

Table 5.12: *Results when the Martello algorithm is applied to benchmark data. The columns labeled OPT and optimal height denote the known optimal packing height of the data set and that yielded by the Martello algorithm respectively.*

The Martello algorithm

The Martello algorithm, described in §4.2, was also tested on the 542 benchmark instances described in §1.6. Only the few instances shown in Table 5.12 were solved within a time limit of 18 000 seconds. The column labeled *reduction height* indicates the optimal packing height obtained for the sub-instance when the problem was reduced. The total packing height is obtained by adding the reduction height and the value of z, which is the value obtained by the branch and bound algorithm. It is clear, from the result of the B_1 data set, that the algorithm is able to achieve an optimal packing height. The results in this table also indicate the number of branch nodes generated per data set which balloons almost out of control as the number of rectangles in a data set increases. This is a clear indication why exact methods are computationally expensive even though they yield optimal packings.

When comparing the packing height obtained by offline plane heuristics with those obtained by the Martello algorithm for the same test instances, it is observed that the packing heights obtained by the heuristics are very good—and they are obtained within a few seconds.

5.5 Chapter summary

The comparisons carried out in §5.1–§5.3 facilitate ranking of the algorithms in terms of preference starting with the highly recommended and giving only the top three algorithms. Provided the solution quality is deemed more important than execution time, the top three algorithms are the $FCNR_{mod}$, FCNR and SAS algorithms. If packing on the ceiling or stacking of rectangles is not permitted for a particular application (implying that FCNR and SAS algorithms may not be applied) then the top three recommended algorithms are the BFDHDW, FFDHDW, BFDH algorithms. If, on the other hand, the user is interested in obtaining results rapidly, then the algorithms in the best fit and first fit classes are recommended.

Finally, to summarise, this chapter opened with a comparison of packing heights obtained

by known offline packing heuristics (introduced in §2.1) against their respective proposed improvements (introduced in §2.2). The nine best algorithms were then compared in terms of solution quality and frequency with which smallest packing heights were achieved in order to facilitate ranking of the algorithms. Exact algorithms were also tested on the benchmark data and their results were compared with the appropriate heuristics. It was observed that the heuristics yield good results in a fraction of the time expended by the exact methods. Finally, the running times of all the offline algorithms, except that of the KP01 algorithm, were also investigated. The new SAS heuristic was observed to exhibit good performance compared to other algorithms in terms of solution quality, although it requires longer running times and it is ranked third, following the FCNR_{mod} and FCNR algorithms.

Chapter 6

Comparison of online algorithmic results

Analogous to the comparisons carried out in Chapter 5 for offline algorithms, the efficiencies of and solution qualities obtained by the online algorithms presented in Chapter 3 were also compared by applying them to the 542 benchmark instances described in §1.6. Each algorithms' performance was measured by means of the mean packing height obtained by the algorithm in question as well as by the mean execution time, computed over all benchmark data sets. Statistical tools used during the comparison of the algorithms' performance include the student's t-test, ANalyses Of VAriance (ANOVA) and the chi-squared test. All these tests were carried out at a 5% level of significance. The t-test and ANOVA were used to compare the mean packing heights obtained by the algorithms over the 542 instances, while the chi-squared test was used to compare the frequencies with which the smallest packing height was obtained by the various algorithms and to determine whether, statistically, there were any significant differences between these frequencies. Where the results from the ANOVA indicated significant differences, the method of Least Significance Difference (LSD) was employed to determine between which algorithms the differences arose. These statistical tools are fully described in Appendix C.

While testing the online algorithms, it was observed that in most of the 542 data sets, the initial rectangles have larger heights than the rectangles towards the ends of the packing lists. Hence each algorithm was tested three times on each data set, by changing the order in which rectangles enter the system from the data set list—either in the *normal or forward* order, in the *reverse* order and in a *random* order.

6.1 Comparison of online level algorithms

Level algorithms from the literature for *online* packing problems were compared with the suggested modifications in §3.2. The results shown in the first section of Figure 6.1 indicate that the average height in the *forward* traversal order of the benchmark data sets is smaller than in the *reverse* order. This is because, for some reason, packing typically begins with rectangles of greater height in the *forward* order and for those algorithms that allow revisiting of existing levels, the smaller rectangles may be inserted on any available level with sufficient space—thus decreasing the probability of creating new levels in the

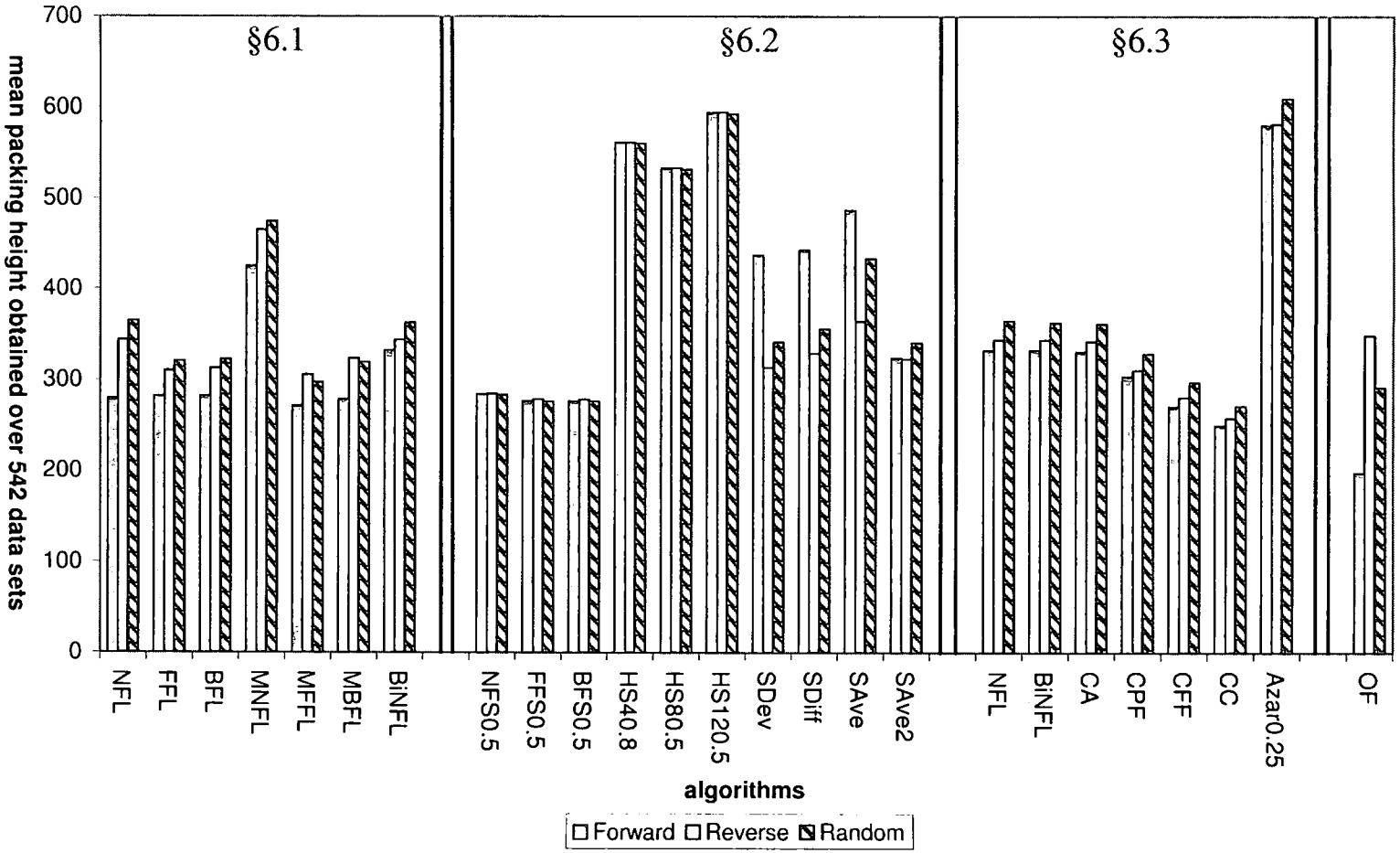

Figure 6.1: *Comparison of the mean packing heights obtained over the 542 benchmark data sets described in §1.6 by the algorithms described in §3.1.1–3.3.2. Each algorithm was tested for three different orders in which rectangles enter the system from the benchmark sets: forward, reverse and random.*

forward order. An ANOVA was carried out separately for each order and in all instances the results revealed that there are significant differences between the average heights obtained for the different algorithms with a fixed traversal order. In all three traversal orders, the suggested MFFL algorithm obtained the smallest average height, although the LSD indicated that there were no significant differences between the mean packing heights obtained by the MFFL, BFL, FFL and MBFL algorithms (indicated by *"no"* entries in Table 6.1). There were significant differences between the means obtained by algorithms that do not revisit existing levels (NFL, MNFL, BiNFL) and those allowing existing levels to be revisited (FFL, BFL, MFFL, MBFL), as expected. The latter group of algorithms achieves better performance based on the mean solution quality obtained.

Further tests were also carried out to determine whether the data set traversal order plays an important role in each algorithm's performance measured. The results shown in Table 6.2 indicate that, in terms of the mean packing height obtained, order does not play a significant role for the NFL and BiNFL algorithms. But when it comes to a frequency analysis, it is only for the MNFL algorithm that traversal order is unimportant (Table 6.2(b)) at a 5% level of significance—for all other algorithms traversal order affected results.

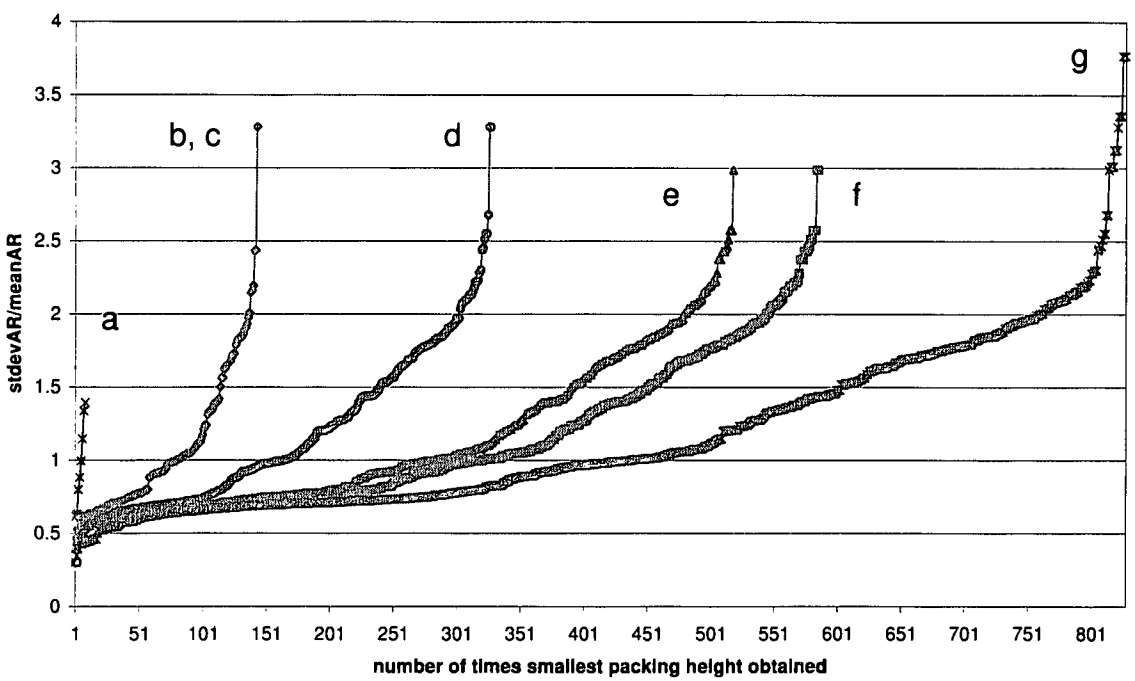

Figure 6.2: *Aspect ratio analysis for the online level algorithms described in §3.1.1: a – MNFL, b,c – NFL and BiNFL, d – MBFL, e – BFL, f – FFL and g – MFFL.*

Another investigation was carried out in terms of the aspect ratios of the 1 626 data sets (a combination of all three traversal orders for all 542 benchmark data sets). Of the 1 626 instances, only instances where an algorithm obtained the smallest packing height were selected and the *standard deviation* (stdevAR) and *mean* (meanAR) of the aspect ratios of the rectangles in these instances were computed. The fraction *stdevAR/meanAR*, known as the *coefficient of variation* (CV), was used to reflect the variation of rectangle aspect ratios relative to the mean. The number of data sets for which an algorithm

obtained the smallest height associated with values of the CV are depicted in Figure 6.2. If, for instance, a value of 3 is selected for the CV, it may be seen in the figure that the BFL, MBFL, FFL and MFFL algorithms were all able to obtain the smallest height, on average, for such a data set. Out of these algorithms, the MFFL algorithm obtained the smallest packing height for the largest number of data sets (825).

An interesting question is the following: when given any data set with a known CV value, which level algorithm should be recommended to give the best solution? To answer this question, the CV values for test instances where each algorithm obtained the smallest packing height were analysed. The objective was to determine a threshold CV value beyond which significant differences occur between frequencies in obtaining the smallest packing height by some of the level algorithms and below which any of the level algorithms may be used. This was achieved by considering a range of CV values starting with the smallest value observed amongst the benchmark data sets and determining the frequency with which each level algorithm obtained the smallest packing height for that particular CV value. Larger CV values within the range were considered at each iteration, and a chi-squared test was performed to determine whether there were any significant differences between the frequencies obtained by the level algorithms. As the value of CV was increased, a point was reached where a slight increment results in significant differences between the frequencies obtained by the level algorithms. Such a point is referred to as the threshold CV value, and this value was found to be 0.429 in the case of level algorithms. This means that for data sets with CV values below the threshold, any of the algorithms may be used, but for CV values greater than the threshold, the MFFL algorithm is recommended.

In terms of the algorithmic frequencies with which the smallest packing height was obtained (which may be seen in the first section of Figure 6.3) the results of the chi-squared test revealed that there were significant differences between those frequencies achieved between every pair of algorithms for a fixed traversal order. The MFFL algorithm achieved the largest frequency in all traversal orders—hence this is statistically the best algorithm within this group of algorithms, at a 5% level of significance.

6.2 Comparison of shelf algorithms

The algorithms discussed in §3.1.2 pose a problem when one attempts to compare them in terms of efficiency, because they depend on the parameter $0 < r < 1$ selected. Over and above this, the HS_{M_r} algorithms also depend on the value of $3 \leq M \leq 12$ chosen. Hence each of the algorithms was applied for the representative values $r = 0.2, 0.5, 0.8$ and $M = 4, 8, 12$ of the parameters, resulting in six classes of the algorithms (NFS_r, FFS_r, BFS_r, HS_{4_r}, HS_{8_r}, HS_{12_r}). An ANOVA was performed on each of these six classes for the three different traversal orders and the results are shown in the first part of Table 6.2. In the NFS_r class, there is no significant difference between $\text{NFS}_{0.5}$ and $\text{NFS}_{0.8}$ algorithms at a 5% level of significance. However, the $\text{NFS}_{0.5}$ algorithm was selected for further comparisons since it achieved a smaller mean packing height over all benchmark data sets. For algorithms whose mean packing heights showed no significant difference, a selection of algorithms to be used for the purposes of further comparison was simply based on the algorithm achieving a smaller mean packing height. Hence the following algorithms were also selected for further comparisons with respect to all traversal orders of the benchmark data sets: $\text{FFS}_{0.5}$, $\text{BFS}_{0.5}$, $\text{HS}_{4_{0.8}}$, $\text{HS}_{8_{0.5}}$, $\text{HS}_{12_{0.5}}$.

Level algorithms

		NFL	FFL	BFL	MNFL	MFFL	MBFL			
NFL	279.426									
FFL	282.215	no								
BFL	282.215	no	no							
MNFL	425.084	yes	yes	yes						
MFFL	271.750	no	no	no	yes					
MBFL	278.584	no	no	no	yes	no				
BiNFL	332.627	yes	yes	yes	yes	yes	yes			

		NFL_R	FFL_R	BFL_R	$MNFL_R$	$MFFL_R$	$MBFL_R$			
NFL_R	343.833									
FFL_R	310.056	yes								
BFL_R	312.650	yes	no							
$MNFL_R$	465.240	yes	yes	yes						
$MFFL_R$	305.780	yes	no	no	yes					
$MBFL_R$	323.679	no	no	no	yes	no				
$BiNFL_R$	343.833	no	yes	no	yes	yes	no			

		NFL_{Rand}	FFL_{Rand}	BFL_{Rand}	$MNFL_{Rand}$	$MFFL_{Rand}$	$MBFL_{Rand}$			
NFL_{Rand}	362.742									
FFL_{Rand}	320.633	yes								
BFL_{Rand}	322.501	yes	no							
$MNFL_{Rand}$	473.743	yes	yes	yes						
$MFFL_{Rand}$	297.316	yes	no	no	yes					
$MBFL_{Rand}$	318.585	yes	no	no	yes	no				
$BiNFL_{Rand}$	362.742	no	yes	yes	yes	yes	yes			

Shelf algorithms

		$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{4_{0.5}}$	$HS_{8_{0.5}}$	$HS_{12_{0.5}}$	SDev	SDiff	SAve
$NFS_{0.5}$	284.532									
$FFS_{0.5}$	276.580	no								
$BFS_{0.5}$	276.499	no	no							
$HS_{4_{0.5}}$	561.443	yes	yes	yes						
$HS_{8_{0.5}}$	333.532	yes	yes	yes	no					
$HS_{12_{0.5}}$	594.578	yes	yes	yes	no	yes				
SDev	437.467	yes	yes	yes	yes	yes	yes			
SDiff	443.317	yes	yes	yes	yes	yes	yes	no		
SAve	486.840	yes	yes	yes	yes	yes	yes	yes	yes	
SAve2	324.343	no	yes	yes	yes	yes	yes	yes	yes	yes

		$NFS_{0.5R}$	$FFS_{0.5R}$	$BFS_{0.5R}$	$HS_{4_{0.5R}}$	$HS_{8_{0.5R}}$	$HS_{12_{0.5R}}$	$SDev_R$	$SDiff_R$	$SAve_R$
$NFS_{0.5R}$	284.638									
$FFS_{0.5R}$	278.716	no								
$BFS_{0.5R}$	278.158	no	no							
$HS_{4_{0.5R}}$	561.443	yes	yes	yes						
$HS_{8_{0.5R}}$	533.532	yes	yes	yes	no					
$HS_{12_{0.5R}}$	594.578	yes	yes	yes	no	yes				
$SDev_R$	313.582	no	no	no	yes	yes	yes			
$SDiff_R$	328.872	yes	yes	yes	yes	yes	yes	no		
$SAve_R$	364.067	yes	yes	yes	yes	yes	yes	yes	no	
$SAve2_R$	323.010	yes	yes	yes	yes	yes	yes	no	no	yes

		$NFS_{0.5Rand}$	$FFS_{0.5Rand}$	$BFS_{0.5Rand}$	$HS_{4_{0.5Rand}}$	$HS_{8_{0.5Rand}}$	$HS_{12_{0.5Rand}}$	$SDev_{Rand}$	$SDiff_{Rand}$	$SAve_{Rand}$
$NFS_{0.5Rand}$	283.514									
$FFS_{0.5Rand}$	276.173	no								
$BFS_{0.5Rand}$	276.174	no	no							
$HS_{4_{0.5Rand}}$	560.310	yes	yes	yes						
$HS_{8_{0.5Rand}}$	531.870	yes	yes	yes	no					
$HS_{12_{0.5Rand}}$	592.719	yes	yes	yes	no	yes				
$SDev_{Rand}$	341.630	yes	yes	yes	yes	yes	yes			
$SDiff_{Rand}$	355.924	yes	yes	yes	yes	yes	yes	no		
$Save_{Rand}$	433.468	yes	yes	yes	yes	yes	yes	yes	yes	
$SAve2_{Rand}$	340.794	yes	yes	yes	yes	yes	yes	no	no	yes

Special case algorithms

		NFL	BiNFL	CA	CPF	CFF				
NFL	332.627									
BiNFL	332.627	no								
CA	331.367	no	no							
CPF	304.215	no	no	no						
CFF	270.604	yes	yes	yes	yes					
CC	249.431	yes	yes	yes	yes	no				

		NFL_R	$BiNFL_R$	CA_R	CPF_R	CFF_R				
NFL_R	343.833									
$BiNFL_R$	343.833	no								
CA_R	342.351	no	no							
CPF_R	310.556	yes	yes	yes						
CFF_R	280.681	yes	yes	yes	no					
CC_R	257.768	yes	yes	yes	yes	no				

		NFL_{Rand}	$BiNFL_{Rand}$	CA_{Rand}	CPF_{Rand}	CFF_{Rand}				
NFL_{Rand}	362.742									
$BiNFL_{Rand}$	362.742	no								
CA_{Rand}	361.416	no	no							
CPF_{Rand}	328.722	yes	yes	yes						
CFF_{Rand}	297.249	yes	yes	yes	no					
CC_{Rand}	271.074	yes	yes	yes	yes	no				

Table 6.1: *LSD results for level, shelf and special case algorithms. A block containing a* "yes" *(*"no" *resp.) indicates that there are (no resp.) significant differences between the means of the algorithms in the corresponding row and column. The subscripts R and Rand refer to the* reverse *and* random *orders of traversing the data sets respectively.*

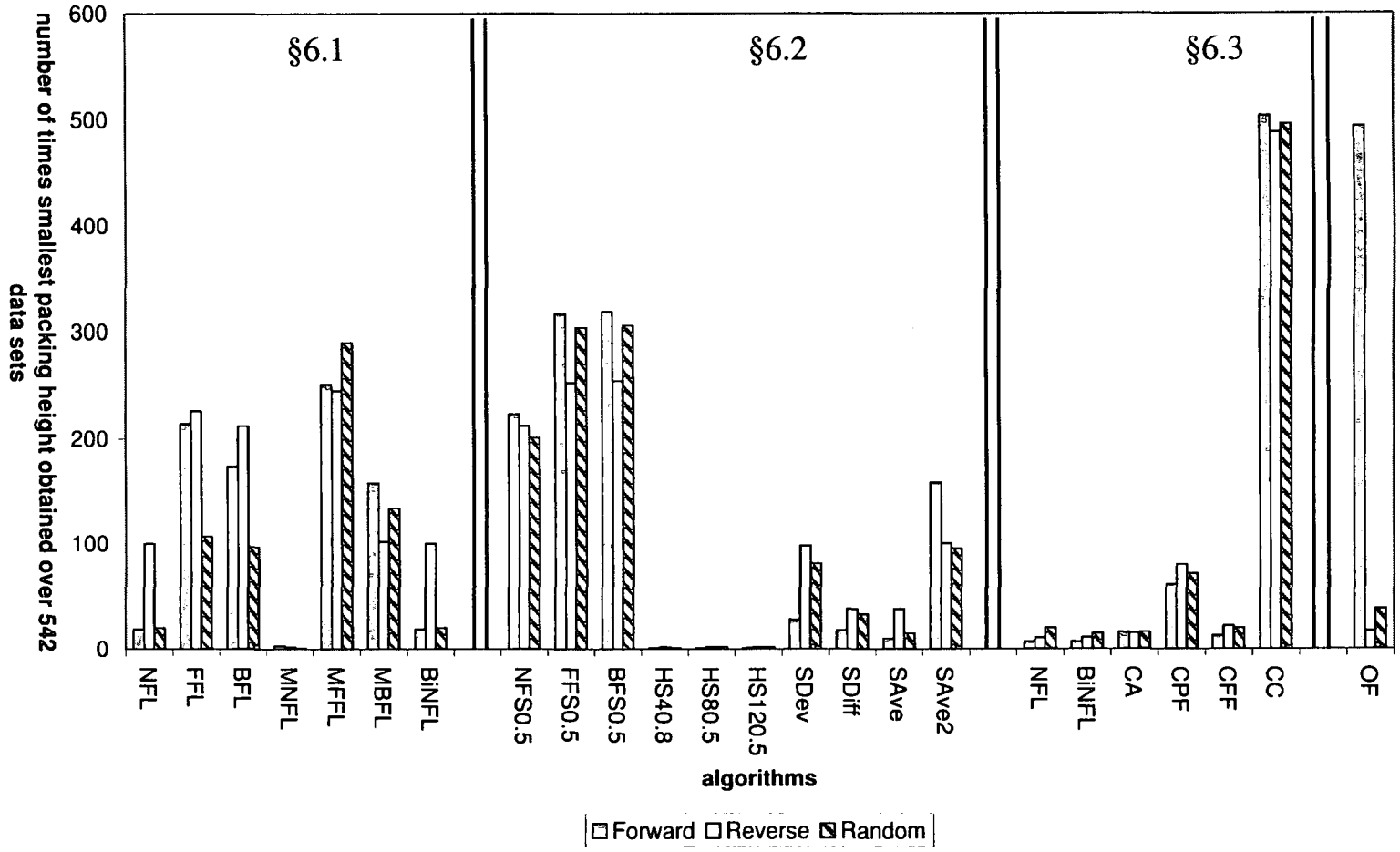

Figure 6.3: *Comparison of the number of times the smallest packing height was obtained for the 542 benchmark data sets described in §1.6 by the various algorithms described in §3.1–3.3.*

Forward Order

		$NFS_{0.2}$	$NFS_{0.5}$			$FFS_{0.2}$	$FFS_{0.5}$			$BFS_{0.2}$	$BFS_{0.5}$			$HS_{4_{0.2}}$	$HS_{4_{0.5}}$			$HS_{8_{0.2}}$	$HS_{8_{0.5}}$			$HS_{12_{0.2}}$	$HS_{12_{0.5}}$
$NFS_{0.2}$	414.002			$FFS_{0.2}$	396.712			$BFS_{0.2}$	400.819			$HS_{4_{0.2}}$	947.618			$HS_{8_{0.2}}$	773.865			$HS_{12_{0.2}}$	847.283		
$NFS_{0.5}$	**284.532**	yes		$FFS_{0.5}$	**276.580**	yes		$BFS_{0.5}$	**278.716**	yes		$HS_{4_{0.5}}$	608.540	yes		$HS_{8_{0.5}}$	533.532	yes		$HS_{12_{0.5}}$	594.578	yes	
$NFS_{0.8}$	336.888	yes	no	$FFS_{0.8}$	333.834	yes	yes	$BFS_{0.8}$	334.377	yes	yes	$HS_{4_{0.8}}$	561.443	yes	no	$HS_{8_{0.8}}$	549.683	yes	no	$HS_{12_{0.8}}$	624.001	yes	no

Reverse Order

		$NFS_{0.2R}$	$NFS_{0.5R}$			$FFS_{0.2R}$	$FFS_{0.5R}$			$BFS_{0.2R}$	$BFS_{0.5R}$			$HS_{4_{0.2R}}$	$HS_{4_{0.8R}}$			$HS_{8_{0.2R}}$	$HS_{8_{0.8R}}$			$HS_{12_{0.2R}}$	$HS_{12_{0.5R}}$
$NFS_{0.2R}$	413.677			$FFS_{0.2R}$	400.819			$BFS_{0.2R}$	398.708			$HS_{4_{0.2R}}$	947.618			$HS_{8_{0.2R}}$	773.865			$HS_{12_{0.2R}}$	847.283		
$NFS_{0.5R}$	**284.638**	yes		$FFS_{0.5R}$	**278.716**	yes		$BFS_{0.5R}$	**278.158**	yes		$HS_{4_{0.5R}}$	608.540	yes		$HS_{8_{0.5R}}$	533.532	yes		$HS_{12_{0.5R}}$	594.578	yes	
$NFS_{0.8R}$	336.838	yes	no	$FFS_{0.8R}$	334.377	yes	yes	$BFS_{0.8R}$	334.275	yes	yes	$HS_{4_{0.8R}}$	561.443	yes	no	$HS_{8_{0.8R}}$	549.683	yes	no	$HS_{12_{0.8R}}$	624.001	yes	no

Random Order

		$NFS_{0.2Rand}$	$NFS_{0.5Rand}$			$FFS_{0.2Rand}$	$FFS_{0.5Rand}$			$BFS_{0.2Rand}$	$BFS_{0.5Rand}$			$HS_{4_{0.2Rand}}$	$HS_{4_{0.5Rand}}$			$HS_{8_{0.2Rand}}$	$HS_{8_{0.5Rand}}$			$HS_{12_{0.2Rand}}$	$HS_{12_{0.5Rand}}$
$NFS_{0.2Rand}$	414.212			$FFS_{0.2Rand}$	395.871			$BFS_{0.2Rand}$	400.819			$HS_{4_{0.2Rand}}$	947.669			$HS_{8_{0.2Rand}}$	773.975			$HS_{12_{0.2Rand}}$	847.150		
$NFS_{0.5Rand}$	**284.263**	yes		$FFS_{0.5Rand}$	**276.922**	yes		$BFS_{0.5Rand}$	**278.716**	yes		$HS_{4_{0.5Rand}}$	608.610	yes		$HS_{8_{0.5Rand}}$	533.671	yes		$HS_{12_{0.5Rand}}$	594.659	yes	
$NFS_{0.8Rand}$	335.585	yes	no	$FFS_{0.8Rand}$	333.832	yes	yes	$BFS_{0.8Rand}$	334.377	yes	yes	$HS_{4_{0.8Rand}}$	561.499	yes	no	$HS_{8_{0.8Rand}}$	549.907	yes	no	$HS_{12_{0.8Rand}}$	624.136	yes	yes

Analysis of variance

	Level algorithms							Shelf algorithms										Special case algorithms					
	NFL	FFL	BFL	MNFL	MFFL	MBFL	BiNFL	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{4_{0.5}}$	$HS_{8_{0.5}}$	$HS_{12_{0.5}}$	SDev	SDiff	SAve	SAve2	NFL	BiNFL	CA	CFF	CFF	CC
Forward	279.426	282.215	282.215	425.084	271.750	278.584	332.627	284.532	276.580	276.499	561.443	533.532	594.578	437.467	443.317	486.840	324.343	332.627	332.627	331.367	304.215	270.604	249.431
Reverse	343.833	310.056	312.650	465.240	305.780	323.679	343.833	284.638	278.716	278.158	561.443	533.532	594.578	313.582	328.872	364.067	323.010	343.833	343.833	342.351	310.556	280.681	257.788
Random	364.898	320.633	322.501	474.483	297.316	319.285	364.898	283.514	276.173	276.174	560.310	531.870	592.719	341.630	355.924	433.468	340.794	362.742	362.742	361.416	328.722	297.249	271.074
$F_{calculated}$	1.844	**4.277**	**4.144**	**3.586**	**3.234**	**6.158**	1.583	0.002	0.0127	0.008	0.002	0.003	0.004	**23.497**	**18.825**	**17.161**	0.767	1.583	1.583	1.644532	1.224979	1.585	1.102
$F_{critical}$	3.001	**3.001**	**3.001**	**3.001**	**3.001**	**3.001**	3.001	3.001	3.001	3.001	3.001	3.001	3.001	**3.001**	**3.001**	**3.001**	3.001	3.001	3.001	3.001	3.001	3.001	3.001

Chi-squared test

	NFL	FFL	BFL	MNFL	MFFL	MBFL	BiNFL	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{4_{0.5}}$	$HS_{8_{0.5}}$	$HS_{12_{0.5}}$	SDev	SDiff	SAve	SAve2	NFL	BiNFL	CA	CFF	CFF	CC
Forward	19	214	174	3	251	158	19	223	317	319	1	1	1	28	18	10	158	7	7	16	61	13	504
Reverse	100	226	212	2	245	102	100	212	252	254	2	2	2	98	38	38	100	11	11	15	80	22	488
Random	20	107	97	1	200	134	20	201	304	306	1	2	2	81	33	15	95	9	9	16	71	20	496
$\chi^2_{calculated}$	**93.251**	**47.082**	**42.646**	1.000	4.557	**12.020**	**93.257**	1.142	**8.131**	**8.075**	0.500	0.400	0.400	**38.638**	**7.303**	**21.238**	**20.844**	0.889	0.889	0.043	2.557	2.436	0.258
$\chi^2_{critical}$	5.990	**5.990**	**5.990**	5.990	5.990	**5.990**	5.990	5.990	5.990	5.990	5.990	5.990	5.990	**5.990**	**5.990**	**5.990**	5.990	5.990	5.990	5.990	5.990	5.990	5.990

Table 6.2: *(a) LSD results for the shelf algorithms described in §3.1.2 for the values $r = 0.2, 0.5, 0.8$ and $M = 4, 8, 12$. (b) Results from the ANOVA and chi-squared test for the level, shelf and special case algorithms obeying the* tetris *constraint. Bold faced entries indicate that significantly different results were achieved for different traversal orders of the data sets.*

The selected shelf algorithms were compared with each other and with the new shelf algorithms in terms of the average packing height obtained and the results are shown in the second section of Figure 6.1. The results indicate that, in terms of the mean packing height obtained, the $NFS_{0.5}$, $FFS_{0.5}$ and $BFS_{0.5}$ algorithms achieved the best performance, followed by the new SAve2, SDev, SDiff and SAve algorithms.

Considering the algorithms individually and comparing the packing heights obtained per traversal order, results from the ANOVA indicated that there are no significant differences at a 5% level of significance over the three data set traversal orders, except with the SDev, SDiff and SAve algorithms. The results shown in Table 6.2 indicate that these three algorithms perform better in the *reverse* order, indicating that the order in which rectangles enter the system plays a vital role with respect to their performance. This was an expected result, because the SDev and SDiff algorithms rely on the first rectangle packed and ideally this rectangle must have the smallest height possible. Based on observation, the majority of the benchmark data sets in *reverse* order start with rectangles of relatively small height—therefore leading to small increments of each shelf height with an overall smaller total packing height. The mean packing heights obtained by the $NFS_{0.5}$, $FFS_{0.5}$ and $BFS_{0.5}$ algorithms were not expected to be similar, because a rectangle is classified according to its height, but depending on the widths of the rectangles that are packed first, it may sometimes be necessary to create an extra shelf of appropriate height due to the insufficient space on shelves of appropriate height. The HS_{M_r} algorithms, on other hand, were expected to yield similar mean packing heights regardless of the traversal order, because the algorithm takes both height and width of the rectangles into consideration before packing on a level.

Moving on to the frequency with which algorithms obtained the smallest packing height, the results of the chi-squared test indicated that only the HS_{M_r} algorithms achieved no significant frequency difference (as r varies), as illustrated in Table 6.2(b). The shelf algorithms with parameter r achieve the largest frequency, followed by SAve2, SDev, SDiff and SAve algorithms (see Figure 6.3). A threshold CV value of 0.39 was computed for shelf algorithms. The $FFS_{0.5}$ algorithm is recommended for use when dealing with data sets with a CV larger than this threshold value.

6.3 Comparison of special case algorithms obeying the tetris constraint

Because the $Azar_Y$ algorithm depends on the threshold constant $0 < Y < 1/2$, three instances were compared with $Y = 0.2, 0.25, 0.3$ to determine only one instance that may be used for further comparisons with other algorithms obeying the tetris constraint. An ANOVA was carried out and the results revealed that there were no significant differences between the mean packing heights obtained by these three algorithmic instances. The $Azar_{0.25}$ algorithm was selected, because after carrying out the chi-squared test, there were significant differences between the frequencies in obtaining the smallest packing heights, showing that the $Azar_{0.25}$ algorithm achieved the largest frequency of 297. As shown in the third section of Figure 6.1, the $Azar_{0.25}$ algorithm performs poorly by achieving very high mean packing heights compared to the other special case algorithms and as such it was excluded from further comparisons because it would always be outperformed by the other algorithms.

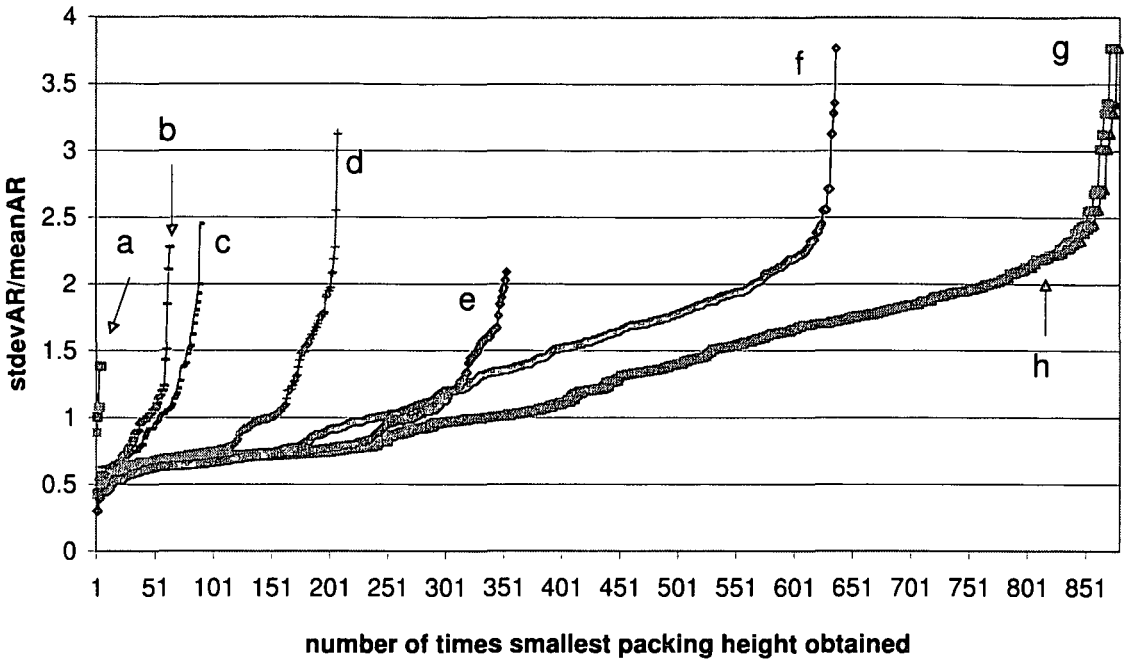

Figure 6.4: *Aspect ratio analysis for the shelf algorithms described in §3.1.2: a* – $HS_{8_{0.5}}$, $HS_{12_{0.5}}$, $HS_{4_{0.8}}$, *b* – *SAve*, *c* – *SDiff*, *d* – *SDev*, *e* – *SAve2*, *f* – $NFS_{0.5}$, *g* – $FFS_{0.5}$ *and h* – $BFS_{0.5}$.

Three versions of the CPF, CFF and CC algorithms are proposed for use when dealing with floating point data. These versions are analogous to those suggested for the Burke algorithm in §5.2.2, since these algorithms also employ a linear array, requiring data to be integral. The first version involves rounding the dimensions (up or down) to the nearest integer, which may not necessarily result in a feasible packing because some rectangles whose dimensions are rounded down may not fit into the packing solution, but the practice of rounding nevertheless maintains the characteristics of the data. The second version wastes space by rounding up the dimensions of all the rectangles to the nearest integers, thereby creating a feasible packing for the original rectangles. The third version involves rounding down the dimensions of all the rectangles, which means that the original rectangles will not fit into the packing solution since their rounded dimensions are smaller than the original data. As expected, the algorithm with rounding down achieved smaller packing heights, however, the results from the ANOVA indicated that at 5% significance level, there was no significant difference. The algorithms with rounding down also obtained the largest frequency with which the smallest packing height was obtained and the chi-squared test indicated that there were significant differences between the different rounding techniques. Based on this outcome, all CPF, CFF and CC algorithmic results considered for further comparisons employ the rounding down technique.

The results shown in the third section of Figure 6.1 indicate that the CC algorithm obtained the smallest mean packing height. An ANOVA was carried out separately for each algorithm to decide whether the traversal order in which rectangles enter the system affects the performance of an algorithm. The results shown in Table 6.2(b) indicate that there are no significant differences between the mean packing heights obtained per

traversal order by each algorithm. Comparing the frequencies of obtaining the smallest packing height separately for each algorithm, the results from the chi-squared test (Table 6.2(b)) revealed that only the CPF algorithm is affected by the order in which rectangles enter the system, achieving the largest frequency in the *reverse* traversal order.

When comparing all the algorithms obeying the tetris constraint, in terms of the mean packing height obtained over all the 542 test instances, the results of the ANOVA indicate that there is a significant difference. The results from the LSD (Table 6.1) suggest that the newly proposed CC and CFF algorithms are the best performing algorithms with no distinguishable difference between the mean packing heights obtained. However, in terms of the frequency of obtaining the smallest packing height, the two algorithms are distinguishable at a 5% level of significance with the CC algorithm achieving the largest frequency, as may be seen from the chi-squared test.

A threshold value of 0.446 was computed for the CV, implying that for data sets with CV values smaller than the threshold, any of the special case algorithms may be used, but for values of CV larger than the threshold, the CC algorithm is recommended. This is shown in Figure 6.5 where the CC algorithm obtained the smallest packing height for over 1 400 instances out of a total of 1 626 instances.

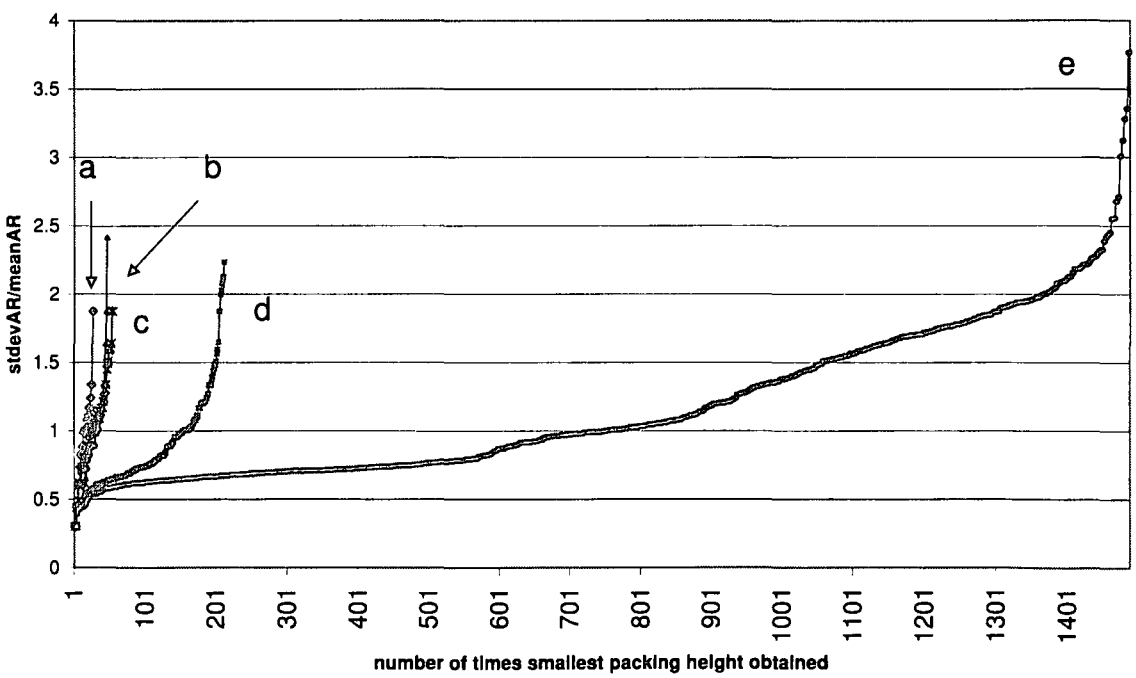

Figure 6.5: *Aspect ratio analysis for the special case algorithms described in §3.1.2: a – BiNFL and NFL, b – CFF, c – CA, d – CPF and e – CC.*

6.4 The OF algorithm

The OF algorithm, as described in §3.3.2, is a plane algorithm utilising a linear array to search for appropriate locations to pack a rectangle. When using the ANOVA to test whether the three traversal orders played any role in the packing height obtained,

the result $F_{calculated}(37.338) > F_{critical}(3.001)$ was obtained. This result indicates that there is a significant difference between the packing heights obtained in each traversal order, with the forward order achieving the smallest mean packing height (Figure 6.1). Using the chi-squared test, the result $\chi^2_{calculated}(794) > \chi^2_{critical}(5.99)$ indicates that there is a significant difference at a 5% significance level between the frequency with which the smallest packing height was obtained in each traversal order (Figure 6.3). The OF algorithm is the only heuristic in this dissertation that was able to achieve an optimal packing for all instances of N1, N2, N3, N4, T1, T3 and T4 benchmark data sets in the forward order. These results have been indicated by the bold face entries in the last column of Table D.1.

6.5 Comparison of execution times

The execution times of the online algorithms, when applied to the benchmark instances described in §1.6, are shown in Tables D.4–D.6 with each table representing the three traversal orders *forward, reverse* and *random* respectively. For the level algorithms, the group of algorithms FFL, MFFL, BFL and MBFL, considered to have achieved better performances in terms of smallest packing height obtained, require correspondingly longer execution times than the poorer performing group of NFL, MNFL and BiNFL algorithms. This is an expected result, because the former group of algorithms involve searching the strip from the bottom upwards for a level with sufficient space and this is time consuming—particularly for a large number of levels. Ideally the best performing algorithm should achieve the smallest packing height in the quickest time. However, the results indicate that a trade-off exits between algorithms that yield good solutions, but take longer to execute, and algorithms yielding solutions of lesser quality, but achieving faster execution times.

Turning to the shelf algorithms, the $FFS_{0.5}$ and $BFS_{0.5}$ algorithms require longer running times than the $NFS_{0.5}$ algorithm, as expected, because the former algorithms search existing levels for sufficient space, while the latter takes only one shelf into consideration at each packing stage. Amongst the newly developed shelf algorithms, the SDev and SDiff algorithms require longer execution times than the SAve and SAve2 algorithms. This may be due to the fact that in the SDev and SDiff algorithms, the appropriate shelf type has to be determined first, and once found, the shelves of this type have to be searched for sufficient packing space, whilst in the SAve and SAve2 algorithms the shelves are just searched for sufficient space.

For the special case algorithms obeying the tetris constraint, the newly proposed improvements, namely the CC, CPF and CFF algorithms, require longer execution times than their counterpart, the CA algorithm. This was an expected result, because the latter only takes a few patterns into consideration, while the other three algorithms take more patterns into consideration—therefore resulting in longer running times. This result indicates that this model shows potential with respect to yielding good solutions.

The execution times required by the OF algorithm increases as the number of rectangles to be packed increases. The mean execution times shown in the tables, in most cases exceed those of the other online algorithms. This was an expected result, because in the OF algorithm, a search for sufficient space is involved due to the empty areas and linear arrays searched.

6.6 Chapter Summary

In this chapter, online level algorithms from the literature (see §3.1.1) were compared with their proposed modifications in §3.2.1. The algorithms were compared in terms of the mean packing heights achieved over the 542 data sets described in §1.6 as well the frequency with which the smallest packing height is obtained for these benchmark data. The execution times of the algorithms were also compared. The classical shelf algorithms were also compared to new shelf algorithms that use a different technique in creating additional space within a shelf. The new shelf algorithms exhibit a very good performance. Special case algorithms obeying the tetris constraint were also compared to some newly proposed algorithms that take more patterns into consideration than those found in the literature and these yield very good results. The downside to these proposed algorithms is that they require rectangle dimensions to be integers, but this problem may be resolved by rounding. Finally, a new plane algorithm, the OF algorithm, was also tested using the three traversal orders and in the forward order it was able to achieve the optimal packing height for a few benchmark instances.

The purpose of comparing the algorithms described in Chapter 3 is to facilitate an informed decision as to which is the best algorithm to use, given a new data set. If the user is interested in online level algorithms, the newly proposed MFFL algorithm is the first choice followed closely by the FFL algorithm and finally by the BFL algorithm. These form the top three recommended online level algorithms. In the class of shelf algorithms, the top three are the NFS_r, FFS_r and BFS_r algorithms. However, the newly proposed SAve2 and SDev algorithms also perform very well, with the advantage that they do not depend on any user defined parameter. This advantage is due to the fact that a poor choice of such a parameter value may potentially lead to a poor performance by the NFS_r, FFS_r and BFS_r algorithms. The newly proposed shelf algorithms, on the other hand, incorporate a different technique of employing the history of already packed rectangles. For applications that obey the tetris constraint, the results indicate that the newly proposed CC algorithm, which takes more patterns into consideration than the original CA algorithm, is the first choice. This is followed by the CFF and CA algorithms.

Chapter 7

A decision support system

Whether in industry or in every day life, decisions have to be made on how to utilise space or material efficiently in applications such as the paper industry where different shapes and sizes are produced with minimum resource. This is the motivation behind developing an active computerised decision support system (DSS) for the strip packing problem, which involves deciding on the placement of items such that the minimum packing height is attained. Such a system is not only expected to aid managers in finding good solutions, but will also provide a visual display on where to pack each item. In industry, a DSS is particularly important in the sense that it is typically able to prompt high quality decisions within a short space of time. This is expected to lead to higher customer satisfaction, increased productivity and reduction in costs.

Computerised systems are typically employed for *speedy computations, technical support* and *quality support* [112]. In terms of speedy computations, it is known that computers are able to perform highly complex computations within a short time period. Technical support refers to the ability to work with data sets which can be stored, processed and retrieved at any time. Finally, quality support is concerned with the ability to make an assessment over different scenarios or conditions and rapidly obtaining results which lead to improved decisions. This chapter serves to describe the various components of such a decision support system called the *strip packing decision support system* (SPDSS).

7.1 The strip packing decision support system

The SPDSS was developed using Visual Basic 6.0. All algorithms included in the SPDSS are able to solve strip packing problems of the form

2D	R	SP	*	MiS	0,*,*,*

approximately. These problems are two dimensional and rectangles that have to packed may not be rotated. SPDSS was developed such that it is general and not tailored to solve any particular strip packing problem. It was also designed to be simple and user friendly. It is not web based as it is developed for use by one user on a personal computer and like any other system it comprises three main components, an *input component, a processes component* and an *output component*. These three components are shown schematically in the form of a flow chart in Figure 7.1. In the flow chart, data comprising each rectangle's

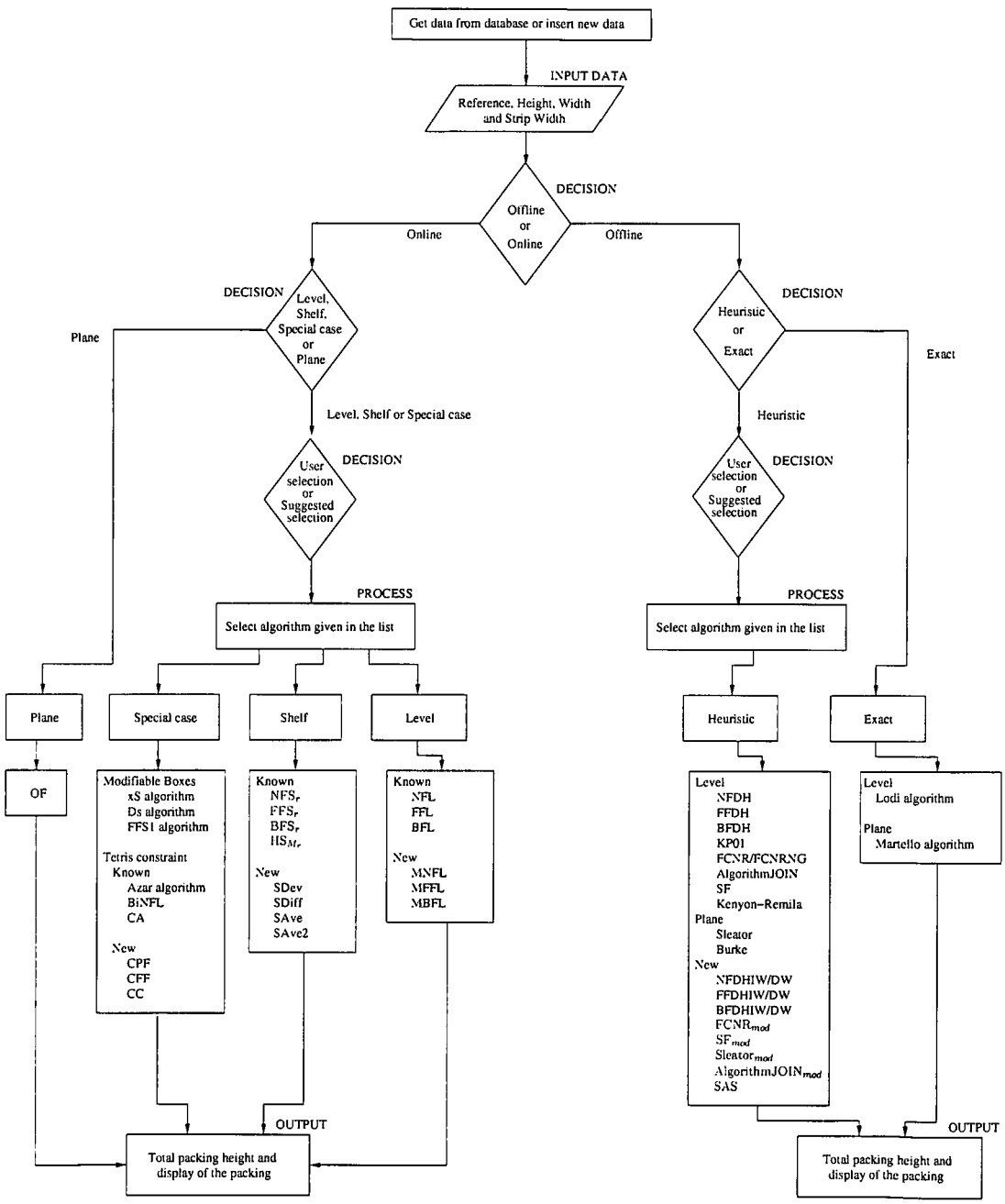

Figure 7.1: *Flow chart of the SPDSS showing the three components, input, process and output.*

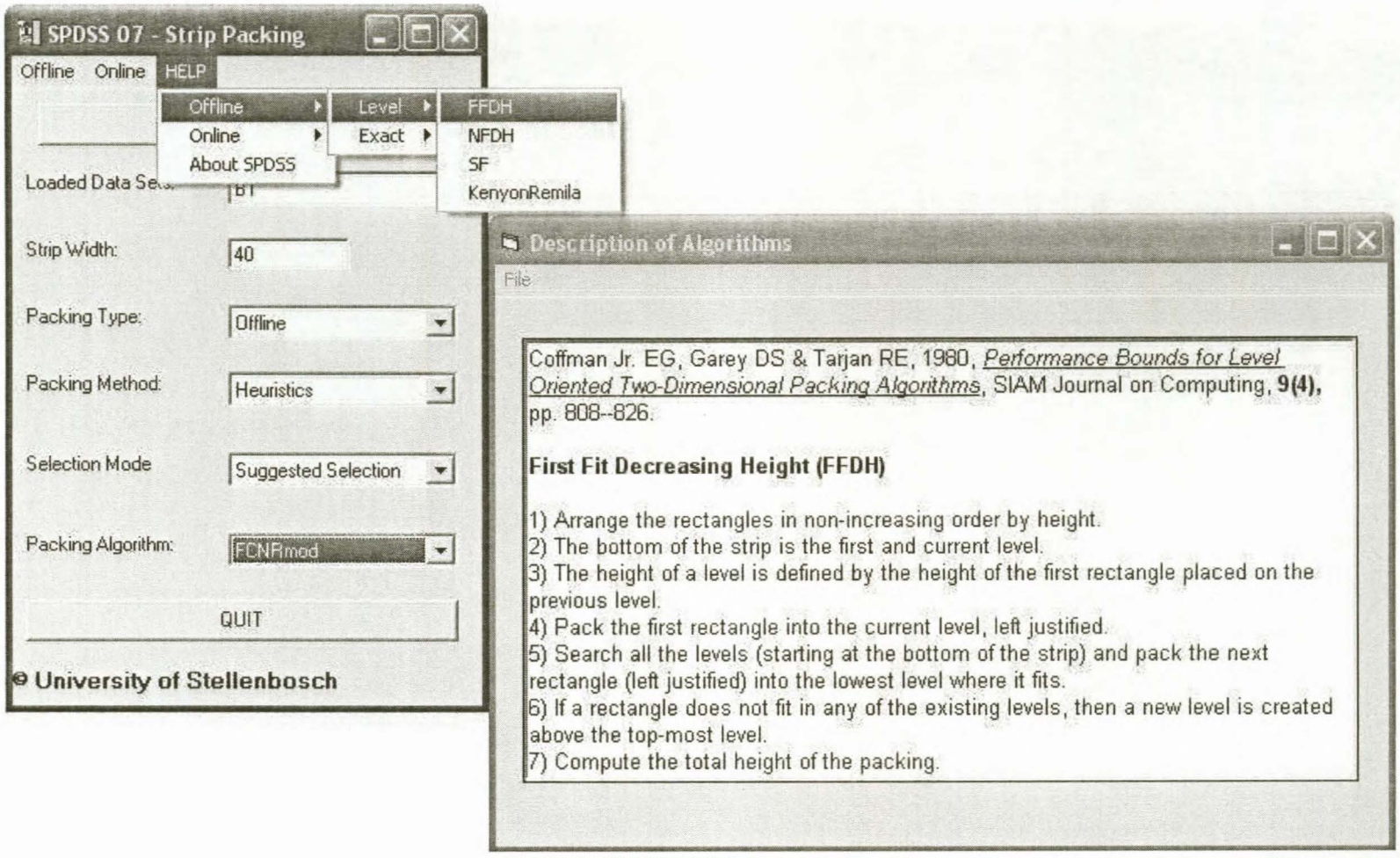

Figure 7.2: *SPDSS main user interface. The user has retrieved the B_1 data set with a strip width of 40 units, and has indicated that an offline packing problem should be solved by means of a suggested list of heuristics. The list of recommended algorithms appears in the combo box labelled packing algorithm. Clicking the help button, a list of algorithms emerges, which upon selection, the algorithm description appears.*

description and the strip width are required. These are the input data and once it has been selected, the user has to decide whether the problem under consideration is an offline or online packing problem. If it is an offline (online, resp.) packing problem, then another decision has to be made regarding which packing method, heuristics or exact (level, shelf, special case or plane resp.)—should be used. If exact (plane, resp.) is selected, then the list of algorithms is displayed corresponding to these methods. If one of the other remaining methods in either category is selected, then the user has to make one more decision about the particular algorithm to be employed: either to choose from a recommended list (denoted by *suggested selection*) or to choose from the entire list (denoted by *user selection*) of algorithms. Once an algorithm is selected, the total packing height and the packing pattern are displayed.

The main user interface, shown in Figure 7.2, is based on the subtypology described in Chapter 1. When the user clicks on the menu item *Help* within this window, the entire list of algorithms appears and under each algorithm a full description of the various steps of the algorithm is displayed, as shown in the *Description of Algorithms* window of Figure 7.2. For the system to function, the user first has to load a data set and enter the strip width, and then select different fields representing the properties of the particular strip packing problem being dealt with. Finally, a selection of an algorithm to solve such an instance is carried out.

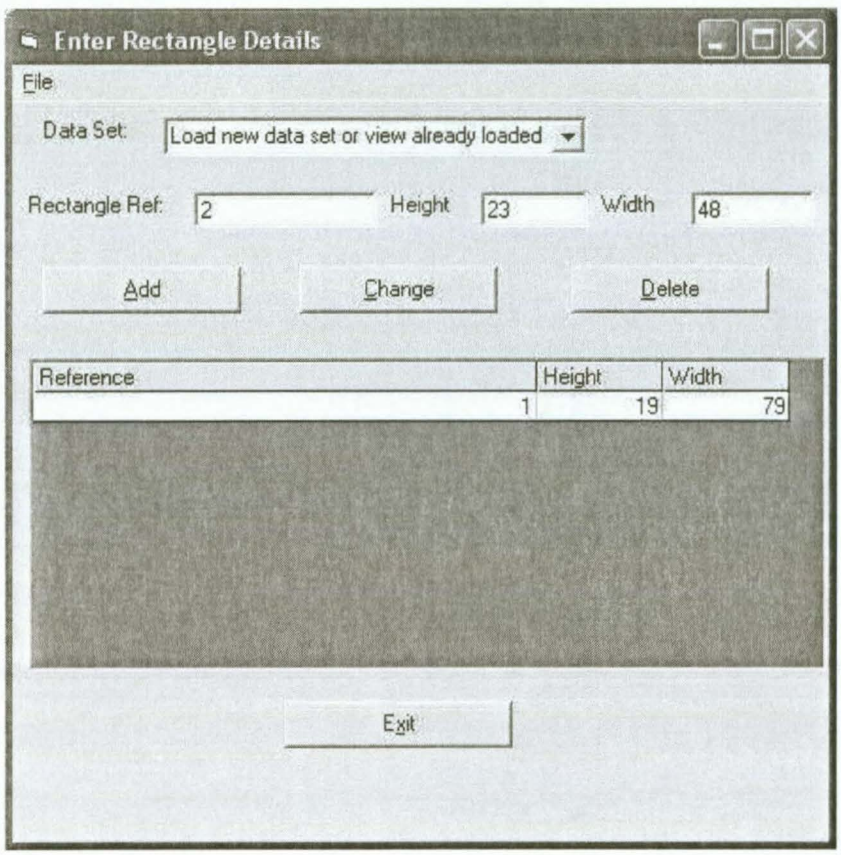

Figure 7.3: *Interface for loading new data sets or maintaining data. In this screen shot, the user is about to add rectangle 2 to a new list being created.*

7.1.1 Input

The input data required by SPDSS is the reference, height and width of the rectangles. The rectangle reference is a label assigned to each rectangle so that rectangles may be distinguished. The program allows the strip width to be entered by the user and it must be a positive integer while the rectangle height and width may be floating point data. To load a new data set, view dimensions of already loaded data sets or simply to maintain the data, the user may click on the button *Enter or Maintain Data* which appears on the main user interface. Such an action causes the *Enter Rectangle Details* window (shown in Figure 7.3) to open, where the user can load new data sets or view the dimensions of data sets already loaded by clicking on the combo box labelled *Data Set.* When the user clicks

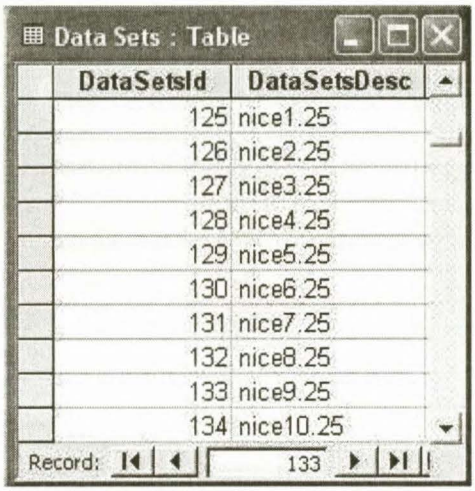

Figure 7.4: *Database table of the loaded data sets. An example of how the Mumford data [94] are stored in the data base, is shown in this figure. Note that a unique key is assigned to each data set.*

on the menu item *File*, four options appear, namely, *Get Excel Data, Get Text Data, Save Excel Data* and *Save Database.* In the case of loading new data sets, they may be loaded from either an Excel file, a text file or entered directly on the grid provided. SPDSS can only read Microsoft Excel [90] data with references in the first column, the rectangle height in the second column and finally rectangle width in the third column. The text data should follow the same format, but the entries on each line must be separated by commas. When loading new data on the grid, the user must click the *Add* button, each time entering dimensions of a single rectangle in the text boxes labelled *Rectangle Ref, Height, Width* until all the rectangles have been loaded. The program also allows the user to change or delete data entries by first selecting a particular row and then clicking either *Change* or *Delete* buttons. When loading new data sets, the user has to save the loaded data sets to a data base. The option *Save Excel Data* may be clicked when the user wants to create an Excel data file of the dimensions just entered on the grid. Part [101] suggested the use of a database to handle the data. Some of the reasons for this suggestion include: (1) it is faster than using an external file, (2) it can automatically sort the data when dealing with algorithms that require pre-sorting of the data and (3) all data may be grouped together (*e.g.* multiple data sets are contained within one location

Rectangles : Table

ProjectId	RectangleId	Reference	Height	Width
125	8265	1	28.48085	31.94849
125	8266	2	46.53127	17.06567
125	8267	3	31.32587	21.41732
125	8268	4	29.71539	19.61116
125	8269	5	19.24619	29.41763
125	8270	6	31.32587	17.67189
125	8271	7	46.53127	11.89663
125	8272	8	25.32328	19.61116
125	8273	9	20.65838	19.47804
125	8274	10	22.59542	16.75077

Record: 1 of 111431

Figure 7.5: *Database table of rectangle dimensions stored. The records shown in this section of the table represent those of the nice1.25 data set. The field DataSetId links the data sets and rectangles tables. Since DataSetId is 125, the dimensions shown are for the nice1.25 data set, which is the top-most record in Figure 7.4.*

as opposed to having multiple excel files containing each data set). The SPDSS uses the Microsoft Access [90] database to store data. This program was chosen, because it is believed that the majority of potential users will have access to the Microsoft office suite on their personal computers. It was also selected because it may be linked to Visual Basic 6.0 for easy access of the data. The database table shown in Figure 7.4 has two fields, *DataSetId* and *DataSetsDesc* representing the key and labels assigned to each loaded data set respectively. This table is used to store the labels of the various data sets. The rectangle dimensions are stored in the *Rectangles* table shown in Figure 7.5.

PackingTypes : Table

	PackTypeId	PackingType
+	1	Offline
+	2	Online
*	(AutoNumber)	

Record: 2

(a)

PackingMethods : Table

PackTypeId	PackMethodId	PackMethod
1	1	Heuristics
1	2	Exact
2	8	Shelf
2	9	Level
2	10	Special Case
2	11	Plane
0	(AutoNumber)	

Record: 7 of 7

(b)

Figure 7.6: *Database table of packing types and packing methods.*

7.1.2 Processes

Relational tables are created in the database and linked to the main user interface. These relational tables are used when the user selects the *packing type, packing method, selection mode* and *packing algorithm* shown in Figures 7.6–7.8 respectively. Packing type refers to either an *offline* or an *online* regime described in §1.2 and it is the fourth field in the subtypology presented in §1.5.

Each of these packing types are associated with various packing methods that are shown in Figure 7.6(b), namely, heuristics, exact, shelf, level, special case and plane algorithms. The field labelled *PackTypeId* in this table indicates that offline types are associated with heuristics and exact methods, while online types are associated with shelf, level, special case and plane methods. All these packing methods were described fully in Chapters 3–4. Once the user has selected the packing type and packing method, the list of algorithms to choose from depends on what is termed *selection mode*. This refers to displaying algorithms that have been recommended in Chapters 5 and 6, referred to as *Suggested Selection*, in the table or just to provide the entire list of algorithms falling within the chosen category denoted as *User Selection*. Finally, the user is required to select one of

SelectionMode : Table

	PackMethodId	AlgorithmSelectionID	Selection Mode
+	1	1	User Selection
+	1	2	Suggested Selection
+	8	3	User Selection
+	8	4	Suggested Selection
+	9	5	User Selection
+	9	6	Suggested Selection
+	10	7	User Selection
+	10	8	Suggested Selection
+	2	9	User Selection
+	11	10	User Selection
▶	0	(AutoNumber)	

Record: 11 of 11

Figure 7.7: *Database table of the selection mode where the user may select suggested algorithms or select from an entire list of algorithms provided.*

the displayed algorithms in the combo box labelled packing algorithms.

7.1.3 Output

Upon selecting an algorithm, SPDSS was designed to display the resulting packing along with a numeric value of the total packing height achieved, as shown in Figure 7.9. Also included is a label of the algorithm used and, as shown in the figure, FCNR_{mod} algorithm was used. The displayed packing is also calibrated with tick marks for easy access to the measurements. For data sets with a large number of rectangles, the displayed packing becomes cluttered and it is not easy to see the packing locations of each rectangle—this problem was solved by including a grid and a zooming feature. The grid is used

PackingAlgorithims : Table

AlgorithmSelectionID	AlgorithimId	AlgorithimDesc
1	110	FFDH
1	111	NFDH
1	112	SF
1	114	NFDHDW
1	115	NFDHW
1	116	FFDHW
1	117	FFDHDW
1	118	SAS

Record: 1 of 66

Figure 7.8: *Database table of the various packing algorithms described in Chapters 2–4.*

when dealing with algorithms that partition the strip into horizontal segments regardless of whether they are online or offline. These include level, shelf, special case and some exact methods whereby the rectangles packed on each level are shown according to their references. Each row of the grid represents rectangles packed from the left to the right of the strip on each level or shelf. The number of rows in the grid depends on the number of levels or shelves created and the number of columns depends on the number of rectangles packed on each level. The grid appears in its own window to allow the user to view both the packing pattern and the grid simultaneously. The zooming feature is applied to all packing methods and the user is able to zoom in and out by clicking on the second horizontal scroll bar shown in Figure 7.9.

7.2 Chapter Summary

This chapter served as a description of the active decision support system referred to as SPDSS, developed in Visual Basic 6.0 for the 2D strip packing problem. First a flow chart of the working of the entire system was shown, depicting the various components of the model. Various screen shots of the different components of the decision support system were also shown and described. The system incorporates all the algorithms described in Chapters 3–4. The results based on the comparisons carried out in Chapters 5 and 6 are also incorporated by means of an option whereby the user is able to select an algorithm from a recommended list as opposed to selecting an algorithm from the entire list of algorithms. SPDSS is simple, user friendly and is expected to aid managers in making rapid, informed decisions with respect to suitable packing patterns.

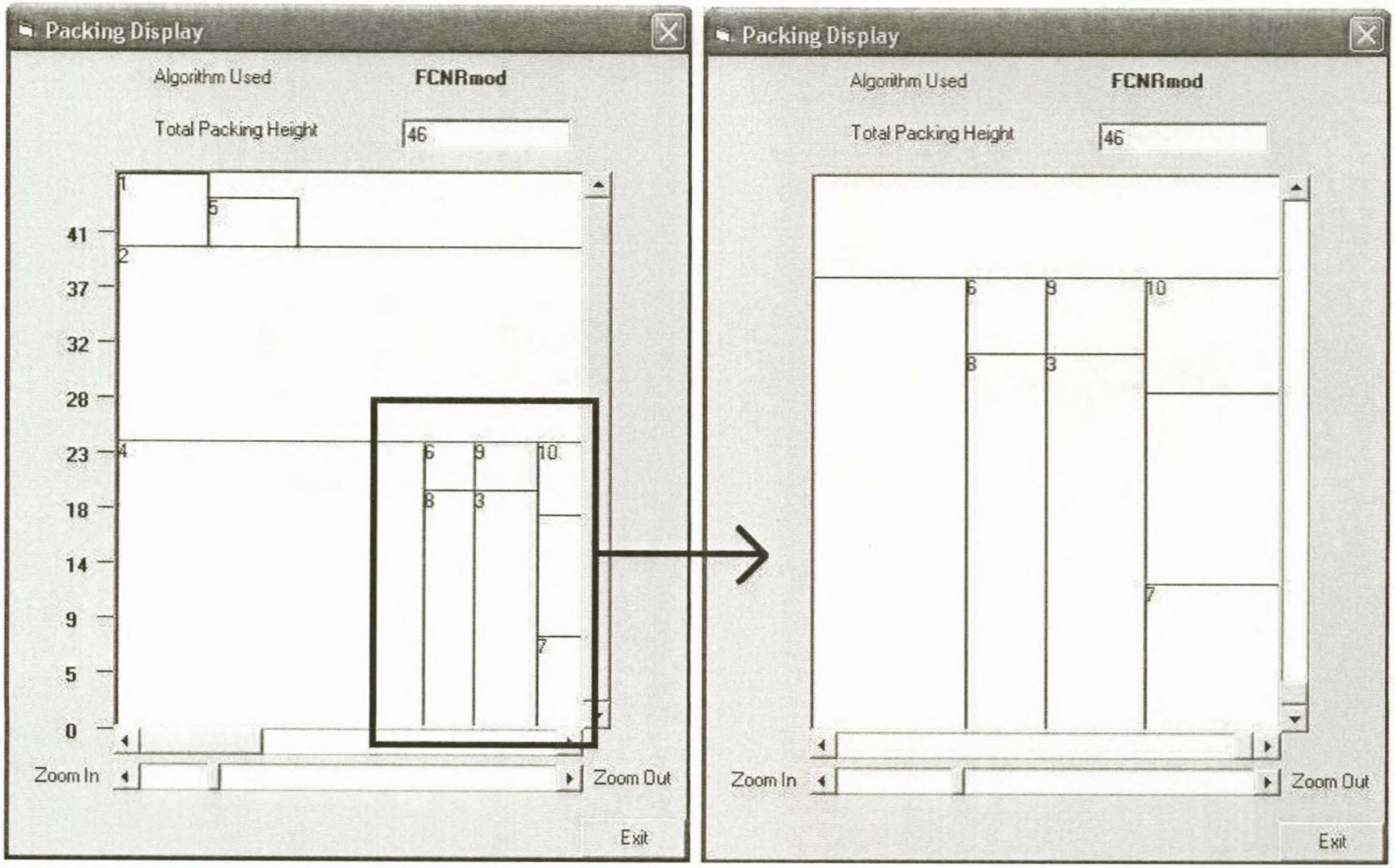

Figure 7.9: *Display screen of the total packing height of 46 units achieved by the $FCNR_{mod}$ algorithm when applied to the B_1 data set, as well as the corresponding packing pattern. The enlargement of the area indicated by the bold rectangle is shown on the right-hand side.*

Chapter 8

Conclusion and Future Work

An overview of the work carried out in this dissertation is presented in §8.1. Some recommendations about possible future work follow in §8.2.

8.1 Dissertation Summary

Because packing problems have a wide variety of applications in business and industry, it is important to construct a unifying classification scheme for these problems. A sub-typology of six fields incorporating some of the basic and common characteristics encountered in packing problems was introduced in Chapter 1. The most important characteristic is dimensionality, because it characterises the geometry of the items and it is represented in the first field. The second characteristic captures the shape of the items (whether they are regular or irregular). The third characteristic determines the class of packing problems under consideration, whilst the fourth field deals with the type of information about the items to be packed. Objectives differ in packing problems and these are addressed in the fifth field. Finally, packing problems are subject to many different constraints, depending on the application, and these are included in the sixth field and represented in binary fashion. Objective 1 of §1.4 was achieved by the introduction of this classification scheme.

In this dissertation, a survey and review of twenty six known exact and heuristic methods for solving two-dimensional packing problems was carried out in Chapters 2–4. Three new improvements to the special case algorithms dealing with rectangles obeying the *tetris* constraint were presented in §3.2.2. The algorithms seek to take more patterns into consideration than their counterpart from the literature by using a linear array. The modified algorithms were observed to yield solutions of high qualities. A total of twenty possible modifications to some of the heuristics were suggested in Chapters 2 and 3—thus realising Objective 2 of §1.4.

It was observed in some of the classical heuristics from the literature that when there is a significant difference in height between rectangles packed on the same level, this leads to a poor solution quality. This led the author to the development of a new level algorithm in §2.3, named the Size Alternating Stack (SAS) algorithm, which seeks to utilise a level fully by stacking rectangles on top of one another within a level, provided there is sufficient space. Algorithms such as the new SAS algorithm and the Floor-Ceiling

algorithm from the literature, which fully utilise space within a level, were observed to exhibit high quality solutions. Other shelf algorithms from the literature reviewed in this dissertation employ a parameter whose value is selected beforehand by the user, and a poor selection of such a parameter value may possibly lead to poor solution quality. This was the motivation behind seeking a new approach when dealing with shelf algorithms. Such a new approach to shelf algorithms was introduced in §3.3 whereby the history of rectangles packed is employed when creating additional space within each shelf. Four new shelf algorithms applying this new technique were presented in §3.3.1. This method has proven to yield good solutions. A new plane algorithm, named the Online Fit (OF) algorithm, was also developed in §3.3.2. These six new heuristics contributed to the realisation of Objective 3 in §1.4.

All the algorithms (known and new) were implemented in Visual Basic 6.0 with the objective of testing and comparing them. The algorithms were tested in Chapters 5 and 6 on a total of 542 benchmark data sets that have been obtained from the literature, as described in §1.6. The benchmark instances were generated differently and comprise different characteristics. It is imperative to select such a variety of data to investigate whether some algorithms are more inclined to yield good solutions for particular data set types—thus eliminating bias. Once implemented, the algorithms were compared in terms of solution quality (*i.e.* the mean packing heights obtained by the algorithms), the frequency with which the smallest packing height was achieved and efficiency in terms of execution time. The coefficient of variation is an attribute of the data sets used to determine the best algorithm to use for a given data set—thereby fulfilling Objective 4 of §1.4.

A computerised decision support system is desirable to aid managers in industry when making quick and good packing decisions. Such a system, known as the *strip packing decision support system* (SPDSS), was developed in Chapter 7. The SPDSS is able to incorporate large data sets that may be retrieved from storage and modified at any time, since it uses a database to store the data. Such an automated process is required as opposed to manual assembly of layouts, since the latter is costly, time consuming and does not necessarily yield solutions of good quality. The final objective of §1.4 was therefore accomplished by the development of the SPDSS.

8.2 Future Work

Although packing problems have been researched extensively for many years, there is still more work to be done in finding efficient algorithms that are not computationally expensive. Also, new techniques, approaches and methods may be introduced. The author realised that some aspects of the algorithms in this dissertation may be explored further in future, and some of these aspects are outlined in this section.

8.2.1 Future work on offline algorithms

In the FFDH algorithm, described in §2.1.1.2, existing levels are revisited. A rectangle is packed on the lowest level where there is sufficient horizontal space. In future, packing in sub-levels within a level may be explored. Since the rectangles are ordered according to

non-increasing height, sub-levels may be defined as the space between the upper edge of a level and the top edges of rectangles (see Figure 8.1 where three sublevels are denoted by sub 1, sub 2 and sub 3). When searching for the lowest level to accommodate a rectangle, the search may begin with the sub-levels and finally consider the residual horizontal space. If there is insufficient space, then the search may continue to the next level or a new level may be created above the top-most level if none of the existing levels have enough room. The objective should be to attempt to utilise the space within a level fully, because it has been observed that such algorithms yield good solution qualities.

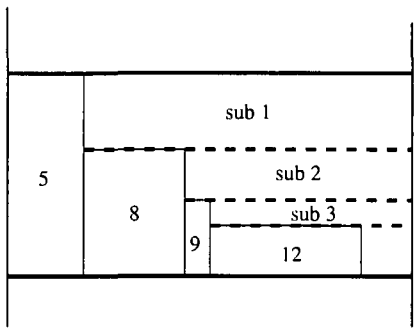

Figure 8.1: *Illustration of three sub-levels (sub 1, sub 2 and sub 3) within a level in the FFDH algorithm.*

In the Sleator$_{mod}$ algorithm, described in §2.1.2.1, the search is terminated if the first narrow rectangle or the first subsequent rectangle cannot be inserted near the wide rectangles. There is a possibility of improving this procedure by not terminating the search, but rather traversing the entire list of narrow rectangles, because there might be some that fit. Nonetheless, if the list of rectangles to be packed is long, then this may have a significantly negative impact on the execution time.

The new size alternating stack (SAS) algorithm, described in §2.3, has been observed to yield good solutions. In future, this algorithm may be incorporated into an exact algorithm at the root node to form an initial solution or at the descendant nodes to improve the incumbent solution.

In the literature, bin packing problems have been solved in two-phases, whereby the first phase comprises strip packing and the second phase packs the resulting blocks into bins [14, 79]. A block is defined as the collection of rectangles packed on the same level into the strip. In a similar fashion, the SAS algorithm may be used as a first phase in the 2D bin packing problem.

It would also be of interest to carry out a theoretical analysis of the asymptotic performance bounds of the newly proposed algorithms and thus compare them to the known algorithms. This would give an indication of how good or poor, in theory, the algorithms perform in relation to an optimal solution. The performance of the SAS algorithm should be compared to that of the FFDH algorithm and for online algorithms, the new shelf algorithms should be compared with those from the literature that employ the parameter r.

In recent years, evolutionary methods, such as genetic algorithms (GAs) and neural networks, have been used to solve packing problems [56, 64, 74, 94]. These are methods that

search the solution space of feasible solutions in an attempt to find the best solution. Genetic algorithms employ biological evolution as an optimisation tool. In genetic algorithms a population consisting of chromosomes (which, in turn, consist of genes) is used to move from one population of candidate solutions to the next. The size of the population is important, because it determines how accurate and how long the GA may take to reach a good solution—moderately sized populations are preferred. The two important features in a GA are *crossover* and *mutation*—the former refers to the exchange of genes from parent chromosomes to form new offspring, while in the latter the newly formed offspring are randomly perturbed. The idea is to take good genes from the parents in the hope of producing good offspring solutions. The most important step when using GAs to solve a particular problem is to decide on an appropriate encoding scheme for the chromosomes—some of the commonly used coding schemes are binary encoding and permutation encoding. In binary encoding, the chromosomes are represented as a string of zeros and ones and this type of encoding is suitable for use in solving problems such as the knapsack problem. Permutation encoding, on the other hand, is suitable for use in bin packing problems, since the chromosomes represent the order in which items are packed.

The potential of integrating the SAS algorithm into a GA may be explored. A permutation encoding scheme may be used with each chromosome assigned either a "+" or "–" sign denoting a narrow and a wide rectangle respectively. The fitness of a candidate solution may be taken as the packing height obtained, and if two offspring produce the same fitness, then the offspring with the largest wasted area may be selected, because potentially such space may be utilised further to pack rectangles.

8.2.2 Future work on online algorithms

A plane algorithm, referred to as the Online Fit (OF) algorithm, was described in §3.3.2. This online algorithm achieved an average performance, but the most interesting aspect of the algorithm was that it was the only heuristic in this dissertation that was able to achieve an optimal packing height for certain benchmark instances. This was a clear indication that with some adjustments to some of the steps, the OF algorithm may achieve much better performance. The OF algorithm uses a linear array to store the packing heights at various x–coordinates of the strip and involves four main steps:

1. empty areas are searched for sufficient space,
2. the linear array with identical entries is searched for sufficient space,
3. the linear array with decreasing order entries is searched for sufficient space and
4. items are packed at the top.

A possible improvement to the OF algorithm would be to eliminate step 3 and instead search for sufficient space by considering the widths of the nearest neighbouring rectangle. A rectangle may have one neighbour (left or right) or two neighbours (left and right). A left (right, resp.) neighbour of rectangle L_i is defined as a upper edge of a rectangle that coincides with the left-hand (right-hand, resp.) edge of rectangle L_i. The bottom edge of the strip may also be a considered a neighbour if no rectangle has been packed on the

right-hand side of rectangle L_i. In Figure 8.2(a), the left neighbour of rectangle 5 is the top edge of rectangle 4 and the right neighbour is part of the top edge of rectangle 2 from $x = 3$. On the other hand, in Figure 8.2(b) the left neighbour of rectangle 3 is part of the top edge of rectangle 2 from $x = 3$ and the right neighbour is the bottom edge of the strip from $x = 8$. The total width considered for packing is the sum of the widths of the rectangle and its neighbours. A rectangle is packed on the width of sufficient space and at the lowest height (this is the height coinciding with the top edge of the rectangle whose neighbours are being considered). In Figure 8.2(c), there are two total widths to consider before packing rectangle 6 and these are a width of 7 at height 16 or a width of 6 at height 8. Clearly, the second option should be selected, because it has sufficient width and it occurs at the lowest height.

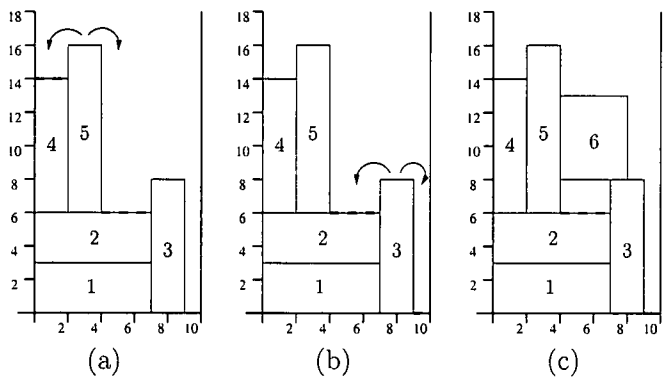

Figure 8.2: *Illustration of the notion of nearest neighbours which are indicated by the dashed lines. In (a) a total width of 2 + 2 + 3 = 7 units occurs at a height of 16, while in (b) a total width of 3 + 2 + 1 = 6 units occurs at a height of 8.*

Online packing problems still present a challenging area in the packing problem literature and more work has to be done by either introducing new packing techniques (as was done in this dissertation) or by introducing new solution methods.

8.2.3 Future work on the decision support system

The newly developed SPDSS has been developed so that it may be incorporated into a larger decision support system. It may, for example, be combined with a bin packing decision support system so that the user has a choice between strip packing or bin packing algorithms at his/her disposal during the decision support process.

References

[1] Arjomand L, 2002, *Descriptive Statistics*, [Online], [cited 2007, June 28], Available from: `http://business.clayton.edu/arjomand/business/l3.html`

[2] Azar Y & Epstein L, 1997, *On Two-Dimensional Packing*, Journal of Algorithms, **25(2)**, pp. 290–310.

[3] Azar Y, 2007 (March 10), *One of the authors of [2]*, [Personal Communication via e-mail], Contactable at `azar@post.tau.ac.il`

[4] Baker BS, Coffman EG & Rivest RL, 1980, *Orthogonal Packings in Two Dimensions*, SIAM Journal on Computing, **9(4)**, pp. 846–855.

[5] Baker BS, Brown DJ & Katseff HP, 1981, *A 5/4 Algorithm for Two-Dimensional Packing*, Journal of Algorithms, **2**, pp. 348–368.

[6] Baker BS & Schwarz JS, 1983, *Shelf Algorithms for Two-Dimensional Packing Problems*, SIAM Journal on Computing, **12(3)**, pp. 508–525.

[7] Barnes FW, 1979, *Packing the Maximum Number of $m \times n$ Tiles in a Large $p \times q$ Rectangle*, Discrete Mathematics, **26**, pp. 93–100.

[8] Bartholdi III JJ, Vate JHV & Zhang J, 1989, *Expected Performance of the Shelf Heuristic for 2-Dimensional Packing*, Operations Research Letters, **8**, pp. 11–16.

[9] Beasley JE, 1985, *Algorithms for Unconstrained Two-Dimensional Guillotine Cutting*, Journal of the Operational Research Society, **36(4)**, pp. 297–306.

[10] Beasley JE, 1985, *An Exact Two-Dimensional Non-Guillotine Cutting Tree Search Procedure*, Operations Research, **33(1)**, pp. 49–64.

[11] Beasley JE, 1990, *OR-Library: Distributing Test Problems by Electronic Mail*, Journal of the Operational Research Society, **41(11)**, pp. 1069–1072.

[12] Benati S, 1997, *An Algorithm for a Cutting Stock Problem on a Strip*, Journal of the Operational Research Society, **48(3)**, pp. 288–294.

[13] Bengtsson B-E, 1982, *Packing Rectangular Pieces—A Heuristic Approach*, The Computer Journal, **25(3)**, pp. 353–357.

[14] Berkey JO & Wang PY, 1987, *Two-Dimensional Finite Bin Packing Algorithms*, Journal of the Operational Research Society, **38(5)**, pp. 423–429.

[15] Biro M & Boros E, 1984, *Network Flows and Non-Guillotine Cutting Patterns*, European Journal of Operational Research, **16**, pp. 215–221.

[16] Bischoff EE & Wascher G, 1995, *Cutting and Packing*, European Journal of Operational Research, **84**, pp. 503–505.

[17] Brown DJ, Baker BS & Katseff HP, 1982, *Lower Bounds for On-line Two-Dimensional Packing Algorithms*, Acta Informatica, **18**, pp. 207–225.

[18] Burke EK, Kendall G & Whitwell G, 2004, *A New Placement Heuristic for the Orthogonal Stock-Cutting Problem*, Operations Research, **52(4)**, pp. 655–671.

[19] Caprara A & Monaci M, 2004, *On the Two-Dimensional Knapsack Problem*, Operations Research Letters, **32**, pp. 5–14.

[20] Chartrand G & Oellermann OR, 1993, *Applied and Algorithmic Graph Theory*, McGraw-Hill Inc., New York (NY).

[21] Chazelle B, 1983, *The Bottom-Left Bin Packing Heuristic: An Efficient Implementation*, IEEE Transactions on Computers, **32(8)**, pp. 697–707.

[22] Christofides N & Hadjiconstantinou E, 1995, *An Exact Algorithm for Orthogonal 2-D Cutting Problems using Quillotine Cuts*, European Journal of Operational Research, **83**, pp. 21–38.

[23] Christofides N & Whitlock C, 1977, *An Algorithm for Two–Dimensional Cutting Problems*, Operations Research, **25(1)**, pp. 31–44.

[24] Chung FRK, Garey MR & Johnson DS, 1982, *On Packing Two-Dimensional Bins*, SIAM Journal on Algebraic and Discrete Methods, **3(1)**, pp. 66–76.

[25] Coffman Jr. EG, 2007 (March 09), *One of the authors of [26], [27], [29], [30]*, [Personal Communication via e-mail], Contactable at `coffman@cs.columbia.edu`

[26] Coffman Jr. EG & Shor PW, 1990, *Average-case Analysis of Cutting and Packing in Two Dimensions*, European Journal of Operational Research, **44**, pp. 134–144.

[27] Coffman Jr. EG & Shor PW, 1993, *Packings in Two Dimensions: Asymptotic Average-Case Analyisis of Algorithms*, Algorithmica, **9**, pp. 253–277.

[28] Coffman Jr. EG, Csirik J & Woeginger GJ, 2002, *Approximate Solutions to Bin Packing Problems*, pp. 607–615 in Pardalo PM & Resende MGC (EDS.), *Handbook of Applied Optimization*, Oxford University Press, New York (NY).

[29] Coffman Jr. EG, Downey PJ & Winkler P, 2002, *Packing Rectangles in a Strip*, Acta Informatica, **38**, pp. 673–693.

[30] Coffman Jr. EG, Garey DS & Tarjan RE, 1980, *Performance Bounds for Level Oriented Two-Dimensional Packing Algorithms*, SIAM Journal on Computing, **9(4)**, pp. 808–826.

[31] Coffman Jr. EG, Garey MR & Johnson DS, 1996, *Approximation Algorithms for Bin Packing: A Survey*, pp. 46–93 in Hochbaum D, (ED.), *Approximation Algorithms for NP Hard Problems*, PWS Publishing, Boston (MA).

[32] Coppersmith D and Raghavan P, 1989, *Multidimensional On-Line Bin Packing: Algorithms and Worst-Case Analysis*, Operations Research Letters, **8**, pp. 17–20.

[33] Csirik J & Woeginger GJ, 1997, *Shelf Algorithms for On-line Strip Packing*, Information Processing Letters, **63**, pp. 171–175.

[34] Czumaj A, 2006, *Complexity of Algorithms* [Online], [cited 2007, January 24], Available from: `http://www.dcs.warwick.ac.uk/~czumaj/cs301/Lecture-28-CIS-301.ppt`

[35] Davies OL & Goldsmith PL (EDS.), 1980, *Statistical Methods in Research and Production*, Longman Group Inc., New York (NY).

[36] Deacon J, 2006, *The Really Easy Statistics Site*, [Online], [cited 2005, November 28], Available from: `http://www.biology.ed.ac.uk/research/groups/jdeacon/statistics/tress1.html`

[37] DEIS (Dipartmento di Elettronica, Informatica e Sistemistica Operations Research Group), *Operations Research Group: Library of Instances*, [Online], [cited 2005, April 22], Available from: `http://www.or.deis.unibo.it/research.html`

[38] Dogrusoz U, 2002, *Two-Dimensional Packing Algorithms for Layout of Disconnected Graphs*, Information Sciences, **143**, pp. 147–158.

[39] Downey PJ, 2006 (March 23), *One of the authors of [29]*, [Personal Communication via e-mail], Contactable at `pete@cs.arizona.edu`

[40] Dowsland KA & Dowsland WB, 1992, *Packing Problems*, European Journal of Operational Research, **56**, pp. 2–14.

[41] Dyckhoff H, 1990, *A Typology of Cutting and Packing Problems*, European Journal of Operational Research, **44**, pp. 145–159.

[42] Dyckhoff H, Kruse H-J, Abel D & Gal T, 1985, *Trim Loss and Related Problems*, Omega, **13**, pp. 59–72.

[43] Erickson J, 2003, *NP-hard, NP-easy, and NP-complete*, [Online], [cited 2007, June 28], Available from: `http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/16-nphard.pdf`

[44] Fernandez de la Vega W & Leuker GS, 1981, *Bin Packing can be Solved within* $1+\epsilon$ *in Linear Time*, Combinatorica, **1(4)**, pp. 349–355.

[45] Fernandez de la Vega W & Zissimopoulos V, 1998, *An Approximation Scheme for Strip Packing of Rectangles with Bounded Dimensions*, Discrete Applied Mathematics, **82**, pp. 93–101.

[46] Garey MR & Johnson DS, 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York (NY).

[47] Gradisar M, Resinovic G & Kljajic M, 2002, *Evaluation of Algorithms for One-Dimensional Cutting*, Computers and Operations Research, **29**, pp. 1207–1220.

[48] Goldberg DE, 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Inc., New York (NY).

[49] Harlvorson M, 2003, *Miscrosoft Visual Basic 6.0 Professional Step by Step*, 2nd Edition, Microsoft Press, Washington DC.

[50] Henning MA & Van Vuuren JH, 2007, *A First Course in Graph Theory*, In Press.

[51] Hifi M, 1997, *The DH/KD Algorithm: A Hybrid Approach for Unconstrained Two-Dimensional Cutting Problems*, European Journal of Operational Research, **97**, pp. 41–52.

[52] Hifi M, 1998, *Exact Algorithms for the Guillotine Strip Cutting/Packing Problem*, Computers and Operations Research, **25(11)**, pp. 925–940.

[53] Hifi M, 2002, *Approximate Algorithms for the Container Loading Problem*, International Transactions in Operational Research, **9**, pp. 747–774.

[54] Hinxman AI, 1980, *The Trim-Loss and Assortment Problems: A Survey*, European Journal of Operational Research, **5**, pp. 8–18.

[55] Hofri M, 1980, *Two-Dimensional Packing: Expected Performance of Simple Level Algorithms*, Information and Control, **45**, pp. 1–17.

[56] Hopper E & Turton BCH, 2001, *A Review of the Application of Meta-Heuristic Algorithms to 2D Strip Packing Problems*, Artificial Intelligence Review, **16**, pp. 257–300.

[57] Hopper E & Turton BCH, 2001, *An Empirical Investigation of Meta-heuristic and Heuristic Algorithms for a 2D Packing Problem*, European Journal of Operational Research, **128(1)**, pp. 34–57.

[58] Hopper E & Turton BCH, 2002, *Problem Generators for Rectangular Packing Problems*, Studia Informatica Universalis, **2(1)**, pp. 123–136.

[59] Hromkovic J, 2004, *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*, 2nd Edition, Springer-Verlag, Berlin.

[60] Imahori S, 2007 (June 18), *Cutting and Packing*, [Online], [cited 2005, March 11], Available from: `http://www.simplex.t.u-tokyo.ac.jp/~imahori/packing/instance.html`

[61] Imahori S, Yagiura M & Nagamochi H, 2006, *Practical Algorithms for Two-dimensional Packing*, Mathematical Engineering Technical Report METR 2006-19, Department of Mathematical Informatics, University of Tokyo, Tokyo.

[62] Imreh C, 2001, *Online Strip Packing with Modifiable Boxes*, Operations Research Letters, **29**, pp. 79–85.

[63] Imreh C, 2007 (March 10), *Author of [62]*, [Personal Communication via e-mail], Contactable at `cimreh@inf.u-szeged.hu`.

[64] Jakobs S, 1996, *On Genetic Algorithms for the Packing of Polygons*, European Journal of Operational Research, **88**, pp. 165–181.

[65] Jansen K, 2006 (November 29), *Researcher on packing problems*, [Personal Communication via e-mail], Contactable at `kj@informatik.uni-kiel.de`

[66] Johnsonbaugh R & Schaefer M, 2004, *Algorithms*, Pearson Prentice–Hall, Upper Saddle River (NJ).

[67] Kantorovich LV, 1960, *Mathematical Methods of Organising and Planning Production* (translated from a report in Russian, dated 1939), Management Science, **6**, pp. 366–422.

[68] Karmarkar N & Karp RM, 1982, *An Efficient Approximation Scheme for the One-Dimensional Bin Packing Problem*, Proceedings of the 23rd Symposium on Foundations of Computer Science (FOCS), pp. 312–320.

[69] Kendall G, 2007 (February 19), *One of the authors of [18]*, [Personal Communication via e-mail], Contactable at `gxk@cs.nott.ac.uk`.

[70] Kenyon C & Remila E, 2000, *A Near Optimal Solution to a Two-Dimensional Cutting Stock Problem*, Mathematics of Operations Research, **25**, pp. 645–656.

[71] Kenyon C, *Claire Kenyon's Research Statement*, [Online], [cited 2007, February 6], Available from: `http://www.lix.polytechnique.fr/~kenyon/`

[72] Lee CC & Lee DT, 1985, *A Simple On-line Bin Packing Algorithm*, Journal of the Association of Computing Machinery, **32(3)**, pp. 562–572.

[73] Lesh N, Marks J, McMahon A & Mitzenmacher M, 2005, *New Heuristic and Interactive Approaches to 2D Rectangular Strip Packing*, ACM Journal of Experimental Algorithmics, **10**, Article No. 1.2, pp. 1–18.

[74] Liu D & Teng H, 1999, *An Improved BL-Algorithm for Genetic Algorithm of the Orthogonal Packing of Rectangles*, European Journal of Operational Research, **112**, pp. 413–420.

[75] Lodi A, 2006 (November 30), *One of the authors of [77, 78, 79, 80]*, [Personal Communication via e-mail], Contactable at `alodi@deis.unibo.it`

[76] Lodi A, 2000, *Algorithms for Two-Dimensional Bin Packing and Assignment Problems*, PhD Thesis, University of Bologna, Bologna.

[77] Lodi A, Martello S & Monaci M, 2002, *Two-Dimensional Packing Problems: A Survey*, European Journal of Operational Research, **141**, pp. 241–252.

[78] Lodi A, Martello S & Vigo D, 1998, *Neighbourhood Search Algorithm for the Guillotine Non-Oriented Two-Dimensional Bin Packing Problem*, pp. 125–139 in Voss S, Martello S, Osman IH & Roucairol C (EDS.), *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston (MA).

[79] Lodi A, Martello S & Vigo D, 1999, *Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems*, INFORMS Journal on Computing, **11**, pp. 345–357.

[80] Lodi A, Martello S & Vigo D, 2004, *Models and Bounds for Two-Dimensional Level Packing Problems*, Journal of Combinatorial Optimization, **8(3)**, pp. 363–379.

[81] Lowry R, 2007, *Concepts and Applications of Inferential Statistics*, [Online], [cited 2007, June 28], Available from: `http://faculty.vassar.edu/lowry/webtext.html`.

[82] Martello S, 2006 (November 30), *One of the authors of [83, 84, 85, 86, 87]*, [Personal Communication via e-mail], Contactable at `smartello@deis.unibo.it`

[83] Martello S & Toth P, 1980, *Optimal and Canonical Solutions of the Change Making Problem*, European Journal of Operational Research, **4**, pp. 322–329.

[84] Martello S & Toth P, 1990, *Lower Bounds and Reduction Procedures for the Bin Packing Problem*, Discrete Applied Mathematics, **28**, pp. 59–70.

[85] Martello S & Vigo D, 1998, *Exact Solution to The Two-Dimensional Finite Bin Packing Problem*, Management Science, **44**, pp. 388–399.

[86] Martello S, Monaci M & Vigo D, 2003, *An Exact Approach to the Strip-Packing Problem*, INFORMS Journal on Computing, **15(3)**, pp. 310–319.

[87] Martello S, Pisinger D & Vigo D, 2000, *The Three Dimensional Bin Packing Problem*, Operations Research, **48**, pp. 256–267.

[88] Matlab: The Language of Technical Computing, 2007, *The MathWorks*, [Online], [cited 2007, March 3], Available from: `http://www.mathsworks.com`.

[89] Microsoft Visual Basic Developer Center, 2007, *MSDN*, [Online], [cited 2004, January 5], Available from: `http://msdn.microsoft.com/vbasic`.

[90] Microsoft Office Online, 2007, *Microsoft Product Information*, [Online], [cited 2007, June 1], Available from: `http://office.microsoft.com`.

[91] Miyazawa FK & Wakabayashi Y, 1998, *Parametric On-Line Packing*, pp. 109–121 in Anais do XXX Simpsio Brasileiro de Pesquisa Operacional / Workshop da III Oficina Nacional de Problemas de Corte and Empacotamento, Curitiba-Pr.

[92] Monaci M, 2004 (July 28), *One of the authors of [86]*, [Personal Communication via e-mail], Contactable at `mmonaci@deis.unibo.it`

[93] Moore DS & McCabe GP, 1993, *Introduction to the Practice of Statistics*, 2^{nd} Edition, W.H. Freeman and Company, New York (NY).

[94] Mumford-Valenzuela C, Wang PY & Vick J, 2001, *Heuristics for Large Strip Packing Problems with Guillotine Patterns: An Empirical Study*, Proceedings of the 4th Metaheuristics International Conference, pp. 417–421.

[95] Murty KG, 1995, *Operations Research: Deterministic Optimization Models*, Prentice–Hall, Upper Saddle River (NJ).

[96] National Institute of Standards and Technology, 1998, *Dictionary of Algorithms and Data Structures* [Online], [cited 2007, June 16], Available from: http://www.nist.gov/dads/.

[97] Ntene N & Van Vuuren JH, *A Survey and Comparison of Level Heuristics for the 2D Oriented Strip Packing Problem, Discrete Optimization*, Submitted.

[98] Obitko M, 1998, *Genetic Algorithms*, [Online], [cited 2007, June 17], Available from: http://cs.felk.cvut.cz/~xobitko/ga/.

[99] O'Leary N, 2007 (May 17), *Consultant at Optinum*, [Personal Communication via e-mail], Contactable at support@optinum.co.za

[100] OR-Library, [Online], [cited 2005, February 25], Available from: http://mscmga.ms.ic.ac.uk/info.html.

[101] Part J, 2005 (February 8), *Information Technology Specialist*, [Personal Communication via e-mail], Contactable at jp@iol.ie

[102] Philips JL, 1999, *How to Think About Statistics*, 6^{th} Edition, Henry Halt and Company, New York (NY).

[103] Preiss BR, 1998, *Algorithm Analysis*, [Online], [cited 2007, March 5], Available from: http://www.brpreiss.com/books/ opus5/html/page36.html

[104] Remila E, 2004 May 17, *One of the authors of [70]*, [Personal Communication via e-mail], Contactable at eric.remila@ens-lyon.fr

[105] Rohlf FJ & Sokal RS, 1981, *Statistical Tables*, W.H. Freeman and Company, New York (NY).

[106] Schuurman P & Woeginger GJ, 2001, *Approximation Schemes: A Tutorial*, To be published, in Moehring RH, Potts CN, Schulz AS, Woeginger GJ & Wolsey LA (EDS.), *Lectures on Scheduling*, to appear, http://www.win.tue.nl/~egwoegi/papers/ptas.ps.

[107] SICUP (Special Interest Group in Cutting and Packing), [Online], [cited 2005, March 18], Available from: http://www.apdio.pt/sicup.

[108] Sleator D, 1980, *A 2.5 Times Optimal Algorithm for Packing in Two Dimensions*, Information Processing Letters, **10(1)**, pp. 37–40.

[109] Smith H, 2001, *Bin Packing*, [Online], [cited 2003, November 16], Available from: http://www.cs.cf.ac.uk/user/C.L.Mumford/heidi/Approaches.html

[110] Steinberg A, 1997, *A Strip-Packing Algorithm with Absolute Performance Bound 2*, SIAM Journal on Computing, **26(2)**, pp. 401–409.

[111] Talbot J & Welsh D, 2006, *Complexity and Cryptography: An Introduction*, Cambridge University Press, Cambridge.

[112] Turban E & Aronson JE, 2001, *Decision Support Systems and Intelligent Systems*, 6^{th} Edition, Prentice–hall, Inc., Upper Saddle River (NJ).

[113] Van Vuuren JH, 2001, *Repositories (Applied Mathematics Division of the Department of Mathematical Sciences)*, [Online], [cited 2005, November 22], Available from: `http://dip.sun.ac.za/~vuuren/ repositories/strippacking.htm`

[114] Wäscher G, Haußner H & Schumann H, 2006, *An Improved Typology of Cutting and Packing Problems*, European Journal of Operational Research, Article in Press, [Online], [cited 2006, October 12], Available from: `http://www.sciencedirect.com`

[115] Wäscher G, 2006 (October 12), *Corresponding author of [114]*, [Personal Communication via e-mail], Contactable at `gerhard.waescher@ww.uni-magdeburg.de`

[116] Wang PY & Valenzuela CL, 2001, *Data Set Generation for Rectangular Placement Problems*, European Journal of Operational Research, **134**, pp. 378–391.

[117] Whelan P, *Vision Systems Laboratory*, Book Chapter Draft #1, (12 March 2001).

[118] Whitwell G, 2005 (May 9), *One of the authors of [18]*, [Personal Communication via e-mail], Contactable at `gxw@cs.nott.ac.uk`

[119] Wikipedia, 2007, *Pseudo-polynomial Time*, [Online], [cited 2007, February 1], Available from: `http://en.wikipedia.org/wiki/Pseudo-polynomial_time`

[120] Wikipedia, 2007, *Polylogarithmic*, [Online], [cited 2007, February 16], Available from: `http://en.wikipedia.org/wiki/Polylogarithmic`

[121] Wigderson A, 2000, *P, NP and Mathematics-A Computational Complexity Perspective*, [Online], [cited 2007, June 16], Available from: `http://www.math.ias.edu/~avi/PUBLICATIONS/MYPAPERS/W06/W06.pdf`

[122] Wu Y-L, Huang W, Lau S-C, Wong CK & Young GH, 2002, *An Effective Quasi-Human Based Heuristic for Solving the Rectangle Packing Problem*, European Journal of Operational Research, **141**, pp. 341–358.

[123] Xiaodong G, Guoliang C & Yinlong X, 2005, *Average-Case Performance Analysis of a 2D Strip Packing Algorithm–NFDH*, Journal of Combinatorial Optimization, **9**, pp. 19–34.

[124] Zhang G, 2005, *A 3-Approximation Algorithm for Two-Dimensional Bin Packing*, Operations Research Letters, **33**, pp. 121–126.

[125] Zhang G, Cai X & Wong CK, 2000, *Linear Time-Approximation Algorithms for Bin Packing*, Operations Research Letters, **26**, pp. 217–222.

Appendix A

Basic concepts in algorithmic complexity and performance

Algorithmic complexity, as defined by Chartrand and Oellermann [20], measures the amount of computational effort expended when a computer solves a problem using a specific algorithm. The objective is to understand the intrinsic difficulty in computational problems [111]. It is usually measured by two variables, the *time complexity* and the *space complexity*. The former measures the time required to execute the algorithm, while the latter measures the computer memory space required to execute the algorithm. In this dissertation only time complexity is considered, because the space complexities of all algorithms in this dissertation are negligible. Most algorithms have running time dependent on the input—as such, the performance of an algorithms is typically measured in terms of running time over all inputs. Since computing the exact time required by a computer to implement an algorithm is affected by various factors, such as processor speed and operating system, a more universal notation measuring the order of magnitude of this time as a function of the input size of the problem instance is preferred above measuring the time in seconds (say) on a specific computer. Suppose the input size is denoted by n, then the "big O" notation, $O(expression)$ where *expression* is a function of n, is a measure of growth as the input size increases. This method removes all constants from *expression* and gives an estimate of the execution time in relation to some function of n as $n \to \infty$. Some of the common classifications of complexities are:

- Constant, $O(1)$: complexity is independent of n.
- Linear, $O(n)$: complexity is directly proportional to n.
- Logarithmic, $O(\log n)$: complexity is proportional to the number of times n can be divided by 2.
- Quadratic, $O(n^2)$: complexity is proportional to the square of n.
- Exponential, $O(a^n)$: complexity grows exponentially in n.

Let f and g be non-negative functions on the positive integers. If

$$f(n) \leq c_1\, g(n) \quad \text{for all } n \geq N_1 \tag{A.1}$$

for some constants $c_1 > 0$ and $N_1 \in \mathbb{N}$, then one writes $f(n) = O(g(n))$ and it is said that "$f(n)$ is of order at most $g(n)$". In this case g is referred to as an *asymptotic upper bound* for f. On the other hand, if

$$f(n) \geq c_2\, g(n) \quad \text{for all } n \geq N_2 \tag{A.2}$$

for some constants $c_2 > 0$ and $N_2 \in \mathbb{N}$, then one writes $f(n) = \Omega(g(n))$ and it is said that "$f(n)$ is of order at least $g(n)$". In this case g is referred to as an *asymptotic lower bound* for f. $\Omega(expression)$ denotes all functions that grow faster than or at the same rate as *expression* as $n \to \infty$. Lastly, if

$$c_3\, g(n) \leq f(n) = c_4\, g(n) \quad \text{for all } n \geq N_3 \tag{A.3}$$

for some constants $c_3, c_4 > 0$ and $N_3 \in \mathbb{N}$, then one writes $f(n) = \Theta(g(n))$ and it is said that "$f(n)$ is of order $g(n)$". In this case g is referred to as an *asymptotically tight bound* for f [66]. $\Theta(expression)$ denotes all functions which grow at the same rate as *expression* as $n \to \infty$. The advantage of using these three measurements ($O(expression)$, $\Omega(expression)$ and $\Theta(expression)$), is that they are system-independent, meaning that they are not affected by the processing power of a specific computer. There are different ways of comparing execution times of algorithms and these include *best-case, average-case* and *worst-case* time complexities. The best-case (average-case, resp.) time complexity is the minimum (average, resp.) running time over all possible inputs of size n. In this dissertation, the *worst-case time* complexity is considered throughout and it refers to the maximum time required to execute an algorithm for inputs of size n based on the number of "basic operations" performed.

Algorithms for which the complexity is $O(n^c)$, for some fixed $c \in \mathbb{R}^+$, are said to be *polynomial* in the input size n. When considering certain special cases of packing problems, a number of polynomial time algorithms and even optimal packing strategies are known ([4, 70]). The computational complexity and efficiency of an algorithm may be evaluated by applying it to established and well documented benchmark packing problems ([73, 122]), and comparing its performance with those of other algorithms. Performance bounds may also be derived (either absolute and asymptotic performance bounds). Given a list $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ of items to be packed, the objective function value of an optimal packing of $\mathcal{L}$ is denoted by $OPT(\mathcal{L})$—this value may be the minimum number of bins packed in the case of a multiple bin packing problem, or the height of the packing obtained in a strip packing problem, or even the maximum number of items that may fit into a single closed bin without overlapping. Suppose A is an algorithm, and that $A(\mathcal{L})$ is the objective function value of the packing produced by A, when packing items in the list $\mathcal{L}$. If α is a constant such that

$$A(\mathcal{L}) \;\leq\; \alpha\, OPT(\mathcal{L}), \tag{A.4}$$

then α is known as an absolute worst case performance bound of A. If α and β are constants such that, for every possible list $\mathcal{L}$ of rectangles,

$$A(\mathcal{L}) \;\leq\; \alpha\, OPT(\mathcal{L}) + \beta \tag{A.5}$$

then, α is known as an asymptotic worst case performance bound of A. Worst case bounds are most commonly used as standard measures of the performance in the evaluation of heuristics. However, the so-called asymptotic packing efficiency may also be used as a

measure of performance, and is defined as the expected rate of increase of waste in the packing, expressed as a function of the number of objects packed. To determine the best algorithm for a specific application, Dowsland and Dowsland [40] suggest testing it empirically on typical or benchmark data sets.

In complexity theory, the aim is to classify problems according to their computational hardness. A problem is considered to be "hard" if there is no deterministic algorithm that solves it efficiently [59]. In this context, efficiently means a low degree polynomial algorithm exists that may solve the problem. A problem with running time in $O(n^{1000})$ is obviously impractical since $n^{1000} > 2^n$ for all reasonable input sizes n. Fortunately, polynomial time algorithms for "natural" problems almost always have low degree polynomial running time in practice [59].

A large part of decision theory deals with a special type of problem known as a *decision problem*—such a problem takes a string as input and returns "yes/no" answer as output [111]. If a "yes" is returned in polynomial time, then the problem is classified as a problem in the class **P** (an acronym for *polynomial*). Polynomial time algorithms have been selected to represent an efficient computation, because they typify "slow growing" functions [121]. Yet even a low degree polynomial time complexity is not a guarantee of speediness, as there may be very large constants in this time complexity masked by O-notation.

If, a "yes" is returned, given additional information to the problem instance, then the problem is classified as a member of the class **NP** (an acronym for *non-deterministic polynomial*). The additional information given with respect to a particular instance of the decision problem is known as a *certificate* (a certificate specific to a decision problem might exist but it may be difficult to determine such a certificate). Currently there is no method to prove that most of the practical problems of interest are not in **P**—this led to the introduction of the concept of NP-completeness. This concept provides at least a good reason to believe that a specific problem is hard when one is unable to prove evidence of this fact [59]. *Polynomial time reducibility* is a concept introduced for "how" to prove the hardness of some problems even though there is no direct mathematical method for this purpose. This involves the transformation of any hypothetical algorithm into an efficient algorithm for any other decision problem in **NP**. More formally stated, let K_1 and K_2 be two decision problems. Then K_1 is said to be *polynomial-time reducible* to K_2, denoted $K_1 \rightsquigarrow K_2$, if

a) there exists an algorithm A_1 capable of solving any instance of K_1 ,

b) an algorithm A_2 exists which is capable of solving all instances of K_2 and

c) A_1 contains as subroutine algorithm A_2 such that A_1 is a polynomial time algorithm if A_2 is a polynomial time algorithm.

The implication of polynomial time reducibility is that $K_1 \in \mathbf{P}$ if $K_1 \rightsquigarrow K_2$ and $K_2 \in \mathbf{P}$ [50]. If $K_1 \rightsquigarrow K$ for all $K_1 \in \mathbf{NP}$, then K is said to be **NP**-hard. A problem K is **NP**-hard if a polynomial-time algorithm for K would imply a polynomial-time algorithm for every problem in **NP** [43]. If $K \in \mathbf{NP}$ and K is **NP**-hard, then K is said to be **NP**-complete. Loosely speaking, an **NP**-complete problem is a problem instance B coupled with the fact that no other problem in **NP** is more than a polynomial factor harder to solve than B [96]. Problems outside **P** are "real" computational problems which in theory, can be

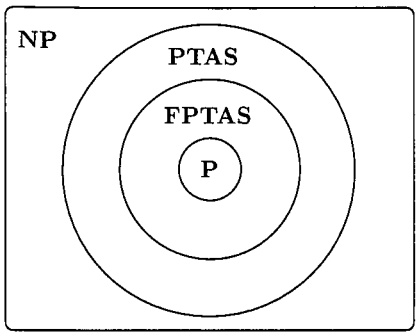

Figure A.1: *Complexity classes and their relationships*

solved by computer, but in many cases the only known algorithms for this purpose are completely impractical. As such, more work has concentrated on the use of *approximation algorithms*. In these algorithms, time complexity may be improved although the solution quality is compromised, since an approximation algorithm searches for an approximation of the optimal solution rather than searching for an optimal solution. The objective is to determine, for a fixed $\epsilon > 0$, an algorithm whose cost differs from the cost of the optimal solution by at most $\epsilon\%$. An *approximation scheme* for a problem is a family of algorithms that use a precision requirement $0 < \epsilon < 1$. A *polynomial time approximation scheme* (PTAS) for a problem is a class of algorithms $\{A_\epsilon : \epsilon > 0\}$ such that for each fixed ϵ, A_ϵ is a polynomial time algorithm for the problem at hand. A *fully polynomial time approximation scheme* (FPTAS) is bounded by a function which is polynomial in both the size of the input and $1/\epsilon$. Problems are often classified in relation to P and this relationship between the different classes of complexities are shown in Figure A.1. A FPTAS for the two-dimensional strip packing problem was developed by Kenyon and Remila [70].

Appendix B

One-dimensional bin packing problems

This appendix serves to describe some of the principles used in the Kenyon-Remila algorithm described in §2.1.3.

In the one dimensional (1D) bin packing problem the objective is, given a list $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ of rectangles and a bin capacity $C > 0$, to pack the rectangles in $\mathcal{L}$ into a minimum number of bins of capacity C. The rectangles in $\mathcal{L}$ have width equal to the width of the bin, but varying heights.

Karmarkar and Karp [68] introduced the notion of a *fractional bin packing* and of a *linear grouping* when solving packing problems of the form

1D	R	MFB	Off	MiB	0,0,0,0

and these notions are described here. Karmarkar and Karp [68] developed an algorithm for the 1D bin packing problem, which requires n rectangles of m different types to be packed into a minimum number of bins, such that the sum of the heights of the rectangles is less than or equal to 1 (here 1 represents the bin capacity). A *configuration* is defined as a multi-set of rectangle types capable of being packed together within a single bin. The *fractional bin packing problem* is the linear program in which the objective is to

$$\begin{array}{lrcl} \text{minimise} & z & = & \underline{1}.\underline{x}, \\ \text{subject } to & A\underline{x} & \geq & \underline{b}, \\ & \underline{x} & \geq & 0, \end{array} \tag{B.1}$$

where $\underline{1}$ is the q-vector of all ones, $\underline{x} = [x_1, \ldots, x_q]^T$, where x_j is the number of occurrences of bins in which configuration C_j is packed and $\mathbf{A}$ is an $m \times q$ matrix whose $(i,j)^{th}$ entry $\mathbf{A}_{ij}$ denotes the number of type i rectangles in configuration C_j. The number of possible configurations is denoted by q and $\underline{b} = [b_1, \ldots, b_m]^T$, where b_i is the number of rectangles of type i to be packed. The notion of a *linear grouping* refers to the process of partitioning the set of rectangles into groups $G_1, G_2, \ldots, G_s$ so that G_1 contains the $k \in \mathbb{Z}^+$ largest rectangles in terms of area, G_2 contains the next k largest rectangles, and so on.

A process of *elimination* is employed by Karmarkar and Karp [68] whereby the list $\mathcal{L}$ is partitioned into narrow and wide rectangles and packing resumes with wide rectangles, since the narrow rectangles may be inserted amongst the wide rectangles.

Appendix C

Statistical tools

This appendix serves to describe several statistical tools that are employed in this dissertation. Packing problems often involve large sized data sets and these tools are used to gain more information about the data as well as during the comparison of algorithms.

The three most commonly used measures of the location of central tendency are the mode, the median, and the mean [81]. The mode is the value which occurs most frequently, the median is the midpoint of all the individual measures when arranged in order of magnitude, and the mean is the arithmtic average of all the individual measures.

The *mean* of a data set is computed as

$$\bar{x} = \frac{1}{n}\sum x_i, \tag{C.1}$$

where n is the input size and x_i is the *i–th* datum [102]. *Variance* (σ^2) is a measure of variability and it is defined as the average of the squared deviations from the arithmetic mean [1], that is

$$\sigma = \frac{1}{n-1}\sum (x_i - \bar{x})^2 . \tag{C.2}$$

It is of interest to know how the data are spread out in relation to the mean of the distribution and a common measure used in this regard is the *standard deviation* [93]. It is only used when the mean is the measure of the centre since other measures such as the mode and median may be used to locate central tendency. The *standard deviation* (σ) is the square root of the variance computed as

$$\sigma^2 = \sqrt{\frac{1}{n-1}\sum (x_i - \bar{x})^2} . \tag{C.3}$$

Clearly, variance and standard deviation provide the same information and one can always be obtained from the other.

Confidence levels deal with the precision of an estimated value by stating limits within which it may be reasonable to assert that the true value lies [35]. The range of values between the limits is referred to as a *confidence interval.* Loosely speaking, a confidence level is the probability value $1 - \alpha$ associated with a confidence interval and is often expressed as a percentage. Normally values from 90% upwards are used so that conclusions may be deemed accurate. In this dissertation the significance test is used to dispute some

stated claim. This claim is called the null hypotheses and is a statement that typically declares "no difference" between two values and a significane level (α) is selected such that $100(\alpha)\%$ of the time the null hypothetsis is rejected.

The *students' t-test* is used when comparing the means of two data sets [93]. When comparing the means of more than two sets of data an ANalysis Of VAriance (ANOVA) is used [93]. In this case the null hypothesis states that the means are all equal. If there are observed differences among sample means, the ANOVA is used to assess whether the differences between means are statistically significant. The F-distribution is used in the ANOVA and the $F_{calculated}$ value is used to compare the variation among sets with variation within the set. When $F_{calculated} > F_{critical}$ there are significant differences between the observed means, where $F_{critical}$ is a tabulated value from a so-called F–table [105, pages 110–130]. Unfortunately the results of the ANOVA only indicate whether there are significant differences between observed means, but if so, it is not clear which means differ, because it may be that only one mean differs from the rest or that all of them are different, for example. In the case where there are significant differences, the method of *Least Significant Difference* (LSD) is used to determine which sample means differ from which others.

The *chi-squared test* is used to assess whether the observed differences in the number of individuals in various categories are statistically significant at a particular level of significance [93]. In this dissertation it is used to compare the frequency with which an algorithm obtains the smallest packing height. The $\chi^2_{calculated}$ value is calculated by using the values of the expected frequencies (expected if the null hypothesis is true) and observed frequencies (actual performance of an experiment) such that

$$\chi^2_{calculated} = \sum \frac{(O - E)^2}{E}, \tag{C.4}$$

where O and E denote the observed and expected frequencies respectively. If $\chi^2_{calculated} > \chi^2_{critical}$, then there is a significant difference between the frequencies, where $\chi^2_{critical}$ is a tabulated value from a so-called χ^2–table [105, pages 98–99].

Appendix D

Mean packing heights and execution times for online algorithms

This appendix contains a summary of the mean packing heights obtained by the online algorithms described in Chapter 3 when tested on the 542 benchmark data described in §1.6. There are three tables representing the three data traversal orders—forward, reverse and random—shown in Tables D.1–D.3 respectively.

The mean execution times expended by the online algorithms are also shown in Tables D.4–D.6 for the three traversal orders respectively. All the tables in this appendix are partitioned into four segments by means of vertical double lines. The first segment represents the results obtained by the online level algorithms, followed by the shelf algorithms in the second segment. The third and the fourth segments represent results achieved by the special case algorithms obeying the tetris constraint and the plane algorithms respectively.

Author	Data Sets	# of Rec	OPT	NFL	FFL	BFL	MNFL	MFFL	MBFL	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{40.8}$	$HS_{80.5}$	$HS_{120.8}$	SDev	SDiff	SAve	SAve2	CA	CPF	CFF	CC	$Azar_{0.25}$	BiNFL	OF
Bousley [9, 10]	U_1	10	-	1016	1016	1016	1016	1016	1016	1536	1536	1536	1266.56	1500	1500	1503	1503	1503	1247.309	1016	1016	1016	1016	1086	1016	1086
	U_2	20	-	1823	1615	1615	1961	1667	1667	2432	2304	2304	2143.04	2500	2500	1693.94	1686	1797.55	1815.556	1604	1724	1823	1724	2367	1823	1993
	U_3	30	-	2459	2123	2123	2883	2254	2155	3584	2944	2944	3193.15	4000	4000	4098.04	4586	3196.69	2551.829	2395	2443	2459	2443	3317	2459	3180
	U_4	50	-	4388	3983	3950	5019	3900	3856	6208	5184	5184	5206.59	6625	6625	4445.10	4540	4799.71	4454.351	4387	4336	4388	4336	6125	4388	5377
	V_1	10	-	30	30	30	33	28	28	44	44	44	38.99	35	35	33.77	38	32.88	32.98	27	27	30	25	35.00	30	35
	V_2	17	-	38	37	37	40	36	36	47	47	47	46.67	58.75	58.75	50.27	57	47.40	41.64	37	37	38	37	78.00	38	34
	V_3	21	-	45	39	42	51	47	47	53	53	53	54.77	71.25	71.25	52.24	59	58.12	54.77	42	42	45	42	71.00	45	33
	V_4	7	-	27	27	27	35	35	35	24	24	24	46.13	40	40	30.00	30	30.00	40.18	27	23	27	23	45.00	27	24
	V_5	14	-	47	47	47	47	47	47	66	50	50	62.10	92.5	92.5	59.77	62	58.30	56.62	47	47	47	47	118.00	47	71
	V_6	15	-	41	41	41	56	56	56	64	64	64	65.16	85	85	70.63	80	64.28	59.87	41	40	37	36	78.00	41	44
	V_7	8	-	21	21	21	56	56	56	54	54	54	41.55	47.5	47.5	25.50	33	50.04	50.04	21	21	21	21	48.00	21	20
	V_8	13	-	64	41	41	60	47	47	64	64	64	71.74	110	110	67.50	66	49.33	42.58	64	54	51	44	138.00	64	54
	V_9	18	-	76	76	76	89	77	77	90	90	90	102.27	153.75	153.75	127.71	124	140	96.67	76	75	76	75	126.00	76	104
	V_{10}	13	-	86	86	86	85	85	85	99	99	99	93.17	95.625	95.625	114.00	150	96	96.26	86	83	86	83	144.00	86	83
	V_{11}	15	-	94	94	94	90	84	84	100	100	100	151.25	191.25	191.25	135.00	174	115.5	106.00	94	94	91	86	156.00	94	81
	V_{12}	22	-	120	111	111	136	120	120	136	136	136	156.19	225	225	237.66	240	336	190.95	120	120	120	120	172.50	120	111
Burke *et al.* [18]	B_1	10	40	66	46	66	94	66	66	60	60	60	81.86	100.00	125.00	83.87	58	69.51	69.51	66	66	54	52	90.00	66	52
	B_2	20	50	84	67	67	85	67	67	100	100	100	110.61	157.50	157.50	113.47	120	111.61	88.26	78	84	84	84	105.00	84	81
	B_3	30	50	105	95	95	127	97	91	72	72	72	176.34	292.50	292.50	97.29	108	115.12	101.34	105	104	96	95	191.50	105	87
	B_4	40	80	190	217	217	218	167	191	328	328	328	379.10	275.00	400.00	276.06	296	233.45	192.79	190	181	169	149	284.00	190	121
	B_5	50	100	143	138	138	177	142	142	216	208	208	265.13	293.75	331.25	170.70	168	169.54	186.72	143	137	143	133	247.50	143	184
	B_6	60	100	198	165	172	239	138	184	144	140	140	293.77	290.63	346.88	293.26	302	330.00	193.60	198	189	185	174	353.13	198	125
	B_7	70	100	349	275	345	417	309	329	290	290	290	459.42	655.00	727.50	574.00	574	574.00	336.99	349	300	337	295	800.00	349	241
	B_8	80	80	229	227	227	284	188	225	156	156	156	327.83	318.75	371.88	173.02	228	270.46	259.92	229	223	189	182	348.25	229	176
	B_9	100	150	379	347	337	437	311	322	296	284	284	483.23	531.25	634.38	286.83	273	408.55	340.73	379	370	346	327	684.00	379	258
	B_{10}	200	150	408	379	393	479	320	321	304	300	300	547.62	479.06	562.19	342.39	439	753.02	404.37	408	377	317	306	541.81	408	247
	B_{11}	300	150	432	374	391	540	290	355	276	250	250	596.95	476.88	468.13	284.64	308	469.35	415.59	432	415	376	343	668.13	432	237
	B_{12}	500	300	1134	1033	1022	1406	935	942	572	556	556	1576.61	1284.38	1381.25	774.17	770	1128.56	1052.05	1134	1079	948	793	1749.00	1134	691
Christofides [22]	G_1	16	-	35	33	33	30	28	28	38	34	34	37.76	52.5	52.5	48	48	48	40.12	35	35	33	33	57.50	35	32
	G_2	23	-	106	106	106	123	95	95	112	112	112	142.75	210	210	143.07	146	155	124.84	106	95	97	97	180.00	106	74
	G_3	62	-	975	743	743	975	745	745	1184	928	928	1153.41	1260	1260	846.05	845	845.40	802.20	975	975	975	975	1376.50	975	907

Table D.1: *Summary of mean packing heights in the forward traversal order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author.*

Author	Data Sets	# of Rec	OPT	NFL	FFL	BFL	MNFL	MFFL	MBFL	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{8,s}$	$HS_{8,s}$	$HS_{12,s}$	SDev	SDiff	SAvu	SAvg2	CA	CPF	CFF	CC	$Azar_{0.25}$	BINFL	OF
Hopper et al. [57, 58]	C1	16/7	20	36.00	34.00	34.00	38.33	34.67	35.33	43.00	43.00	43.00	61.26	94.58	94.58	62.14	60.33	56.30	42.78	35.33	33.00	30.33	27.33	90.25	36.00	23.33
	C2	25	15	24.00	23.33	23.33	24.33	20.33	22.00	23.33	23.33	23.33	40.72	52.92	61.67	33.02	27.67	30.03	24.09	24.00	23.67	21.00	19.33	57.00	24.00	17.67
	C3	28/9	30	47.00	45.00	45.00	53.33	44.00	44.00	54.67	54.67	54.67	93.69	123.75	148.13	60.99	59.00	65.12	55.15	47.00	46.33	44.33	43.00	123.67	47.00	33.67
	C4	49	60	137.00	115.00	119.00	175.67	122.33	140.33	133.00	127.67	127.67	207.90	248.13	288.13	201.13	223.33	194.30	160.70	137.00	129.33	126.00	117.33	249.33	137.00	118.00
	C5	72/3	90	189.67	177.33	176.00	238.00	174.67	173.33	156.33	145.67	145.67	278.25	316.88	357.50	204.62	204.67	203.35	181.27	189.00	178.67	164.67	150.67	368.04	189.67	176.00
	C6	97	120	275.33	254.67	252.33	349.33	234.67	239.67	251.33	235.33	235.33	377.40	385.83	474.58	419.70	464.33	475.83	287.09	273.00	252.67	250.00	221.33	443.08	275.33	233.33
	C7	196/7	240	672.00	613.33	627.33	845.67	539.00	557.00	457.33	445.33	445.33	881.90	758.75	956.25	527.40	543.67	781.26	608.84	672.00	592.33	551.00	497.67	875.17	672.00	508.33
	N1	17	200	374.40	352.20	352.20	451.20	371.80	376.80	547.20	547.20	547.20	803.79	935.00	1055.00	583.73	576.40	567.94	425.97	372.60	350.40	329.60	295.60	866.10	374.40	200.00
	N2	25	200	369.60	356.00	356.00	458.20	352.60	362.40	444.80	432.00	432.00	716.39	986.25	1156.25	507.44	478.00	502.77	406.99	367.80	344.00	347.80	311.20	817.35	369.60	200.00
	N3	29	200	429.40	421.60	421.60	503.00	395.40	408.60	464.60	464.60	464.60	758.88	970.94	1153.44	572.58	618.80	615.61	495.21	429.40	398.60	391.00	345.00	922.24	429.40	200.00
	N4	49	200	452.20	443.00	444.40	595.80	401.40	447.60	406.00	399.60	399.60	751.18	972.50	1240.63	677.94	705.80	705.83	476.02	452.20	408.40	407.00	366.40	970.48	452.20	200.00
	N5	73	200	512.40	462.00	462.00	605.00	414.40	431.00	424.60	418.20	418.20	780.12	823.44	1154.06	468.59	485.80	636.18	472.67	512.40	476.60	407.40	375.80	943.89	512.40	365.80
	N6	97	200	517.80	506.00	502.20	642.80	429.80	474.80	448.60	448.60	448.60	857.46	889.69	1169.69	733.94	742.80	863.61	535.74	517.80	469.80	407.80	371.20	948.31	517.80	367.20
	N7	197	200	556.40	536.00	536.40	779.40	483.20	512.20	455.00	435.80	435.80	1020.15	816.25	1002.81	569.29	598.60	842.33	536.35	556.40	511.20	450.60	404.40	879.21	556.40	614.20
	T1	17	200	395.80	367.80	367.80	445.60	373.40	403.80	578.00	565.20	565.20	749.62	881.88	996.88	524.76	544.00	594.20	491.76	377.60	367.20	363.80	334.00	774.58	395.80	200.00
	T2	25	200	384.80	371.40	374.80	452.00	375.60	380.60	548.00	528.80	528.80	745.94	911.88	1041.88	508.73	501.40	533.43	417.03	384.80	360.80	344.40	318.80	818.28	384.80	221.20
	T3	29	200	421.80	415.00	415.00	513.20	403.20	412.00	508.00	508.00	508.00	751.59	925.00	1051.25	608.72	637.20	680.85	477.44	421.80	402.60	369.20	341.20	841.15	421.80	200.00
	T4	49	200	460.00	445.00	445.00	624.00	406.20	447.40	418.00	411.60	411.60	737.53	953.75	1239.38	581.43	603.40	694.19	476.59	460.00	419.80	405.40	364.20	978.23	460.00	200.00
	T5	73	200	514.40	472.40	472.40	588.00	413.80	432.40	402.00	402.00	402.00	781.62	800.00	1134.38	443.87	487.80	676.02	472.92	514.40	476.40	408.60	378.40	860.88	514.40	349.60
	T6	97	200	533.00	507.60	509.20	644.20	436.60	481.60	435.80	435.80	435.80	922.76	927.19	1234.69	778.70	796.40	856.78	547.36	533.00	478.80	432.80	409.80	1044.93	533.00	355.00
	T7	199	200	515.20	498.60	498.60	676.20	462.00	495.40	484.80	473.60	473.60	984.38	832.50	1013.75	749.87	757.80	827.56	509.70	515.20	472.40	409.00	371.20	920.14	515.20	426.40
Munford et al. [94]	nice.25	25	100	171.85	158.42	158.35	214.72	164.81	166.12	216.48	209.44	210.72	299.66	398.25	448.50	253.80	256.89	261.92	188.73	171.31	161.70	159.91	149.53	318.65	171.85	157.74
	nice.50	50	100	178.39	164.33	164.56	243.88	169.67	172.10	210.24	206.24	205.60	302.09	354.00	458.25	244.62	247.30	260.61	184.90	178.39	163.30	166.48	152.44	300.91	178.39	172.28
	nice.100	100	100	185.06	173.76	173.83	283.32	169.40	174.94	187.76	183.44	182.96	348.36	315.38	381.25	256.03	259.38	279.78	187.37	185.06	166.43	170.37	156.69	286.63	185.06	191.33
	nice.200	200	100	194.78	184.50	184.68	326.44	174.87	181.43	184.40	181.60	182.32	450.75	320.63	315.13	249.69	250.32	296.16	191.88	194.78	173.21	172.53	157.77	267.03	194.78	203.78
	nice.500	500	100	201.01	192.48	192.48	377.03	173.52	179.28	163.20	160.80	160.80	676.77	445.16	326.41	257.68	258.57	297.48	190.25	201.01	175.26	173.57	161.02	268.59	201.01	237.83
	path.25	25	100	274.11	200.06	208.89	313.59	228.07	229.38	263.02	261.10	261.10	481.75	477.75	540.13	397.64	402.42	393.97	276.44	271.81	258.66	244.68	223.59	656.80	274.11	130.83
	path.50	50	100	349.59	254.85	262.65	415.57	271.23	273.54	277.82	274.90	274.74	577.39	535.91	588.03	477.02	477.37	481.37	332.88	349.28	322.12	281.00	258.61	766.01	349.59	131.77
	path.100	100	100	448.55	325.86	329.88	547.06	327.22	333.17	278.43	275.83	275.57	700.37	589.73	644.29	624.82	624.17	656.74	409.77	447.00	404.69	324.86	298.75	825.62	448.55	135.07
	path.200	200	100	574.00	436.98	443.08	728.43	393.89	402.53	282.85	277.99	276.97	891.24	690.79	684.87	723.27	726.52	876.15	501.79	573.39	505.24	373.77	340.90	825.79	574.00	145.43
	path.500	500	100	792.02	639.29	641.16	1061.94	501.59	505.67	260.13	258.03	256.43	1295.23	902.03	746.09	985.90	994.24	1409.59	686.85	792.02	709.92	461.36	430.57	813.76	792.02	169.68

Table D.1 (continued) : *Summary of mean packing heights in the forward order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author. The bold faced entries represent instances where the optimal height was achieved.*

Author	Data Sets	# of Rec	OPT	NFL_R	FFL_R	BFL_R	$MNFL_R$	$MFFL_R$	$MBFL_R$	$NFS_{0.5R}$	$FFS_{0.5R}$	$BFS_{0.5R}$	$HS_{8,8R}$	$HS_{8,8R}$	$HS_{12,8R}$	$SDev_R$	$SDif_R$	$SAve_R$	$SAve2_R$	CA_R	CPF_R	CFF_R	CC_R	$Azar_{0.25R}$	$BiNFL_R$	OF_R
Beasley [9, 10]	U_1	10	-	1016	1016	1016	1184	1016	1114	1536	1536	1536	1266.56	1500	1500	1182.24	1135	1107.67	1127.02	1016	1016	1016	1016	1086.00	1016	1086
	U_2	20	-	1821	1378	1429	2078	1378	1411	2432	2304	2176	2143.04	2500	2500	1746.96	1731	1740.04	1656.61	1716	1741	1821	1741	2367.00	1821	1975
	U_3	30	-	2459	2374	2374	2872	2237	2240	3584	3072	3072	3193.15	4000	4000	3025.26	3242	2965.18	2438.11	2360	2360	2459	2360	3317.00	2459	2892
	U_4	50	-	4423	3717	3623	4953	3786	3731	6208	4800	4800	5206.59	6625	6625	4201.02	4287	4414.21	4198.20	4380	4380	4423	4380	6125.00	4423	5525
	V_1	10	-	25	25	25	31	30	31	44	44	44	38.99	35.00	35.00	43.86	48	40.54	37.03	25	23	25	23	35.00	25	28
	V_2	17	-	36	33	36	40	33	33	47	47	47	46.67	58.75	58.75	67.07	67	65.53	52.79	36	33	36	33	78.00	36	38
	V_3	21	-	46	42	42	49	38	42	53	53	53	54.77	71.25	71.25	72.00	72	72.00	54.65	45	45	46	45	71.00	46	38
	V_4	7	-	27	27	27	37	34	37	24	24	24	46.13	40.00	40.00	42.02	42	40.53	38.13	27	24	27	24	45.00	27	27
	V_5	14	-	48	48	48	66	47	55	66	66	66	62.10	92.50	92.50	71.46	82	53.46	52.80	48	48	48	48	118.00	48	66
	V_6	15	-	42	50	50	44	40	40	64	64	64	65.16	85.00	85.00	67.85	74	77.00	61.29	42	42	36	35	78.00	42	33
	V_7	8	-	21	21	21	25	23	24	54	54	54	41.55	47.50	47.50	35.81	43	40.00	28.86	21	21	20	20	48.00	21	20
	V_8	13	-	64	57	57	69	44	41	64	64	64	71.74	110.00	110.00	53.86	52	72.00	65.82	64	54	64	54	138.00	64	70
	V_9	18	-	80	80	80	100	87	89	90	90	90	102.27	153.75	153.75	101.85	75	94.17	84.55	73	68	80	68	126.00	80	87
	V_{10}	13	-	85	85	85	87	86	86	99	99	99	93.17	95.63	95.63	168.00	172	180.00	154.56	85	85	85	85	144.00	85	83
	V_{11}	15	-	76	76	76	128	119	122	100	100	100	151.25	191.25	191.25	160.93	163	134.19	123.26	76	76	76	76	156.00	76	74
	V_{12}	22	-	138	129	129	120	111	111	136	136	136	156.19	225.00	225.00	237.03	240	235.68	143.77	138	132	138	132	172.50	138	103
Burke *et al.* [18]	B_1	10	40	66	60	60	72	66	66	60	60	60	81.86	100.00	125.00	63.44	74	66.81	66.74	66	52	66	52	90.00	66	60
	B_2	20	50	91	85	78	97	85	80	100	100	100	110.61	157.50	157.50	92.41	106	97.62	96.44	91	88	91	88	105.00	91	86
	B_3	30	50	117	99	99	134	107	110	72	72	72	176.34	292.50	292.50	181.06	167	131.29	121.93	117	99	94	93	191.50	117	124
	B_4	40	80	222	217	217	270	205	241	328	328	328	379.10	275.00	400.00	378.81	344	302.75	251.48	222	157	177	122	284.00	222	171
	B_5	50	100	151	143	143	185	145	144	216	208	208	265.13	293.75	331.25	182.06	176	188.63	167.64	151	139	150	142	247.50	151	194
	B_6	60	100	198	179	179	234	175	186	144	140	140	293.77	290.63	346.88	162.96	169	210.67	184.51	198	197	185	160	353.13	198	140
	B_7	70	100	385	340	340	462	252	313	290	290	290	459.42	655.00	727.50	502.53	586	449.13	351.12	385	385	318	268	800.00	385	255
	B_8	80	80	203	191	191	297	199	220	156	156	156	327.83	318.75	371.88	165.80	154	360.40	227.16	203	203	174	151	504.50	203	194
	B_9	100	150	424	359	388	445	315	330	296	284	284	483.23	531.25	634.38	356.08	362	409.09	379.37	422	388	386	357	684.00	424	228
	B_{10}	200	150	423	388	388	376	317	326	304	300	300	547.62	479.06	562.19	290.98	376	383.87	318.55	423	403	368	321	533.06	423	304
	B_{11}	300	150	413	393	393	471	311	323	276	250	250	596.95	476.88	468.13	376.09	402	536.20	356.14	413	377	364	332	670.31	413	243
	B_{12}	500	300	1196	1155	1155	1360	857	884	572	556	556	1576.61	1284.38	1381.25	801.49	944	1195.66	1017.97	1196	1054	841	716	1520.88	1196	728
Christofides [22]	G_1	16	-	36	36	36	36	33	36	38	38	38	37.76	52.50	52.50	39.67	46	31.13	34.17	36	36	36	36	57.50	36	37
	G_2	23	-	102	102	102	121	106	112	112	112	112	142.75	210.00	210.00	114.65	125	122.30	121.48	102	101	102	101	180.00	102	100
	G_3	62	-	972	972	972	1019	1001	1004	1184	1184	1184	1153.41	1260.00	1260.00	1089.67	1056	1065.08	1065.33	972	972	972	972	1376.50	972	1375

Table D.2: *Summary of mean packing heights in the reverse traversal order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author.*

Author	Data Sets	# of Rec	OPT	NFI_H	FFI_H	BFI_H	$MNFI_H$	$MFFI_H$	$MBFI_H$	$NFS_{0.5,H}$	$FFS_{0.5,H}$	$BFS_{0.5,H}$	$HS_{8,\infty,H}$	$HS_{8,5,H}$	$HS_{12,\infty,H}$	$SD_{div,H}$	$SDiff_R$	$SA_{ver,H}$	$SA_{ver2,R}$	CA_R	CPF_R	CFF_H	CC_R	$Azuro_{0.25,R}$	$BiNFI_{\infty,H}$	OF_H
Hopper *et al.* [57, 58]	C1	16/7	20	36.33	34.67	34.67	54.67	34.67	43.00	43.00	43.00	43.00	61.26	94.58	94.58	45.51	48.67	40.22	40.42	36.33	35.67	34.33	33.33	90.25	36.33	32.67
	C2	25	15	23.67	23.33	23.33	28.67	21.67	23.33	23.33	23.33	23.33	40.72	52.92	61.67	22.23	24.33	25.37	25.60	23.67	22.67	23.33	21.67	57.00	23.67	27.33
	C3	28/9	30	46.33	43.67	43.67	63.67	43.33	46.67	54.67	54.67	54.67	93.69	123.75	148.13	52.48	51.67	49.27	51.51	46.33	44.00	46.00	45.33	123.67	46.33	52.67
	C4	49	60	138.00	129.00	129.00	168.00	131.67	134.33	133.00	127.67	127.67	207.90	248.13	288.13	183.31	188.33	209.02	141.05	138.00	128.33	121.00	114.33	239.33	138.00	131.00
	C5	72/3	90	188.33	170.67	170.67	245.00	158.67	166.00	156.33	145.67	145.67	278.25	316.88	357.50	237.72	242.33	321.33	185.28	188.33	180.67	162.33	153.33	364.29	188.33	206.67
	C6	97	120	276.00	265.33	259.00	369.67	238.00	246.00	251.33	235.33	235.33	377.40	385.83	474.58	291.92	276.33	369.99	270.30	273.67	259.67	246.00	224.67	428.08	276.00	221.67
	C7	196/7	240	655.67	600.67	604.33	801.33	517.33	563.67	457.33	440.00	440.00	881.90	758.75	956.25	607.27	712.67	1018.72	617.19	653.67	571.67	517.33	472.33	1001.42	655.67	608.00
	N1	17	200	408.00	358.60	358.60	513.60	387.80	438.00	547.20	547.20	547.20	803.80	935.00	1055.00	437.13	444.60	423.58	387.53	395.00	378.60	369.00	346.40	866.10	408.00	432.00
	N2	25	200	391.40	361.40	361.40	537.40	405.80	444.20	444.80	432.00	432.00	716.39	986.25	1156.25	499.96	470.20	443.24	425.34	391.40	355.60	371.40	328.80	817.35	391.40	425.80
	N3	29	200	428.20	388.20	388.20	587.20	427.80	490.20	464.60	464.60	464.60	758.88	970.94	1153.44	508.60	477.40	424.68	420.21	428.20	408.40	399.80	379.60	922.24	428.20	412.00
	N4	49	200	482.80	448.80	448.80	689.80	474.60	500.40	406.00	399.60	399.60	751.18	972.50	1240.63	456.76	455.00	480.94	462.57	482.80	442.40	449.40	401.20	967.98	482.80	501.20
	N5	73	200	501.80	463.20	461.20	747.00	409.60	501.40	424.60	418.20	418.20	780.12	823.44	1154.06	455.13	446.20	486.76	469.27	501.80	452.00	422.80	373.20	899.51	501.80	521.00
	N6	97	200	529.60	502.00	502.00	706.40	446.20	546.00	448.60	435.80	435.80	857.46	889.69	1169.69	416.31	465.00	528.39	506.75	529.60	489.00	466.00	424.20	972.06	529.60	513.60
	N7	197	200	588.40	552.40	552.40	890.80	446.40	630.60	455.00	435.80	435.80	1020.15	816.25	1002.81	434.08	428.80	520.82	522.83	588.40	522.80	467.00	431.20	866.71	588.40	576.80
	T1	17	200	376.60	367.80	367.80	551.80	431.00	486.40	578.00	565.20	565.20	749.62	881.88	996.88	495.58	531.20	452.62	456.54	373.00	369.20	364.40	359.20	774.58	376.60	408.40
	T2	25	200	408.20	380.20	380.20	591.60	430.80	448.80	548.00	528.80	528.80	745.94	911.88	1041.88	495.77	517.60	435.44	424.93	408.20	383.60	364.40	350.00	818.28	408.20	466.80
	T3	29	200	429.20	411.60	411.60	626.80	421.60	485.00	508.00	508.00	508.00	751.59	925.00	1051.25	556.33	505.80	469.43	446.57	429.20	409.20	402.40	385.60	841.15	429.20	465.60
	T4	49	200	475.60	428.80	428.80	661.40	466.40	475.80	418.00	398.80	398.80	737.53	953.75	1239.38	449.94	448.40	477.48	442.16	475.60	438.00	443.00	395.40	975.73	475.60	511.40
	T5	73	200	501.40	458.60	458.60	737.00	407.40	500.80	402.00	402.00	402.00	781.62	800.00	1134.38	434.72	413.00	463.91	446.81	501.40	445.20	421.40	371.80	884.63	501.40	525.20
	T6	97	200	551.00	514.00	514.00	723.80	459.60	526.00	435.80	435.80	435.80	922.76	927.19	1234.69	439.04	501.40	523.70	506.11	551.00	489.40	464.80	424.20	1052.43	551.00	535.60
	T7	199	200	521.80	511.80	508.60	825.60	423.00	514.20	484.80	473.60	473.60	984.38	832.50	1013.75	442.46	463.20	545.31	508.89	521.80	488.80	453.80	409.60	860.45	521.80	483.00
Mingozzi *et al.* [94]	nice.25	25	100	180.87	176.56	176.90	261.31	200.08	226.13	215.84	214.56	215.20	299.66	398.25	448.50	217.00	223.74	205.37	200.47	180.73	168.54	173.10	160.45	319.90	180.87	243.98
	nice.50	50	100	185.88	180.02	180.40	272.64	193.16	212.23	210.40	209.12	208.96	302.09	354.00	458.25	201.87	209.93	200.50	193.18	185.88	168.88	176.40	161.64	303.66	185.88	303.56
	nice.100	100	100	190.46	185.06	185.06	307.62	195.84	214.52	187.12	185.84	185.84	348.36	315.38	381.25	192.60	202.73	202.64	194.90	190.46	171.32	177.66	162.79	288.50	190.46	352.79
	nice.200	200	100	197.27	192.56	192.56	343.36	187.77	204.11	184.80	183.36	183.12	450.75	320.63	315.13	180.46	197.67	202.03	190.56	197.27	176.54	179.54	164.47	274.10	197.27	428.02
	nice.500	500	100	205.69	199.66	199.66	385.87	195.28	210.52	162.80	162.00	162.40	676.77	445.16	326.41	159.97	185.14	199.84	182.25	205.69	180.08	179.92	164.40	271.09	205.69	464.36
	path.25	25	100	291.53	262.69	270.77	359.27	295.88	314.78	263.02	263.02	263.02	481.75	477.75	540.13	318.26	324.70	341.50	307.98	289.08	268.10	243.90	220.38	656.80	291.53	244.97
	path.50	50	100	369.93	324.56	333.18	472.59	326.21	351.02	279.18	279.18	277.10	577.39	535.91	588.03	327.02	344.02	404.60	343.16	366.23	327.49	291.62	263.77	766.01	369.93	278.51
	path.100	100	100	463.11	395.21	401.50	598.58	372.90	425.31	278.39	277.65	278.05	700.37	589.73	644.29	373.10	401.49	482.39	400.24	461.59	404.80	330.01	299.68	825.84	463.11	323.26
	path.200	200	100	594.62	496.16	501.55	781.51	445.23	484.71	283.44	282.94	281.11	891.24	690.79	684.87	403.00	443.02	586.65	478.62	593.75	523.72	389.74	357.56	828.07	594.62	352.89
	path.500	500	100	814.84	699.70	700.01	1141.23	601.67	646.76	260.33	259.38	257.88	1295.23	902.03	746.09	469.13	494.84	767.60	658.89	813.22	716.19	475.81	439.17	811.53	814.84	391.84

Table D.2 (continued) : *Summary of mean packing heights in the reverse order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author.*

Author	Data Sets	# of Rec	OPT	NFL_{Rand}	FFL_{Rand}	BFL_{Rand}	$MNFL_{Rand}$	$MFFL_{Rand}$	$MBFL_{Rand}$	$NFS_{0.5Rand}$	$FFS_{0.5Rand}$	$BFS_{0.5Rand}$	$HS_{6,6Rand}$	$HS_{6,8Rand}$	$HS_{12,5Rand}$	$SDev_{Rand}$	$SDiff_{Rand}$	$SAve_{Rand}$	$SAve2_{Rand}$	CA_{Rand}	CPF_{Rand}	CFF_{Rand}	CC_{Rand}	$Azar_{0.25Rand}$	$BiNFL_{Rand}$	OF_{Rand}
Beasley [9, 10]	U_1	10	-	1086	1020	1020	1086	1020	1020	1664	1536	1536	1266.56	1500	1500	1438.07	1652	1392.14	1216.22	1086	1086	1086	1086	1086	1086	1086
	U_2	20	-	1759	1604	1604	1908	1564	1584	2560	2432	2432	2143.04	2500	2500	1903.34	1857	1730.95	1822.24	1672	1689	1759	1689	2367	1759	2166
	U_3	30	-	2491	2174	2113	2633	2202	2110	3072	2816	2816	3193.15	4000	4000	3396.51	3743	3136.74	2530.33	2421	2421	2491	2421	3317	2491	3054
	U_4	50	-	4485	3878	3878	5075	3897	3770	6080	5184	5184	5206.59	6625	6625	5134.64	4908	4837.97	4424.54	4171	4485	4485	4485	6125	4485	5373
	V_1	10	-	40	35	35	33	28	28	44	44	44	38.99	35.00	35.00	35.07	38	52.44	45.12	35	35	40	35	45	40	30
	V_2	17	-	48	42	42	53	33	42	47	47	47	46.67	58.75	58.75	42.41	43	39.52	44.24	48	48	48	48	88	48	42
	V_3	21	-	54	41	46	70	45	48	53	53	53	54.77	71.25	71.25	59.41	60	72.00	54.29	51	54	54	54	71	54	41
	V_4	7	-	27	27	27	35	35	35	24	24	24	46.13	40.00	40.00	30.00	30	30.00	41.29	27	23	27	23	45	27	24
	V_5	14	-	49	49	49	70	49	49	66	66	66	62.10	92.50	92.50	58.41	59	60.00	58.70	49	49	49	49	118	49	78
	V_6	15	-	56	42	42	56	46	46	64	64	64	65.16	85.00	85.00	67.94	70	62.49	54.72	56	56	46	42	78	56	45
	V_7	8	-	21	21	21	21	21	21	54	54	54	41.55	47.50	47.50	40.00	40	40.00	28.86	21	21	21	21	48	21	20
	V_8	13	-	55	54	54	75	54	62	64	64	64	71.74	110.00	110.00	72.00	72	72.00	61.63	55	45	49	39	138	55	52
	V_9	18	-	90	78	81	120	76	76	90	90	90	102.27	153.75	153.75	108.24	110	120.00	118.98	84	87	83	77	126	90	76
	V_{10}	13	-	116	86	86	142	86	86	99	99	99	93.17	95.63	95.63	163.47	164	180.00	149.82	116	116	116	116	204	116	84
	V_{11}	15	-	127	76	94	137	90	90	100	100	100	151.25	191.25	191.25	121.94	126	174.00	120.91	127	116	98	97	156	127	76
	V_{12}	22	-	146	103	121	155	121	121	136	136	136	156.19	225.00	225.00	170.86	180	147.85	158.16	146	146	129	129	210	146	95
Burke *et al.* [18]	B_1	10	40	60	60	60	66	66	66	60	60	60	81.86	100.00	125.00	91.90	88	87.00	74.23	60	60	60	60	90.00	60	60
	B_2	20	50	86	80	80	91	82	82	112	100	100	110.61	157.50	157.50	114.67	138	131.25	100.32	80	86	74	71	105.00	86	99
	B_3	30	50	111	106	106	125	102	118	72	72	72	176.34	292.50	292.50	215.20	248	192.00	128.70	111	99	86	86	191.50	111	122
	B_4	40	80	180	164	164	200	171	171	328	328	328	379.10	275.00	400.00	309.58	432	281.96	190.91	180	166	191	176	284.00	180	131
	B_5	50	100	166	154	156	206	148	156	208	208	208	265.13	293.75	331.25	177.27	182	188.67	185.03	166	160	164	152	266.25	166	196
	B_6	60	100	201	194	194	251	155	172	152	140	140	293.77	290.63	346.88	155.70	189	276.78	199.64	201	201	196	190	362.50	201	174
	B_7	70	100	359	346	346	386	342	342	290	290	290	459.42	655.00	727.50	286.82	296	436.57	392.71	359	327	289	271	800.00	359	214
	B_8	80	80	264	254	254	331	240	239	156	156	156	327.83	318.75	371.88	276.67	302	345.80	261.42	264	225	209	186	348.25	264	192
	B_9	100	150	382	322	322	404	282	282	284	284	284	483.23	531.25	634.38	562.54	664	864.00	375.44	382	349	342	317	705.88	382	219
	B_{10}	200	150	431	400	398	455	312	350	310	300	300	547.62	479.06	562.19	635.79	637	638.14	443.01	431	403	407	370	572.44	431	281
	B_{11}	300	150	402	376	384	513	268	329	264	250	250	596.95	476.88	468.13	265.79	293	371.50	310.07	402	371	346	309	655.00	402	261
	B_{12}	500	300	1177	1097	1097	1577	1049	1115	556	556	556	1576.61	1284.38	1381.25	720.94	683	1462.92	1133.72	1177	997	890	789	1627.13	1177	731
Christofides [22]	G_1	16	-	41	28	28	44	28	28	38	34	34	37.76	52.50	52.50	48.00	48	48.00	36.49	41	41	36	36	57.50	41	34
	G_2	23	-	137	129	138	175	144	144	112	112	112	142.75	210.00	210.00	103.04	106	186.05	170.60	137	121	103	93	180.00	137	90
	G_3	62	-	938	778	759	1079	786	815	1152	992	960	1153.41	1260.00	1260.00	853.31	845	853.98	835.41	924	912	907	884	1411.50	938	1313

Table D.3: *Summary of mean packing heights in a random traversal order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author.*

Author	Data Sets	# of Rec	OPT	NFL_{Hand}	FFL_{Hand}	BFL_{Hand}	$MNFL_{Hand}$	$MFFL_{Hand}$	$MBFL_{Hand}$	$NFS_{0.5Hand}$	$FFS_{0.5Hand}$	$BFS_{0.5Hand}$	$HS_{4\times8Hand}$	$HS_{8\times8Hand}$	$HS_{8\times16Hand}$	$SDev_{Hand}$	$SDiff_{Hand}$	$SAvel_{Hand}$	$SAvel2_{Hand}$	CA_{Hand}	CFF_{Hand}	CFF_{Hand}	CC_{Hand}	$Azuri_{0.5Hand}$	$BINFL_{Hand}$	OF_{Hand}
Hopper *et al.* [57, 58]	C1	16/7	20	38.67	37.67	37.67	44.67	37.33	38.00	45.67	43.00	43.00	61.26	94.58	94.58	45.56	42.00	56.56	46.88	38.33	36.33	37.67	34.00	90.25	38.67	32.67
	C2	25	15	26.00	25.00	25.00	33.00	23.00	25.00	24.00	24.00	24.00	40.72	52.92	61.67	29.78	28.67	29.90	25.76	26.00	23.33	25.33	23.00	57.00	26.00	28.00
	C3	28/9	30	58.67	54.00	54.00	77.67	51.00	52.67	56.00	56.00	56.00	93.69	123.75	148.13	69.00	72.00	66.61	58.40	58.67	57.67	53.67	51.33	117.42	58.67	58.67
	C4	49	60	135.00	127.33	127.33	158.33	110.67	124.67	133.00	127.67	127.67	207.90	248.13	288.13	122.49	132.67	146.83	130.99	135.00	121.67	113.33	98.67	238.71	135.00	136.33
	C5	72/3	90	195.33	173.67	178.00	235.33	158.67	173.33	156.33	140.33	140.33	278.25	316.88	357.50	206.98	226.33	198.01	181.06	195.33	188.00	175.33	164.00	326.17	195.33	173.67
	C6	97	120	297.33	260.00	260.00	348.33	222.67	240.33	254.00	235.33	235.33	377.40	385.83	474.58	355.43	383.33	402.93	270.36	297.33	278.33	258.33	231.33	459.75	297.33	231.00
	C7	196/7	240	666.33	601.33	606.00	883.00	504.00	580.33	449.33	440.00	440.00	881.90	758.75	956.25	514.54	542.33	872.58	607.33	666.33	582.33	558.00	504.67	954.33	666.33	571.67
	N1	17	200	417.60	386.20	372.80	491.60	444.60	447.40	597.20	565.20	565.20	749.62	881.88	996.88	651.94	698.00	608.23	490.72	416.60	391.60	398.60	362.40	779.58	417.60	375.40
	N2	25	200	448.80	420.80	412.00	499.60	400.60	410.80	460.40	460.40	460.40	617.09	716.63	831.63	425.15	479.00	412.82	365.93	443.60	413.60	431.00	390.80	818.28	448.80	409.60
	N3	29	200	496.80	415.40	432.80	583.40	456.60	471.40	527.20	514.40	514.40	751.59	925.00	1051.25	549.12	524.00	621.56	483.73	496.80	474.00	469.60	420.80	841.15	496.80	463.40
	N4	49	200	522.40	484.80	473.00	664.60	438.60	495.00	414.80	398.80	398.80	737.53	953.75	1239.38	503.29	572.40	634.27	486.26	522.40	496.60	447.20	392.40	979.48	522.40	471.80
	N5	73	200	506.60	470.20	475.40	677.80	467.40	472.20	414.80	405.20	405.20	786.68	802.50	1135.63	490.06	495.80	684.81	526.08	506.60	484.80	444.00	398.00	898.38	506.60	474.40
	N6	97	200	626.60	555.80	556.60	755.20	513.40	571.20	445.40	435.80	435.80	922.76	927.19	1234.69	462.89	505.00	669.43	554.86	626.60	572.00	506.00	448.60	1098.68	626.60	471.60
	N7	197	200	616.80	594.40	594.80	834.20	533.60	556.40	494.40	473.60	473.60	984.38	832.50	1013.75	555.60	549.00	675.88	578.46	616.80	550.40	489.20	446.40	903.58	616.80	524.40
	T1	17	200	407.60	382.00	385.60	512.60	400.00	434.60	547.20	547.20	547.20	803.79	935.00	1055.00	436.41	444.20	506.00	439.53	397.80	357.60	391.40	346.20	866.10	407.60	392.80
	T2	25	200	433.20	392.40	396.20	552.20	432.80	432.20	438.40	432.00	432.00	716.39	986.25	1156.25	572.83	584.20	514.33	444.19	433.20	395.00	396.20	363.00	847.35	433.20	448.00
	T3	29	200	462.20	417.00	419.60	603.40	440.60	451.20	477.40	464.60	464.60	758.88	970.94	1153.44	558.77	616.20	558.58	459.84	462.20	440.20	411.80	380.40	927.24	462.20	403.60
	T4	49	200	522.00	469.80	469.80	625.00	431.00	452.20	399.60	399.60	399.60	751.18	972.50	1240.63	520.27	581.40	568.97	471.57	522.00	476.60	430.40	384.20	984.23	522.00	411.40
	T5	73	200	532.20	508.00	508.00	672.20	450.00	454.60	418.20	418.20	418.20	780.12	823.44	1154.06	508.62	584.60	630.22	521.36	532.20	492.40	449.00	395.80	917.01	532.20	520.00
	T6	97	200	566.40	541.60	541.80	724.80	496.80	561.00	448.60	435.80	435.80	857.46	889.69	1169.69	582.40	605.00	737.19	533.63	566.40	533.60	474.00	440.20	992.06	566.40	504.20
	T7	199	200	641.40	598.00	598.00	854.40	511.60	583.00	448.60	435.80	435.80	1020.15	816.25	1002.81	573.21	597.00	897.38	602.04	641.40	595.80	478.00	437.00	985.46	641.40	577.80
Mumford *et al.* [96]	nice.25	25	100	191.62	179.91	179.73	238.65	177.66	188.52	213.60	210.40	211.68	299.66	398.25	448.50	223.48	231.74	222.38	196.68	191.48	176.79	177.74	162.18	333.15	191.62	213.40
	nice.50	50	100	194.23	182.09	182.46	274.05	184.02	191.13	210.88	205.12	205.44	302.19	355.25	459.00	206.78	217.45	227.06	194.99	194.23	176.67	181.31	164.62	329.93	194.23	246.04
	nice.100	100	100	202.85	196.27	196.26	307.23	184.86	196.21	188.24	184.08	184.56	348.36	315.38	381.25	199.02	208.19	233.02	197.67	202.85	182.93	183.65	166.40	338.25	202.85	267.06
	nice.200	200	100	210.62	203.98	203.96	350.84	186.95	201.13	185.04	181.92	182.56	450.75	320.63	315.13	198.59	210.66	241.45	202.45	210.62	187.64	184.68	167.36	334.28	210.62	276.57
	nice.500	500	100	221.63	215.07	215.07	410.81	194.07	208.22	163.20	160.80	160.80	676.77	445.16	326.41	162.76	173.61	238.24	204.47	221.63	194.40	185.10	166.87	328.91	221.63	298.24
	path.25	25	100	284.50	241.46	245.16	348.76	246.69	258.74	262.06	261.10	261.10	481.75	477.75	540.13	362.95	367.94	368.91	301.60	282.56	263.68	252.54	226.98	658.05	284.50	210.06
	path.50	50	100	380.20	311.10	320.18	476.17	293.46	317.85	279.34	276.22	276.06	577.39	535.91	588.03	372.18	382.39	440.73	343.63	379.90	341.34	304.27	275.03	772.32	380.20	225.68
	path.100	100	100	490.69	408.12	410.90	632.94	353.29	404.83	280.01	276.41	275.77	700.37	589.73	644.29	400.21	412.41	595.08	439.13	490.42	443.54	366.53	333.38	841.02	490.69	269.11
	path.200	200	100	635.97	536.04	538.66	816.51	462.00	495.59	282.98	278.20	277.27	891.24	690.79	684.87	443.23	463.81	763.96	536.11	635.97	551.23	433.11	393.88	892.73	635.97	285.82
	path.500	500	100	901.59	784.66	793.50	1223.54	580.59	662.54	259.83	258.23	256.53	1295.23	902.03	746.09	456.94	468.13	1049.64	753.13	901.59	780.30	543.81	500.83	939.03	901.59	311.62

Table D.3 (continued) : *Summary of mean packing heights in a random traversal order obtained by the online algorithms over the 542 benchmark data sets described §1.6. The shaded columns represent mean packing heights achieved by the proposed modifications and new heuristics as part of the contribution by the author.*

Author	Data Sets	# of Rec	NFL	FFL	BFL	MNFL	MFFL	MBFL	SDev	SDiff	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{0.8}$	$HS_{80.5}$	$HS_{120.5}$	SAve	SAve2	CA	CPF	CFF	CC	Azar	BiNFL	OF
Beasley [9, 10]	U_1	10	10	10	10	10	10	11	31	16	10	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	U_2	20	10	10	10	10	11	10	31	32	10	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	U_3	30	10	20	20	10	10	10	31	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	U_4	50	10	10	10	10	10	10	32	31	10	20	10	10	10	10	10	10	10	10	10	10	10	10	47
	V_1	10	10	10	10	10	10	10	31	16	10	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	V_2	17	10	10	10	10	10	10	16	31	20	10	10	10	10	10	10	10	10	11	10	10	10	10	31
	V_3	21	10	10	10	10	10	10	32	16	10	10	10	10	10	10	10	10	10	10	10	10	10	10	16
	V_4	7	10	10	10	10	10	10	16	32	11	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	V_5	14	10	10	10	10	10	10	32	16	20	10	10	10	10	10	10	10	10	10	10	10	10	10	16
	V_6	15	10	10	10	10	10	10	31	32	10	10	11	10	10	10	10	10	10	10	10	10	10	10	32
	V_7	8	10	10	10	10	10	10	31	32	10	10	10	10	10	10	10	10	10	10	10	10	10	10	16
	V_8	13	10	10	10	10	10	10	31	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	V_9	18	10	10	10	10	10	10	15	32	10	10	10	10	10	10	10	10	10	10	10	10	10	10	16
	V_{10}	13	10	10	10	10	10	10	31	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	V_{11}	15	10	10	10	10	10	10	32	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	15
	V_{12}	22	10	10	10	10	10	10	16	16	10	10	10	10	10	10	10	10	11	10	10	10	10	10	31
Burke *et al.* [18]	B_1	10	10	10	10	10	10	10	31	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	B_2	20	10	10	10	10	10	10	16	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	B_3	30	10	10	10	10	10	10	31	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	B_4	40	10	10	10	10	10	10	31	31	10	20	10	10	10	10	10	10	10	10	10	10	10	10	31
	B_5	50	10	10	10	10	10	10	32	31	10	10	10	10	10	10	10	10	10	10	10	10	10	10	31
	B_6	60	10	10	10	10	10	20	31	32	10	10	10	10	10	10	10	10	10	10	10	10	10	10	47
	B_7	70	10	10	10	10	10	10	31	47	20	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	B_8	80	10	10	20	10	20	20	32	46	10	10	20	10	10	10	10	10	10	10	10	10	10	10	31
	B_9	100	10	20	20	10	20	20	32	47	20	10	20	10	20	20	10	10	10	10	10	10	10	10	31
	B_{10}	200	10	40	40	20	30	30	47	47	20	40	20	20	10	10	20	20	10	10	10	10	20	10	78
	B_{11}	300	10	61	60	20	50	60	47	62	30	50	50	30	20	21	10	20	20	20	20	10	20	20	78
	B_{12}	500	30	90	101	20	80	80	78	78	40	71	60	51	40	30	20	20	10	20	21	20	30	20	109
Christofides [22]	G_1	16	10	10	10	10	10	10	31	16	10	10	10	10	10	10	10	10	10	10	10	10	10	10	32
	G_2	23	10	10	10	10	10	10	15	32	10	10	10	10	11	10	10	10	10	10	10	10	10	10	32
	G_3	62	11	20	20	11	10	20	31	31	20	20	20	20	10	10	10	10	10	10	10	10	10	10	46

Table D.4: *Summary of mean execution times (in milliseconds) in the forward traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rec	NFL	FFL	BFL	MNFL	MFFL	MBFL	SDev	SDiff	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{4,8}$	$HS_{8,5}$	$HS_{12,5}$	SAve	SAve2	CA	CPE	CFF	CC	Azar	BiNFL	OF
Hopper *et al.* [56, 58]	C1	16/7	10.00	10.00	10.00	10.00	3.33	10.00	26.33	26.33	6.67	6.67	10.00	10.00	10.00	10.00	10.00	0.00	6.67	10.00	10.00	0.00	10.00	6.67	31.33
	C2	25	6.67	6.67	10.00	0.00	10.00	10.00	31.00	31.67	6.67	10.00	10.00	10.00	10.00	10.00	10.00	3.33	10.00	10.00	10.00	6.67	6.67	6.67	31.67
	C3	28/9	10.00	10.00	10.00	10.00	10.00	10.00	20.33	31.33	10.00	10.00	10.00	10.00	6.67	10.00	10.00	10.00	10.00	10.00	6.67	10.00	10.00	3.33	26.67
	C4	49	6.67	10.33	10.00	13.33	10.00	10.00	31.67	31.67	10.00	13.33	13.33	10.00	6.67	6.67	10.00	6.67	6.67	10.00	10.00	10.00	10.33	10.00	46.67
	C5	72/3	10.00	13.33	16.67	10.00	13.33	13.33	31.67	31.33	13.33	13.33	13.33	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	36.67
	C6	97	13.33	13.33	13.33	10.00	20.00	13.33	31.00	31.00	13.33	13.67	16.67	10.00	13.33	13.33	10.00	10.00	10.00	10.00	10.00	13.33	13.33	10.00	41.67
	C7	196/7	13.33	30.00	33.33	20.00	30.00	30.00	47.00	46.33	23.33	26.67	26.67	13.33	16.67	16.67	10.00	10.00	10.00	16.67	16.67	10.33	10.00	13.33	62.67
	N1	17	10.00	8.00	10.00	0.00	10.00	10.00	24.80	24.60	8.00	8.00	10.00	10.00	10.00	4.00	4.00	10.20	10.00	0.00	10.00	6.00	6.20	6.00	27.80
	N2	25	10.00	10.00	8.00	8.00	10.00	10.00	28.60	27.80	10.00	12.00	10.00	8.00	8.00	10.00	10.20	4.00	10.00	8.00	6.00	10.00	10.00	6.00	31.00
	N3	29	6.00	8.00	8.00	10.00	10.00	10.00	25.20	31.00	8.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	8.00	10.00	8.00	8.00	10.00	27.80
	N4	49	10.00	10.00	12.00	10.00	10.00	10.00	28.00	31.00	12.00	12.00	10.00	10.00	10.00	10.00	8.00	8.00	8.00	10.00	8.00	10.00	10.00	8.00	31.40
	N5	73	10.00	16.00	14.00	10.00	14.20	16.00	31.40	31.40	10.00	14.00	12.00	10.00	12.00	10.00	10.00	8.00	10.00	10.00	10.00	10.00	10.00	10.20	37.60
	N6	97	10.00	12.00	16.00	12.00	16.00	14.00	41.00	37.40	16.20	18.00	16.20	12.00	14.20	10.00	10.00	10.00	10.00	12.00	10.00	10.00	12.00	12.00	43.40
	N7	197	14.20	26.00	26.00	14.20	22.00	26.00	46.60	46.60	20.00	24.00	22.00	20.00	16.00	16.00	10.00	10.00	14.00	14.00	14.00	12.00	20.00	14.00	72.20
	T1	17	10.00	10.00	10.20	10.00	6.00	6.00	22.40	25.20	10.00	8.00	8.00	6.00	10.00	10.00	10.00	10.00	10.00	6.20	10.00	8.00	10.00	6.00	25.00
	T2	25	10.00	10.00	8.00	10.00	10.00	10.00	24.80	28.40	10.00	12.00	8.00	10.00	8.00	8.00	10.00	10.00	10.00	6.00	6.00	8.00	8.00	8.00	28.00
	T3	29	8.00	10.00	8.00	6.00	10.00	8.00	27.80	28.60	10.00	10.20	10.00	8.00	10.00	10.00	6.00	10.00	4.00	10.00	10.00	10.00	10.00	8.00	28.20
	T4	49	10.00	12.00	10.00	10.00	10.00	12.00	31.00	31.40	12.00	10.00	10.00	10.00	10.00	10.20	10.00	10.00	10.00	10.00	8.00	8.00	10.00	10.00	31.40
	T5	73	10.00	14.00	16.00	10.00	16.00	16.00	40.60	34.20	14.00	14.00	14.00	10.00	10.00	10.00	10.00	10.00	8.00	10.00	10.20	10.00	14.20	10.00	37.60
	T6	97	10.00	16.00	16.00	10.00	12.00	16.00	40.80	34.00	14.00	16.00	16.00	14.00	12.00	10.00	10.00	10.20	12.00	10.00	10.00	10.00	14.00	10.00	40.60
	T7	199	14.00	24.00	30.00	14.00	24.20	28.00	44.00	46.80	20.00	26.00	24.00	18.00	14.00	16.00	12.00	16.00	10.00	14.00	16.00	16.00	16.00	10.00	59.40
Mumford *et al.* [94]	nice.25	25	9.02	9.02	9.02	8.42	9.42	9.22	27.14	27.74	9.42	10.42	10.42	8.82	9.00	8.82	7.80	8.20	7.60	9.22	9.22	9.02	8.82	8.22	31.90
	nice.50	50	9.00	11.02	11.82	9.22	10.80	12.22	30.24	30.90	10.62	11.82	11.22	9.80	9.82	9.82	8.82	9.02	9.22	10.02	10.42	9.42	9.82	9.02	36.82
	nice.100	100	10.82	16.82	17.82	11.22	16.82	20.04	35.56	35.70	13.22	16.60	14.22	12.22	11.40	11.22	9.82	10.22	10.60	11.62	11.42	10.82	11.02	10.20	52.80
	nice.200	200	14.00	30.24	31.24	14.24	28.44	30.06	45.94	46.22	18.24	29.66	23.62	19.82	17.22	16.02	11.60	12.02	13.42	15.60	14.84	14.02	14.82	13.40	89.96
	nice.500	500	24.00	91.20	93.10	23.00	79.10	82.20	78.20	78.20	32.00	75.10	53.10	55.00	38.10	33.00	20.00	20.00	20.00	24.10	22.00	23.10	22.00	20.00	248.30
	path.25	25	8.62	9.22	9.22	9.02	9.40	9.20	27.50	27.14	9.62	10.00	9.40	9.22	9.22	9.00	9.42	9.02	8.00	9.42	8.60	8.82	9.40	8.40	30.60
	path.50	50	9.80	11.02	11.42	10.00	10.62	12.82	29.58	31.28	11.42	11.00	11.22	9.60	9.20	9.62	9.42	9.22	9.02	10.02	10.00	9.82	9.80	9.20	38.24
	path.100	100	11.02	17.04	17.44	11.22	16.62	16.62	35.34	35.96	16.42	15.42	16.04	12.60	11.82	12.22	10.02	10.22	10.42	13.02	12.22	11.42	11.82	10.02	49.76
	path.200	200	14.04	33.04	33.86	14.82	30.46	33.04	46.90	45.96	25.64	27.84	26.04	18.44	16.00	15.64	12.62	13.42	13.22	17.62	15.22	14.02	15.64	12.62	84.04
	path.500	500	24.00	106.10	106.20	26.00	91.10	98.10	78.00	76.40	58.10	79.10	67.10	55.10	35.10	30.10	20.00	22.00	20.10	31.00	24.10	24.00	29.00	20.00	210.70

Table D.4 (continued) : *Summary of mean execution times (in milliseconds) in the forward traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rec	NFL_R	FFL_R	BFL_R	$MNFL_R$	MFL_R	$MBFL_R$	$SDev_R$	$SDiff_R$	$NFS_{0.5R}$	$FFS_{0.5R}$	$BFS_{0.5R}$	$HS_{0.8R}$	$HS_{80.8R}$	$HS_{i^2_{0.5}R}$	$SAve_R$	$SAve2_R$	CA_R	CPF_R	CFF_R	CC_R	$Azam_{0.25R}$	$BiNFL_R$	OF_R
Beasley [9, 10]	U_1	10	31	31	31	31	31	31	31	31	31	16	31	31	31	31	16	31	32	31	16	32	31	16	31
	U_2	20	15	31	31	31	31	31	15	31	31	31	31	31	15	15	31	15	31	15	31	31	16	32	31
	U_3	30	31	31	31	31	31	47	31	31	31	31	47	31	31	31	16	31	31	31	16	32	31	31	31
	U_4	50	31	47	47	31	47	47	31	31	31	47	47	31	31	31	32	15	31	31	31	32	31	32	31
	V_1	10	15	31	16	31	16	32	15	31	31	32	16	31	15	15	16	31	31	15	31	16	16	32	31
	V_2	17	31	31	32	31	31	31	31	15	31	31	32	31	31	31	32	16	15	31	16	32	31	16	31
	V_3	21	31	31	32	31	31	32	31	31	31	31	32	31	31	15	16	31	31	15	31	16	32	32	31
	V_4	7	15	31	16	16	16	16	31	15	15	16	16	15	31	31	31	31	15	31	15	31	32	16	31
	V_5	14	31	31	32	31	31	31	32	31	31	31	32	31	16	15	31	32	31	15	31	16	31	31	32
	V_6	15	31	31	16	47	15	16	31	16	15	31	16	31	31	31	15	16	31	31	31	31	32	31	31
	V_7	8	31	15	31	32	31	31	32	31	31	16	31	31	16	16	31	32	16	32	31	15	31	31	32
	V_8	13	31	31	16	31	31	15	16	16	15	31	16	32	32	31	15	16	32	31	31	31	31	31	32
	V_9	18	31	15	31	32	31	31	31	32	31	15	31	31	16	32	31	31	16	47	16	47	31	31	31
	V_{10}	13	32	31	31	16	31	15	16	31	31	31	31	32	32	32	16	16	32	32	32	125	31	15	32
	V_{11}	15	31	16	31	31	15	31	31	32	31	31	31	32	16	31	31	31	16	31	16	47	31	31	16
	V_{12}	22	32	31	31	32	31	31	15	31	31	47	31	31	31	32	32	31	31	32	32	32	31	15	31
Burke *et al.* [18]	B_1	10	32	31	15	31	31	31	31	31	16	31	15	32	16	31	16	15	16	32	16	31	31	31	16
	B_2	20	31	32	31	31	31	31	15	31	31	31	31	16	31	31	31	31	31	31	31	32	31	16	31
	B_3	30	32	31	31	31	31	31	31	31	31	47	31	31	15	31	16	16	31	32	32	31	31	31	31
	B_4	40	32	31	47	31	31	31	31	31	31	47	31	32	31	31	31	31	31	31	31	31	31	31	31
	B_5	50	46	31	47	31	31	31	31	31	31	47	32	32	31	31	31	31	31	31	31	32	31	32	31
	B_6	60	47	47	32	32	31	47	31	31	47	31	47	31	31	32	31	16	31	32	32	47	31	31	31
	B_7	70	47	47	47	32	47	32	31	31	46	47	47	47	32	47	31	31	31	32	32	46	31	31	47
	B_8	80	47	47	47	31	31	47	31	32	47	46	47	47	32	46	31	31	31	31	46	47	31	31	47
	B_9	100	31	63	46	31	62	47	31	32	46	63	47	47	31	31	32	31	31	31	47	47	32	31	31
	B_{10}	200	46	94	109	62	94	94	47	47	78	93	78	62	63	62	47	47	47	47	47	47	63	47	62
	B_{11}	300	78	171	172	63	156	141	47	63	110	172	125	78	63	79	63	62	62	63	63	63	78	63	62
	B_{12}	500	109	266	312	110	234	250	94	78	156	266	219	140	110	110	78	78	78	93	78	79	93	78	94
Christofides [22]	G_1	16	31	31	16	31	16	31	16	31	32	31	31	31	31	31	31	31	32	31	16	16	31	31	31
	G_2	23	31	31	32	31	31	31	31	16	32	15	31	31	31	15	31	31	16	15	31	31	16	32	31
	G_3	62	31	62	47	47	47	63	31	31	47	62	47	47	31	31	31	31	31	31	32	31	31	31	47

Table D.5: *Summary of mean execution times (in milliseconds) in the reverse traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rec	NFL_R	FFL_R	BFL_R	$MNFL_R$	$MFFL_R$	$MBFL_R$	$SDev_R$	$SDiff_R$	$NFS_{0.5R}$	$FFS_{0.5R}$	$BFS_{0.5R}$	$HS_{0.8R}$	$HS_{80.8R}$	$HS_{120.8R}$	$SAvg_R$	$SAvg2_R$	CA_R	CPF_R	CFF_R	CC_R	$Azar_R$	$BiNFL_R$	OF_R
Hopper *et al.* [50, 58]	C1	16/7	21.00	31.00	26.67	25.67	21.00	26.00	26.33	26.33	26.33	25.67	31.33	31.33	31.33	31.67	26.33	31.33	26.00	26.67	21.00	20.33	31.00	25.67	20.67
	C2	25	31.00	25.67	31.67	31.00	26.00	31.33	26.33	31.33	26.33	31.00	31.33	20.67	31.00	26.00	31.00	31.67	21.33	26.33	31.67	25.67	25.67	25.67	31.33
	C3	28/9	31.00	31.00	31.67	25.67	31.33	26.00	31.33	26.33	31.67	31.00	31.00	31.00	25.67	26.00	31.00	31.33	26.67	31.00	26.67	31.00	31.00	26.00	26.00
	C4	49	31.00	31.00	42.00	31.67	36.33	36.33	31.67	31.67	36.33	41.33	41.67	31.00	31.33	31.33	31.00	31.00	25.67	31.00	31.67	31.67	31.00	31.67	36.67
	C5	72/3	46.67	47.00	47.00	41.67	47.00	41.67	31.33	31.67	51.67	42.00	41.67	31.00	31.67	31.00	31.00	31.00	31.00	31.67	46.67	31.00	31.33	31.33	41.67
	C6	97	41.67	57.33	47.00	41.33	52.00	52.33	31.00	36.67	57.33	52.00	52.33	36.67	42.00	36.67	31.33	31.67	31.67	31.00	36.33	31.00	36.33	31.00	41.33
	C7	196/7	57.33	89.00	88.67	57.67	88.67	94.00	46.33	42.00	83.33	109.33	83.33	57.33	57.33	52.33	47.00	41.33	46.67	47.00	47.00	52.00	52.00	51.67	67.33
	N1	17	31.00	27.80	28.40	31.20	27.80	27.80	28.20	28.20	28.60	31.00	27.80	24.60	31.60	22.20	25.20	21.40	27.80	22.20	25.40	25.20	24.80	22.40	28.60
	N2	25	28.40	27.80	31.40	27.80	27.80	27.80	28.40	28.40	28.20	31.00	31.00	27.80	28.40	27.80	21.40	31.60	31.60	31.80	28.00	25.40	28.60	28.20	31.80
	N3	29	28.80	31.00	31.20	27.80	31.20	31.00	28.00	28.00	34.20	31.40	31.40	27.80	28.00	24.60	25.20	25.20	25.20	28.40	24.60	28.80	25.60	27.80	28.20
	N4	49	34.20	43.60	37.40	34.20	37.40	34.60	31.00	31.00	34.20	40.60	37.20	31.00	31.40	34.20	28.20	31.00	31.20	31.40	28.20	34.40	31.60	31.00	43.80
	N5	73	37.20	40.40	43.80	37.20	40.60	47.00	31.40	34.80	46.80	43.80	43.80	31.40	31.40	46.80	31.00	31.40	31.40	43.80	31.40	31.20	31.00	31.40	37.60
	N6	97	40.60	52.80	59.40	37.20	47.00	59.40	34.00	31.00	56.20	56.40	50.20	37.80	37.60	37.80	37.40	46.80	34.00	37.60	37.20	34.60	43.60	34.40	43.80
	N7	197	56.20	78.00	81.20	53.20	78.00	87.60	47.00	46.40	84.20	90.80	81.40	56.40	53.00	50.20	44.00	43.80	49.80	59.40	47.00	47.00	50.00	47.00	65.40
	T1	17	31.00	28.40	25.20	28.40	28.20	24.60	22.20	22.40	25.00	28.40	28.20	24.80	25.40	21.80	27.80	31.40	21.40	31.60	21.80	27.80	31.40	21.80	28.60
	T2	25	31.00	31.40	31.20	27.80	31.20	27.80	25.20	28.00	31.00	31.60	31.20	31.00	28.40	28.40	21.80	31.40	25.20	25.20	37.60	31.00	28.40	31.60	31.60
	T3	29	24.60	31.40	34.20	28.00	31.00	27.80	27.80	27.80	31.20	34.40	31.00	27.80	28.20	28.80	28.60	31.00	28.60	28.00	28.60	31.00	28.20	28.40	28.80
	T4	49	31.00	43.80	31.40	31.40	37.60	31.40	24.60	31.00	37.40	40.80	34.40	31.00	31.40	31.40	28.40	28.40	31.20	31.40	31.60	31.00	34.60	28.40	31.00
	T5	73	31.40	40.60	40.60	31.00	37.40	46.80	31.40	31.20	46.80	47.00	40.60	34.60	37.40	37.20	31.60	31.20	37.80	37.40	40.60	31.40	31.20	34.40	37.80
	T6	97	40.60	49.80	50.00	43.60	47.00	47.00	31.00	34.40	49.60	56.20	56.40	40.60	40.40	37.20	31.20	31.00	31.40	37.40	40.80	34.60	37.60	37.80	40.80
	T7	199	56.20	87.60	87.60	53.00	84.40	93.80	49.60	47.00	81.40	90.80	87.40	59.60	50.00	53.20	43.60	43.80	47.00	47.00	47.00	47.00	53.20	46.60	59.20
Munford *et al.* [94]	nice.25	25	28.44	30.26	29.96	28.72	29.14	29.84	26.62	26.58	30.04	30.98	30.00	27.44	27.74	28.88	23.74	27.82	26.52	29.08	27.58	27.48	27.26	26.88	32.86
	nice.50	50	32.52	36.22	36.88	32.16	37.58	34.70	30.32	29.98	36.50	38.78	36.88	31.24	31.60	31.90	28.02	27.62	29.76	31.30	30.88	31.54	31.88	29.64	38.44
	nice.100	100	39.70	52.84	49.98	40.60	51.88	53.38	36.22	34.72	48.16	54.64	48.76	40.64	39.02	39.12	32.22	32.20	36.24	42.12	39.76	38.44	37.76	36.92	54.12
	nice.200	200	57.20	91.02	85.94	56.84	85.02	87.48	46.60	46.84	73.48	92.50	79.34	61.00	54.32	54.04	45.36	43.98	48.74	52.46	53.40	50.92	53.18	47.50	88.78
	nice.500	500	104.80	229.90	223.30	104.70	214.10	225.20	83.00	83.00	150.00	233.10	181.40	146.90	115.70	107.70	75.00	79.80	87.60	93.80	153.20	90.50	90.60	81.20	236.20
	path.25	25	27.84	30.54	29.50	28.18	30.48	29.10	27.52	27.20	31.94	30.92	30.32	28.12	27.92	26.82	24.10	25.64	25.66	28.40	27.80	28.70	27.74	27.48	31.62
	path.50	50	32.18	34.32	33.44	32.18	35.58	37.14	29.64	30.00	39.68	37.24	37.26	31.82	31.22	30.64	29.40	30.64	30.22	37.20	30.68	32.16	31.28	31.34	38.44
	path.100	100	40.92	47.76	51.40	39.98	49.18	50.38	36.64	35.26	56.62	54.08	52.06	40.98	39.10	38.14	36.58	37.50	36.26	39.66	39.98	37.80	39.34	35.94	53.14
	path.200	200	57.16	86.68	82.84	57.16	87.12	86.00	47.16	47.78	93.44	89.58	86.24	60.96	55.94	54.44	43.74	44.98	50.02	54.38	69.68	52.18	53.42	48.14	87.14
	path.500	500	104.60	245.50	234.50	107.80	223.70	220.20	84.40	81.40	220.10	220.30	199.90	146.80	114.00	106.10	75.10	78.20	84.40	99.90	95.30	95.40	98.40	84.20	221.90

Table D.5 (continued) : *Summary of mean execution times (in milliseconds) in the reverse traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rec	NFL_{Rand}	FFL_{Rand}	BFL_{Rand}	$MNFL_{Rand}$	$MFFL_{Rand}$	$MBFL_{Rand}$	$SDev_{Rand}$	$SDiff_{Rand}$	$NFS_{0.5Rand}$	$FFS_{0.5Rand}$	$BFS_{0.5Rand}$	$HS_{0.8Rand}$	$HS_{80.5Rand}$	$HS_{20.5Rand}$	$SAve_{Rand}$	$SAve2_{Rand}$	CA_{Rand}	CPF_{Rand}	CFF_{Rand}	CC_{Rand}	$Azar_{Rand}$	$BiNFL_{Rand}$	OF_{Rand}
Beasley [9, 10]	U_1	10	31	31	16	15	31	47	31	16.00	78.00	31.00	31.00	31.00	31.00	16.00	31	31	20	30	20	20	21	20	31
	U_2	20	31	31	32	31	31	31	32	32.00	46.00	31.00	15.00	31.00	15.00	32.00	32	31	20	30	20	20	20	20	31
	U_3	30	31	47	31	31	31	32	31	31.00	63.00	31.00	31.00	31.00	31.00	32.00	32	31	20	30	30	20	30	20	31
	U_4	50	31	47	63	31	62	47	31	32.00	47.00	47.00	47.00	31.00	31.00	31.00	31	31	20	30	20	30	20	20	31
	V_1	10	31	31	31	15	32	31	32	32	46	32	32	31	31	32	32	15	20	30	30	20	20	20	31
	V_2	17	31	31	32	31	31	15	47	16	47	31	31	31	31	32	16	31	20	30	20	20	30	21	15
	V_3	21	31	31	32	31	31	31	31	32	63	32	31	15	15	16	31	15	20	40	20	20	40	20	31
	V_4	7	31	16	31	31	32	31	32	16	47	16	16	31	31	32	16	31	20	30	20	20	30	20	15
	V_5	14	31	16	31	31	47	31	31	16	47	16	15	31	31	31	15	31	20	40	30	20	30	20	31
	V_6	15	31	32	31	31	32	31	16	31	78	31	31	15	31	16	31	16	20	30	20	20	30	20	31
	V_7	8	31	31	16	31	31	31	16	15	47	31	15	31	31	31	31	32	20	30	20	20	30	20	16
	V_8	13	31	32	31	31	16	31	31	31	47	31	31	16	31	15	16	31	20	40	30	20	40	20	32
	V_9	18	32	32	31	31	32	31	16	31	46	31	31	31	32	31	31	32	20	30	20	30	30	20	31
	V_{10}	13	31	16	31	15	31	31	31	31	63	15	15	16	31	15	16	16	20	41	20	20	30	10	32
	V_{11}	15	32	32	31	31	16	31	15	15	31	31	31	32	16	31	32	31	20	30	20	30	30	20	32
	V_{12}	22	32	32	31	16	32	31	31	31	32	31	31	16	32	15	16	16	20	30	20	20	30	20	31
Burke *et al.* [18]	B_1	10	20	30	30	10	30	30	15	32	10	10	10	10	20	20	10	10	32	31	31	32	31	16	63
	B_2	20	10	30	40	10	30	30	31	32	10	10	10	10	20	10	10	20	32	31	31	31	31	32	62
	B_3	30	20	30	31	20	30	40	15	31	10	10	10	10	20	10	20	10	31	31	31	31	31	32	62
	B_4	40	10	10	10	10	10	10	31	32	10	10	10	10	10	10	10	10	32	31	31	32	31	16	31
	B_5	50	10	10	10	10	10	10	31	32	10	10	10	10	10	10	10	10	16	31	31	31	31	32	16
	B_6	60	10	10	10	10	10	10	31	46	10	10	10	10	10	10	10	10	31	31	31	32	31	16	31
	B_7	70	10	10	10	10	10	10	31	32	10	20	20	10	10	10	10	10	32	31	31	32	31	31	16
	B_8	80	10	10	10	10	10	10	31	32	10	10	10	10	10	10	10	10	16	15	16	31	16	32	32
	B_9	100	10	10	10	10	10	10	31	31	20	20	20	10	10	10	10	10	31	31	31	32	31	16	31
	B_{10}	200	10	10	10	10	10	10	46	47	20	30	31	20	10	10	10	10	16	15	16	32	31	31	16
	B_{11}	300	10	10	10	10	10	10	63	62	30	40	40	21	10	10	10	10	31	31	32	31	32	16	32
	B_{12}	500	10	10	10	10	10	10	78	78	41	80	60	60	10	10	10	10	15	16	16	16	31	31	31
Christofides [22]	G_1	16	31	31	31	31	31	31	31	47	32	15	16	15	31	16	31	16	20	30	20	20	20	20	16
	G_2	23	15	32	31	31	32	31	32	32	31	31	31	31	15	32	16	31	20	20	20	20	30	30	31
	G_3	62	31	62	62	31	47	46	31	31	47	62	47	31	31	32	32	31	20	20	20	20	30	20	31

Table D.6: *Summary of mean execution times (in milliseconds) in a random traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Author	Data Sets	# of Rec	NFL_{Rand}	FFL_{Rand}	BFL_{Rand}	$MiNFL_{Rand}$	$MFFL_{Rand}$	$MBFL_{Rand}$	$SDev_{Rand}$	$SDiff_{Rand}$	$NFS_{0.5Rand}$	$FFS_{0.5Rand}$	$BFS_{0.5Rand}$	$HS_{6,0.5Rand}$	$HS_{8,0.5Rand}$	$HS_{12,0.5Rand}$	$SAve_{Rand}$	$SAve2_{Rand}$	CA_{Rand}	CPF_{Rand}	CFF_{Rand}	CC_{Rand}	$AzuT_{Rand}$	$BiNFL_{Rand}$	OF_{Rand}
Hopper *et al.* [56, 58]	C1	16/7	10.00	13.33	13.33	10.00	13.33	10.33	26.00	25.67	6.67	10.00	3.33	6.67	10.00	10.00	10.33	10.00	0.00	10.00	10.00	10.00	10.00	3.33	36.33
	C2	25	10.00	13.33	13.33	10.00	13.33	16.67	21.33	26.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	0.00	10.00	6.67	10.33	10.00	10.00	41.33
	C3	28/9	10.00	16.67	13.33	10.00	20.00	13.33	31.67	31.33	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	3.33	6.67	10.00	10.00	10.00	36.33
	C4	49	10.00	10.00	10.00	10.33	10.00	10.00	26.33	31.00	10.00	13.67	13.33	10.00	10.00	6.67	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	26.00
	C5	72/3	10.00	13.33	13.33	10.00	10.00	10.00	31.33	37.00	10.00	16.67	16.67	10.00	10.00	10.00	10.00	10.33	10.00	10.00	10.00	10.00	13.33	10.33	36.67
	C6	97	10.33	13.33	16.67	10.00	20.00	16.67	31.67	41.33	16.67	13.33	13.33	10.00	13.33	13.33	10.00	10.00	10.00	10.00	10.00	10.00	13.33	10.00	36.67
	C7	196/7	16.67	67.00	70.00	20.00	57.00	63.33	41.67	47.00	23.33	26.67	23.33	20.00	23.33	26.67	16.67	16.67	10.00	13.33	10.00	16.67	20.00	10.00	78.00
	N1	17	10.00	6.00	6.00	10.00	10.00	10.00	24.80	25.40	8.00	6.00	8.00	6.00	10.00	4.00	10.00	10.00	28.00	25.40	31.00	24.60	31.40	24.60	28.60
	N2	25	10.00	10.00	10.00	10.00	8.00	10.00	27.80	28.20	10.00	10.00	10.00	10.00	6.00	10.00	10.00	10.00	18.60	18.40	15.20	18.60	21.60	11.40	31.20
	N3	29	8.00	8.00	8.00	6.00	10.00	10.00	25.20	25.00	10.00	10.00	10.00	8.00	10.00	8.00	10.00	4.00	10.00	10.20	8.00	10.00	10.00	10.00	28.00
	N4	49	10.00	10.00	10.00	10.00	12.00	12.00	31.40	28.20	10.00	10.00	12.00	10.00	10.20	10.00	8.00	10.00	14.00	14.00	12.00	12.00	18.20	14.00	31.00
	N5	73	10.00	16.00	14.00	10.00	10.00	16.00	31.20	37.80	14.00	14.00	14.20	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	4.00	6.00	10.00	47.00
	N6	97	12.20	16.00	16.00	10.20	18.20	16.00	31.40	34.20	14.20	16.20	16.00	14.20	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	2.00	37.60
	N7	197	16.00	30.00	22.00	14.00	20.00	30.00	46.80	46.80	20.00	26.00	24.00	20.00	20.00	16.00	14.00	16.00	10.20	8.00	6.00	10.00	8.00	10.00	65.20
	T1	17	10.00	10.00	10.20	10.00	10.00	10.00	28.20	31.00	24.00	10.00	8.00	6.00	10.00	10.00	10.00	10.00	10.00	10.00	10.20	10.00	10.00	8.00	28.20
	T2	25	10.00	8.00	8.00	8.00	8.00	10.00	31.60	27.80	20.00	10.00	10.00	8.00	10.00	8.00	10.20	10.00	10.00	10.00	10.00	10.00	12.00	10.00	25.40
	T3	29	8.00	10.00	10.00	8.00	10.00	10.00	24.80	31.00	8.00	10.00	10.00	10.00	8.00	10.00	10.00	10.00	10.00	10.20	14.00	10.00	10.20	10.00	31.20
	T4	49	10.00	10.00	10.00	10.00	12.00	14.00	27.80	31.00	10.00	10.00	14.00	10.00	10.00	10.00	8.00	6.00	8.00	14.00	14.00	12.00	16.00	8.00	31.20
	T5	73	12.00	14.00	14.00	10.00	10.00	14.00	31.40	31.40	16.00	10.00	14.00	12.00	10.00	10.20	10.00	10.00	2.00	10.00	10.00	10.00	10.00	4.00	34.60
	T6	97	10.00	16.20	16.00	12.00	12.00	18.00	31.00	37.40	14.20	18.00	18.20	12.00	12.00	12.00	8.00	10.00	10.00	8.00	6.00	10.00	8.00	10.20	37.40
	T7	199	14.00	26.00	24.00	14.00	26.20	30.20	46.80	47.00	24.00	26.20	24.00	18.00	14.00	14.00	16.00	10.00	10.00	10.00	10.00	8.00	10.00	10.00	66.00
Mumford *et al.* [94]	nice.25	25	8.62	9.42	9.42	8.62	9.20	9.20	26.52	28.66	9.82	9.40	9.40	8.62	8.82	8.82	9.42	8.40	8.82	9.62	10.02	9.62	10.82	9.60	31.60
	nice.50	50	9.22	10.62	11.62	9.22	11.82	12.42	29.94	29.96	11.02	11.82	10.62	10.02	10.02	9.82	9.20	8.82	9.00	9.40	9.82	9.62	9.20	9.42	37.84
	nice.100	100	10.62	16.00	16.22	10.82	15.82	16.84	34.66	36.80	13.22	16.62	14.62	12.02	11.82	11.02	9.62	10.42	10.02	10.62	11.02	10.82	10.42	10.02	52.82
	nice.200	200	14.64	29.24	29.44	14.24	28.04	30.44	46.28	46.50	19.02	27.44	22.62	19.62	16.22	15.82	12.00	14.02	12.82	15.04	13.20	13.02	13.42	10.62	87.84
	nice.500	500	22.00	81.20	85.10	23.00	74.10	82.20	79.80	81.10	36.00	70.10	54.10	53.10	38.10	35.00	19.00	21.10	15.00	15.00	15.00	14.00	20.00	14.00	225.00
	path.25	25	8.02	8.80	9.62	8.22	8.60	9.22	26.82	28.10	9.60	9.82	9.82	9.02	8.82	9.00	7.40	8.22	8.82	9.40	9.20	8.62	8.82	8.62	31.20
	path.50	50	9.80	10.62	11.22	9.20	10.62	12.02	29.36	30.60	11.82	11.02	10.42	9.80	9.40	9.82	9.00	9.00	9.00	10.22	10.00	9.80	10.22	10.02	36.58
	path.100	100	11.22	15.42	17.24	11.42	15.82	16.02	34.70	35.92	15.84	16.24	14.64	15.82	11.82	11.62	9.82	10.02	10.42	12.62	11.80	11.20	11.82	10.82	48.42
	path.200	200	13.82	28.84	30.84	14.00	27.64	30.04	47.20	46.22	25.44	26.44	25.24	19.02	16.62	15.24	12.00	13.62	11.42	13.60	15.02	14.42	14.02	13.82	72.44
	path.500	500	24.00	92.10	96.10	25.10	85.10	92.10	79.60	78.00	56.00	73.10	63.10	53.10	36.10	30.10	20.10	22.00	15.10	12.00	15.10	14.00	20.00	15.00	162.60

Table D.6 (continued) : *Summary of mean execution times (in milliseconds) in a random traversal order expended by the online algorithms over the 542 benchmark data sets described in §1.6. The shaded columns represent execution times expended by the proposed modifications and new heuristics as part of the contribution by the author. A function with resolution of 10ms was used to measure execution time.*

Appendix E

Information on the compact disc accompanying this dissertation

In this appendix information on the compact disc (CD) accompanying this dissertation is provided. There are four directories on the CD, namely *Dissertation, BenchmarkData, SPDSS* and *Results* and their contents are described below.

Dissertation: this directory contains an electronic copy of the dissertation along with the LaTeX source files. All the tables and figures are also included.

BenchmarkData: several benchmark instances from the literature were employed in this dissertation (see §1.6) and they have been arranged according to the names of the authors who generated the data sets. The data sets are excel files with three columns representing the rectangle reference, height and width.

SPDSS: the decision support system was implemented using Visual Basic 6.0 (VB) which has a tool, known as *package and deployment wizard*, that assembles all files needed to run a VB program on computers that do not have VB installed. This tool is important because it automatically builds a setup program for any application that handles the setup process [49]. Once installed, the user may begin to explore SPDSS and all that it has to offer. Included in the directory is a text file labelled *READ ME* that maps out exactly how the setup program should be installed. All the 542 benchmark instances have already been loaded in SPDSS and on the main user interface under the help menu, a complete list of the strip widths corresponding to each instance have been provided.

Results: this directory contains the results obtained when all the algorithms were tested on the 542 benchmark instances. As shown in Figure E.1, the results are partitioned into two categories of execution time and total packing height. Under each category, the different packing types of either online or offline may be explored further. The online packing problems were solved using level heuristics, shelf heuristics, special case heuristics or plane heuristics, while offline packing problems were solved using exact algorithms, level heuristics or plane heuristics.

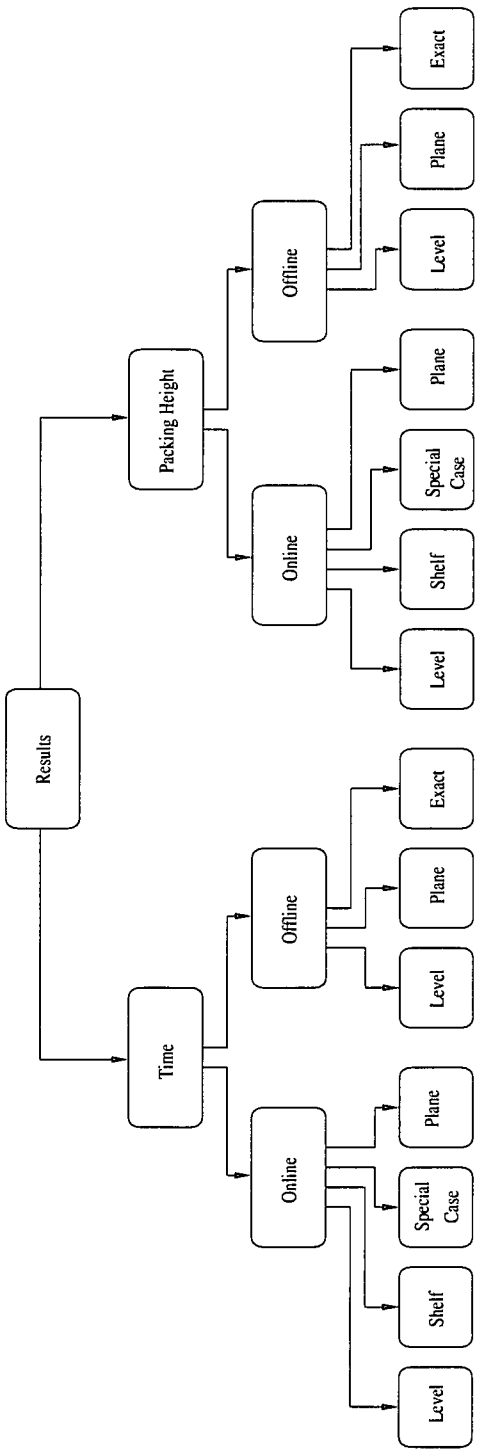

Figure E.1: *Organisation of the results directory.*

The Microsoft Windows XP (home edition 2002) operating system was used to compile all the source code. The Microsoft package Access is used to store data in SPDSS and Microsoft package Excel is also used to store the data, but most importantly to run the statistical tests (*e.g.* ANOVA, chi-squared test, students' t-test) on the results.

Appendix F

Summary of the sub-typology for packing problems

Field	Characteristics
Dimensionality	1D – one dimensional 2D – two dimensional 3D – three dimensional HoD – higher order dimensions
Shapes	R – regular shapes I – irregular shapes
Packing Region	SB – single bin packing MFB – multiple fixed sized bins packing MVB – multiple variable sized bins packing SP – strip packing
Packing type	Off – offline Aon – almost online On – online
Objectives	Mai – maximise items MiA – minimise area MiB – minimise numebr of bins MiC – minimise costs MiS – minimise strip height
Constraints	orientation placement modification guillotine

Table F.1: *Summary of the six fields in the sub-typology developed for packing problems.*