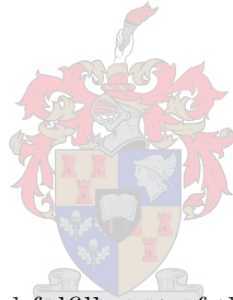


**Formal Concept
Analysis applied to
Pattern Matching and
Automata**
PhD Thesis



submitted towards partial fulfillment of the degree of PhD
at the University of Stellenbosch.

by

Frederick Johannes Venter

March 2021

First Promotor: Prof. Dr. Dr. B.W. Watson
(University of Stellenbosch)
Second Promotor: Prof. Dr. D.G. Kourie
(University of Stellenbosch)

*Dedicated to:
my wife: Issie
and children: Anri, Andries and Theodore*

Acknowledgments

I cannot thank my promotors Derrick Kourie and Bruce Watson enough for their gracious support, guidance and patience. Thanks also to Loek Cleophas for your technical inputs. I thank all the researchers at the FASTAR and ESPRESSO research groups for their support and companionship. A special thank you goes to Deon Oosthuizen who has inspired me to pursue research into Formal Concept Analysis. I thank my family for indulging, supporting and encouraging me over the many years that this research took me to complete.

Fritz Venter
March 2021

Copyright © 2021 Stellenbosch University
All rights reserved

Abstract

This thesis explores the use of formal concept analysis (FCA) to solve pattern matching problems conventionally solved by techniques based on finite automata (FAs). The problems examined in some detail are 2D pattern matching of rectilinear objects, pattern matching on multiple keywords and construction of failure FAs. In addition, broad FCA based approaches to solving problems are proposed that address non-deterministic FA to deterministic FA reduction and that address acyclic deterministic FA pattern matching. Overall, the thesis illustrates that many of these pattern matching problems are amenable to solutions based on FCA. However, the formal concept lattice built to solve any of these problems will invariably encapsulate more information than what is needed to solve the particular problem at hand. While this might be space / time inefficient, it might also represent an opportunity to be exploited for associated problems. Neither of these matters are empirically explored in the thesis.

Abstrak

In hierdie proefskrif word die gebruik van formele konsepanalise (FKA) ondersoek om patroonpassingsprobleme op te los wat gewoonlik opgelos word deur tegnieke gebaseer op eindige outomate (EO's). Die probleme wat in detail bespreek is, is 2D-patroonpassing van reglynige objekte, patroonpassing op veelvoudige sleutelwoorde en konstruksie van faalings-EO's. Daarbenewens word bre FKA-gebaseerde benaderings vir die oplos van probleme voorgestel wat die reduksie van nie-deterministiese EO's tot deterministiese EO's aanspreek en wat asikliese deterministiese EO-patroonaanpassing aanspreek. Oor die algemeen illustreer die proefskrif dat baie van hierdie patroonpassingsprobleme geskik is vir oplossings gebaseer op FKA. Die formele konseprooster wat gebou is om enige van hierdie probleme op te los, sal egter meer inligting bevat as wat nodig is om die betrokke probleem op te los. Alhoewel dit in terme van tyd en ruimte ondoeltreffend mag wees, mag dit ook 'n geleentheid bied wat vir verwante probleme ontgun kan word. Hierdie sake word egter nie empiries in die proefskrif ondersoek nie.

Contents

| | |
|--|-----------|
| List of Figures | xi |
| List of Tables | xv |
| I Prologue | 1 |
| 1 Introduction | 5 |
| 1.1 Background | 5 |
| 1.2 Related Fields of Research | 6 |
| 1.2.1 Pattern Matching | 6 |
| 1.2.2 Formal Concept Analysis | 7 |
| 1.3 Problem Statements and Research Question | 12 |
| 1.4 Research Method | 12 |
| 1.5 Thesis Outline | 12 |
| 1.5.1 Preliminaries | 13 |
| 1.5.2 2D Pattern Matching | 13 |
| 1.5.3 Multiple Keyword Pattern Matching | 14 |
| 1.5.4 Failure DFAs | 15 |
| 1.5.5 NFA to DFA reduction and ADFA pattern matching using FCA | 17 |
| 1.5.6 Conclusion | 17 |
| 2 Preliminaries | 19 |
| 2.1 Introduction | 19 |
| 2.2 General | 19 |
| 2.3 Lattice Theory and Formal Concept Analysis | 21 |

| | | |
|-----------|--|-----------|
| 2.4 | Languages and Automata | 24 |
| | | |
| II | Formal Concept Analysis applied to pattern matching | 33 |
| | | |
| 3 | FCA 2D Pattern Matching | 35 |
| 3.1 | Preamble | 35 |
| 3.2 | Introduction | 35 |
| 3.3 | Background on the application domain | 36 |
| 3.4 | Preliminaries related to layouts and design rules | 37 |
| 3.4.1 | Layouts and Design Rules In Terms of <i>V-Sets</i> | 37 |
| 3.4.2 | From <i>V-Sets</i> to <i>V-Images</i> | 40 |
| 3.4.3 | Functions and operators related to <i>v-images</i> | 42 |
| 3.4.4 | Functions related to matching on <i>v-images</i> | 46 |
| 3.5 | Concept lattices of <i>v-sets</i> for EDA pattern matching | 46 |
| 3.5.1 | High level algorithm to generate a concept lattice from a set of design rules | 46 |
| 3.5.2 | Matching design rules on a layout | 49 |
| 3.6 | Experimental Tools and Initial Results | 55 |
| 3.6.1 | Experimental Software Tools | 55 |
| 3.6.2 | Initial Results | 57 |
| 3.6.2.1 | Experimental Data | 57 |
| 3.6.2.2 | Illustration of the matching process | 59 |
| 3.7 | Conclusion | 64 |
| | | |
| 4 | PEPL Pattern Matching | 67 |
| 4.1 | Overview | 68 |
| 4.2 | Position Encoded Pattern Lattices (PEPLs) | 69 |
| 4.3 | PEPL-based Matching Using <i>PMatch</i> | 73 |
| 4.3.1 | Performance of <i>PMatch</i> | 77 |
| 4.4 | APEPL Automata | 77 |
| 4.5 | The transition function of an <i>APEPL-Automaton</i> | 82 |
| 4.6 | The transition-output mapping of an APEPL Automaton | 88 |
| 4.7 | The state-output mapping of an APEPL Automaton | 94 |
| 4.8 | Matching Using an <i>APEPL-Automaton</i> | 97 |
| 4.9 | Refactored <i>PEPL-Automaton algorithm</i> | 101 |
| 4.10 | Conclusion | 102 |

| | | |
|------------|--|------------|
| III | Formal Concept Analysis applied to automata | 105 |
| 5 | FCA based FDFA Construction | 107 |
| 5.1 | Introduction | 107 |
| 5.2 | Failure Deterministic Finite Automata | 110 |
| 5.3 | State / Out-Transition Formal Concept Lattices | 118 |
| 5.4 | A DFA-Homomorphic Algorithm | 121 |
| 5.4.1 | Recomputing arc redundancy | 125 |
| 5.4.2 | The role of cycles | 130 |
| 5.4.3 | Complexity | 131 |
| 5.5 | A Lattice-Homomorphic Algorithm | 131 |
| 5.5.1 | Lattice-Homomorphic Algorithm Preliminaries | 132 |
| 5.5.2 | Mapping DFA states to State/Out-transition Lattice objects | 133 |
| 5.5.3 | Reducing the DFA based on own objects of the State/Out-transition Lattice of the input DFA | 134 |
| 5.5.4 | Embedding the reduced DFA into the State/ Out-transition Lattice of the initial input DFA | 136 |
| 5.5.5 | Initializing and Deriving the FDFA | 139 |
| 5.5.6 | Lattice-Homomorphic Algorithm: Putting it all together | 142 |
| 5.6 | The Next Steps | 145 |
| 6 | FCA in FA reduction and ADFA matching | 149 |
| 6.1 | Introduction | 149 |
| 6.2 | FCA based NFA to DFA transformation Algorithm | 150 |
| 6.2.1 | Introduction | 150 |
| 6.2.2 | Algorithm Skeleton | 150 |
| 6.2.2.1 | Problem Definition | 150 |
| 6.2.3 | The algorithm overview | 150 |
| 6.2.4 | From NFA to NFA Concept Lattice | 151 |
| 6.2.5 | From NFA Concept Lattice to DFA | 153 |
| 6.2.6 | Conclusion | 157 |
| 6.3 | ADFAs for pattern matching using FCA | 158 |
| 6.3.1 | Overview | 158 |
| 6.3.2 | Prefix lattices | 158 |
| 6.3.2.1 | Introduction to tries | 158 |
| 6.3.2.2 | A lattice of prefixes | 159 |

| | | |
|----------|---|------------|
| 6.3.3 | PEPL based ADFAs | 163 |
| 6.3.3.1 | Introduction | 163 |
| 6.3.3.2 | PEPL based ADFAs construction | 164 |
| 6.3.3.3 | Comparing the size of PEPL based ADFAs to prefix lattice based ADFAs (tries) | 171 |
| 6.3.3.4 | Pattern matching using PEPL based ADFAs . . | 174 |
| 6.3.4 | RPEPL based ADFAs | 175 |
| 6.3.4.1 | Introducing the position misalignment problem | 175 |
| 6.3.4.2 | A proposed solution to the position misalign- ment problem | 176 |
| 6.3.4.3 | Limitation of the RPEPL approach and Future Work | 180 |
| 7 | Conclusion | 183 |
| | References | 185 |

List of Figures

| | | |
|-------|--|----|
| 1.2.1 | The Ipanema Context | 9 |
| 1.2.2 | Line diagram of Ipanema Concept Lattice | 9 |
| 1.5.1 | 2D Pattern Matching Editor | 13 |
| 1.5.2 | 2D Pattern Matching Lattice | 13 |
| 1.5.3 | Position encoded context for $P = \{abc, aabc, abcc\}$ | 14 |
| 1.5.4 | Line diagram of PEPL for $P = \{abc, aabc, abcc\}$ | 14 |
| 1.5.5 | Initial DFA and an equivalent FDFA. | 15 |
| 1.5.6 | The state/out-transition context of DFA in Figure 1.5.5 | 16 |
| 1.5.7 | State/out-transition formal concept lattice of DFA in Figure 1.5.5 | 16 |
| 3.4.1 | Example v-set for two polygons | 40 |
| 3.4.2 | “Example of an image overlaid on a v-set.” | 43 |
| 3.5.1 | An attribute depicted by the shaded area of a v-image | 50 |
| 3.5.2 | Example of a context created from a set of v-images | 51 |
| 3.6.1 | Main user interface of the Design Rule Editor. | 56 |
| 3.6.2 | Main user interface of the Lattice Context Editor. | 57 |
| 3.6.3 | v-sets of chip design rules. | 58 |
| 3.6.4 | Rotations and Inversions of a v-image | 59 |
| 3.6.5 | Lattice Diagram of Design Rules | 60 |
| 3.6.6 | First matching concept | 62 |
| 3.6.7 | Matching applied to the descendants of the initial matching concept | 63 |
| 4.2.1 | Position encoded context for $P = \{abc, aabc, abcc\}$ | 71 |
| 4.2.2 | Cover graph of PEPL for $P = \{abc, aabc, abcc\}$ | 71 |
| 4.4.1 | PEPL derived from $P^\#$ | 81 |

| | | |
|-------|--|-----|
| 4.5.1 | Transition $\delta(\diamond 1, \langle 2, a \rangle) = \diamond 3$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 85 |
| 4.5.2 | Transition $\delta(\diamond 3, \langle 5, b \rangle) = \diamond 1$ shown for PEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 87 |
| 4.5.3 | Transition $\delta(3, \langle 3, a \rangle) = \diamond 3$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 89 |
| 4.6.1 | Transition <i>output</i> $v(\diamond 1, \langle 2, a \rangle) = 0$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 91 |
| 4.6.2 | Transition <i>output</i> $v(\diamond 3, \langle 5, b \rangle) = 5$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 93 |
| 4.6.3 | Transition <i>output</i> $v(3, \langle 3, a \rangle) = 1$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 95 |
| 4.7.1 | State <i>output</i> $\rho(\diamond 3, 4) = \{aabc, (a)abc\}$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$ | 96 |
| 4.7.2 | PEPL Automaton superimposed on a PEPL Cover Graph | 98 |
| 4.9.1 | Visual depiction of $Inv(c, i, t) \triangleq Inv1(i, t) \wedge Inv2(c, i, t)$ | 101 |
| 5.2.1 | Initial DFA: $ \delta = 16, \mathbf{f} = 0$ | 114 |
| 5.2.2 | F DFA equivalent to the DFA in Figure 5.2.1: $ \delta = 8, \mathbf{f} = 3$ | 115 |
| 5.2.3 | F DFA with which worst case bound of $\mathcal{O}(x \times (Q - 1))$ on processing a string x is achievable. | 116 |
| 5.3.1 | The DFA's state/out-transition formal concept lattice | 120 |
| 5.4.1 | F DFA after one iteration. $ \delta = 10, \mathbf{f} = 2$ | 124 |
| 5.4.2 | F DFA after two iterations $ \delta = 8, \mathbf{f} = 3$ | 125 |
| 5.4.3 | Alternative F DFA after two iterations $ \delta = 8, \mathbf{f} = 3$ | 125 |
| 5.4.4 | Part of a DFA | 127 |
| 5.4.5 | Two concepts in the state/out-transition concept lattice derived from Figure 5.4.4 | 127 |
| 5.4.6 | Transformation to F DFA after first step | 128 |
| 5.4.7 | Using failure node as a failure arc target results in optimal solution. | 128 |
| 5.5.1 | Input DFA | 134 |
| 5.5.2 | DFA State/Out-transition Lattice | 134 |
| 5.5.3 | Input DFA states have a 1-to-1 relationship with corresponding State/Out-transition Lattice objects | 135 |
| 5.5.4 | Reduced DFA formed by merging states in the same set of own objects of the input DFA's State/Out-transition Lattice | 137 |

| | | |
|--------|--|-----|
| 5.5.5 | DFA derived using function <i>soer</i> | 138 |
| 5.5.6 | Transforming FDFA: $F DFAadd(\mathcal{F}, \diamond 2, \diamond 1)$ | 141 |
| 5.5.7 | Transforming FDFA: $F DFAadd(\mathcal{F}, c, d)$ applied to all $\langle c, d \rangle \in$ $\{\langle \diamond 2, \diamond 1 \rangle, \langle \diamond 6, \diamond 1 \rangle, \langle \diamond 5, \diamond 1 \rangle, \langle \diamond 7, \diamond 1 \rangle\}$ | 142 |
| 5.5.8 | Transforming FDFA: merging states created by $F DFAadd(\mathcal{F}, c, d)$ as shown in Figure 5.5.7 | 143 |
| 5.5.9 | Final result of applying the Lattice Homomorphic Algorithm to the DFA in Figure 5.5.1 | 145 |
| 5.5.10 | Final result of applying the Lattice Homomorphic Algorithm to the DFA in Figure 5.5.1 without the underlying State/Out Transition Lattice | 146 |
| 6.2.1 | Example NFA | 151 |
| 6.2.2 | Example DFA from NFA using the <i>subset construction algorithm</i> | 152 |
| 6.2.3 | Line Diagram of the NFA Concept Lattice for the NFA in Figure 6.2.1 | 154 |
| 6.2.4 | DFA from NFA Concept Lattice | 157 |
| 6.3.1 | The <i>trie</i> of a set of keywords | 159 |
| 6.3.2 | The line (Hasse) diagram of the concept lattice derived from the prefixes of the keywords $P = \{stringology, studies, strings\}$ compared to the trie of P | 162 |
| 6.3.3 | The line diagram of the concept lattice for the formal context shown in Figure 6.3 | 166 |
| 6.3.4 | ADFA D overlaid onto the line diagram of the concept lattice shown in Figure 6.3.3 | 167 |
| 6.3.5 | ADFA D with underlying lattice removed for clarity | 168 |
| 6.3.6 | Two paths for the same keyword <i>last</i> in the ADFA shown in Figure 6.3.5 | 169 |
| 6.3.7 | ADFA generated by Algorithm 19 on the PEPL for the key- words $P = \{last, lamb, lost\}$ | 172 |
| 6.3.8 | The ADFA generated from the prefix lattice of keywords $P =$ $\{last, lamb, lost\}$ | 173 |
| 6.3.9 | The line diagram of P' | 177 |
| 6.3.10 | The ADFA derived from the PEPL for P' using Algorithm 19 | 178 |
| 6.3.11 | The line (Hasse) diagram of the RPEPL for the keywords $P = \{blast, lamb, lost\}$ | 179 |

6.3.12 The line diagram of the ADFA generated by Algorithm 19 on the RPEPL for the keywords $P = \{blast, lamb, lost\}$ 180

List of Tables

| | | |
|-----|--|-----|
| 3.1 | V-image returned by Function <i>astoi</i> | 52 |
| 3.2 | Target v-image to match | 61 |
| 4.1 | Details of concepts of the PEPL in Figure 4.2.2 | 75 |
| 4.2 | Algorithm 3 trace: matching $\{abc, aabc, abcc\}$ in <i>aaabcdabccd</i> | 76 |
| 4.3 | Context derived by augmenting $P = \{abc, aabc, abcc\}$ | 80 |
| 4.4 | Execution trace showing $ aaabcdabccd = 10$ positions visited when matching <i>abc, aabc, abcc</i> on <i>aaabcdabccd</i> | 101 |
| 5.1 | The DFA's state/out-transition context | 119 |
| 6.1 | NFA Formal Context | 153 |
| 6.2 | Formal context for the prefixes of the keywords $P = \{stringology,$ <i>studies, strings\}</i> | 161 |
| 6.3 | Position encoded formal context for the keywords $P = \{last,$ <i>lamb, lost\}.</i> | 165 |
| 6.4 | Formal context for prefixes of the keywords $P = \{last, lamb,$ <i>lost\}.</i> | 173 |
| 6.5 | Position encoded formal context for the keywords $P' = \{blast,$ <i>lamb, lost\}.</i> | 176 |
| 6.6 | Reverse position encoded formal context for the keywords $P =$ $\{blast, lamb, lost\}$ | 179 |

Part I

Prologue

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D) in the Department of Socio-Informatics, University of Stellenbosch, South Africa. My interest in Formal Concept Analysis (FCA) started in 1993 under the guidance and inspiration of Deon Oosthuizen. I have since moved into the commercial world of software engineering, but kept an active interest in research into FCA. I was later introduced to pattern matching by Bruce Watson and Derrick Kourie. Their inspiration led to me joining the FASTAR and ESPRESSO research groups and to the start of the research reported in this thesis. Along the way, I also published in various conferences and publications related to the fields of both FCA and pattern matching. An examples of an early papers that I published or presented on the subject of FCA appeared in [OV93] and [FV97]. The latest one was in Nature Scientific Reports [VBV⁺18]. A significant part of this thesis is based on adapted versions of papers I wrote for conference proceedings or publications.

Chapter 1

Introduction

1.1 Background

An important interplay between finite automata (FAs) and efficient pattern matching algorithms has existed since the inception of the fields decades ago. An FA is a formalism based on a data structure called a directed graph, that is used to generically represent a matching computational machine. Many of the best algorithms in pattern matching pre-process a set of patterns to create finite automata for matching. Another class of algorithm has members that transform finite automata from one type of automaton to another — for example: algorithms that transform non-deterministic finite automata (NFAs) into a language equivalent deterministic finite automaton (DFAs); or algorithms that create so-called failure automata; or that minimize a given DFA.

The field of pattern matching has often involved algorithm construction, supported by FAs and closely related formalisms. An important question this thesis is trying to answer is whether a fundamental departure from this approach is possible. Specifically, would the application of formalisms developed in other fields of research provide alternative ways to construct algorithms for pattern matching and assist in constructing algorithms that transform finite automata in ways mentioned above?

To answer the above question generally would clearly be an overly ambitious endeavour. Instead, this thesis initiates the journey towards answering the general question. In fact this thesis attempts to answer a sub-question con-

strained to the application of just one formalism called a *concept lattice* that is not traditionally associated with pattern matching. The field of Formal Concept Analysis (FCA) is a principled way to construct concept lattices. Thus, the thesis proposes algorithms that use FCA to solve a selection of classical problems in the field of pattern matching and automaton transformation.

The sections in this chapter

- introduce the areas of pattern matching and FCA;
- elaborate on the research question posed above;
- provide an outline of the research method;
- give a high level introduction of the thesis chapters;

1.2 Related Fields of Research

1.2.1 Pattern Matching

Pattern matching is a vast area of research dating from the early days of Computer Science. For example, the Knuth-Morris-Pratt and Boyer-Moore algorithms for (one-dimensional) string pattern matching appeared in 1977. Since then, researchers have devised many more algorithms that solve the problem of single string pattern matching, multiple string pattern matching, extended string matching, regular expression parsing, approximate pattern matching and so forth.

The field of two dimensional pattern matching also started in 1977. The first linear-time (on fixed alphabets) two-dimensional exact pattern matching algorithm was found independently by Bird [Bir77] and Baker [Bak78]. The first attempt to solve the two-dimensional approximate pattern matching is due to Krithivasan and Sitalakshmi [KS87] but their solution considers the presence of errors along one dimension only. Since then, many papers describing various methods of the two-dimensional exact and approximate pattern matching have appeared, for example the efficient algorithm by Baeza-Yates [BYR93]. As for the one dimensional case, various forms of finite automata have been used in these algorithms.

The research field of tree pattern matching is also relevant to the proposed re-

search. There are different tree algorithms due to many authors. A survey can be found for example in Cleophas [Cle08]. Tree pattern matching using a push-down automaton is due to Janouček and Melichar [JM09, Jan10], and an alternative construction is due to Flouri, Janouček and Melichar [FJM10].

Yet another related field of research is text indexing. Suffix and factor automata for one-dimensional text indexing originated in several works by Blumer et al. and Crochemore [BBE⁺83, BBH⁺85, Cro87]. Other indexing tools, such as suffix trees and suffix arrays are due to Weiner [Wei73], McCreight [McC76], Ukkonen [Ukk95], Farach [Far97], Manber and Myers [MM93].

The first indexing data structure for two dimensional arrays called a PAT-tree has been proposed by Gonnet [Gon88]. (The name PAT-tree refers to Morrison [Mor68] and his one-dimensional structure.) Another variant has appeared in Amir and Farach [AF92]. Since then, two-dimensional suffix trees were addressed by Giancarlo and Grossi [GG97], Kim, Kim and Park [KKP03], and Na, Giancarlo and Park [NGP07].

A novel machine learning approach to DFA construction that is of interest to this research is due to Dana Angluin [Ang87]. The role of the *observation matrix* used in Angluin's algorithm vs the role of the *context* introduced in the next section on Formal Concept Analysis will be investigated. Angluin's algorithm may also be an important point of departure for the development of algorithms in the proposed research.

Further results related to my approach will be investigated during the literary study phase of the proposed research.

1.2.2 Formal Concept Analysis

Intuitively, the notion of a concept is related to the notion of an abstraction, and it was also clear that the ability to abstract is strongly associated with human learning and intelligence. Hence, it is not surprising that, with the rise of research interest in machine learning as a subarea of Artificial Intelligence (AI), there arose also a concurrent interest in developing a mathematically formalised and computationally amenable notion of a *concept*. The instinct was that if one could translate what it means to conceptualise into a computational task, then one would have a platform for machine learning, and hence for AI.

One response to this instinct was a mathematical formalisation of the notion of a concept as developed by a research group in Darmstadt, led by Wille, Ganter and Murmeister in the early 1980s. This formalisation has subsequently formed the heart of a field of study known as formal concept analysis (FCA).

In addition to its application to Machine Learning, FCA offers a powerful and comprehensive approach to clustering and ordering of information. There are several annual conferences and workshops devoted to the field, as well as an FCA mailing list, FCA related software, etc. (See, for example, <http://www.upriss.org.uk/fca/fca.html>.) Ganter and Wille's foundational text on the topic can be found in [GW99], and a subsequent text book by Carpineto and Romano is available in [CR04a].

FCA roots the notion of a concept in a given domain of discourse in a two-dimensional matrix that is called the context. The context consists of a finite number of rows representing objects in the domain of discourse, and a finite number of columns representing the discrete attributes of those objects. A fictitious example of a context representing people on the beach at Ipanema is shown in Figure 1.2.1. The cells of the matrix indicate whether or not an object—one of the named people on the beach—have a given attribute. Thus, either Alice or Amy could be the tall and tanned and young and lovely girl mentioned in the well-known song, Alice being additionally characterised as bright and Amy as big. The boys on the beach, Bob and Bill, also have their attributes noted in the context.

The semantics given to the objects and attributes should be noted. To infer that the first two objects in the context are girls while the latter two are boys goes beyond the data given. If gender was an important attribute, an additional column could be inserted called, say, Female, in which the first two rows receive a cross, and the last two are empty. Alternatively, two additional columns, called Male and Female respectively, could be inserted but this would be somewhat redundant because of the binary nature both of gender and of the attributes—i.e. an object is considered to either have or not have an attribute. However, attributes sometimes fall into non-binary classes such that an object can have only one attribute in the class. For example, there might be a class of attributes—say Black, Red, Brunette, Grey—indicating the hair colour of objects. Non-discrete attributes can also be partitioned into mutually exclusive discrete attributes. For example there could be Short, Medium and Tall attributes characterising objects less than 1.5 meters in height, those

| | Tall | Tanned | Young | Lovely | Big | Fat | Bright |
|-------|------|--------|-------|--------|-----|-----|--------|
| Alice | × | × | × | × | | | × |
| Amy | × | × | × | × | × | | |
| Bob | × | | × | | | | × |
| Bill | | × | | | × | × | |

Figure 1.2.1: The Ipanema Context

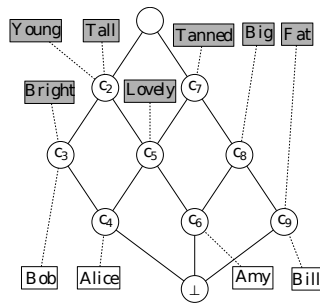


Figure 1.2.2: Line diagram of Ipanema Concept Lattice

between 1.5 and 1.8 meters, and those over 1.8 meters.

The context embeds information that is not immediately evident. It has already been pointed out that the objects Alice and Amy share all and only the attributes Tall and Tanned and Young and Lovely. It is also the case that these attributes do not collectively characterise any objects other than Amy and Alice. FCA therefore views the pair consisting of the set of objects and set of attributes involved in such an exclusive relation as a *concept*. Thus, the context embeds a concept that is designated c_5 (to conform with the line diagram entry to be discussed below) that may be describe as follows:

$$c_5 = \langle \{Alice, Amy\}, \{Tall, Tanned, Young, Lovely\} \rangle$$

The object set $\{Alice, Amy\}$ is called the *extent* of c_5 and is denoted by $ext(c_5)$. Similarly, the attribute set $\{Tall, Tanned, Young, Lovely\}$ is the concept's *intent*, denoted by $int(c_5)$.

The context embeds many more such concepts, two further examples being c_2

and c_8 , respectively defined as:

$$\begin{aligned} c_2 &= \langle \{\text{Alice, Amy, Bob}\}, \{\text{Tall, Young}\} \rangle \\ c_8 &= \langle \{\text{Amy, Bill}\}, \{\text{Tanned, Big}\} \rangle \end{aligned}$$

Furthermore, FCA regards the concepts as being partially ordered by set inclusion on the extents, i.e. for arbitrary concepts c_i and c_j ,

$$c_i \leq c_j \equiv \text{ext}(c_i) \subseteq \text{ext}(c_j)$$

Thus, in the Ipanema beach example, $c_5 \leq c_2$ while c_5 and c_8 are incommensurate.

The full set of concepts that can be induced from a given context are not only partially ordered. They also constitute a complete lattice—i.e. every subset of concepts has a least upper bound and a greatest lower bound. This means that there will always be a top concept (denoted by \top) and bottom concept (denoted by \perp) in such a concept lattice.

Many algorithms have been developed to infer the concept lattice derivable from a given context, to record the ordering of concepts and/or to display the concepts and orderings in a line diagram. Figure 1.2.2 shows the line diagram of concepts derived from the Ipanema context of Table 1.2.1. The Concept Explorer package¹ was used to produce this diagram. Objects and attributes are shown in rectangles connected by dashed lines to various concept nodes, attribute rectangles being shaded. Each of the thirteen nodes in this graph represents a concept, and these are conventionally ordered with the smallest concepts placed at the bottom of the diagram, and the largest concepts at the top.

The extent of any concept may be found by collecting together all objects reachable on downward paths from the concept. A concept's intent is found by collecting together all attributes reachable on upward paths from the concept. In Figure 1.2.2 the top and bottom concepts are respectively:

$$\begin{aligned} \top &= \langle \{\text{Alice, Amy, Bob, Bill}\}, \emptyset \rangle \\ \perp &= \langle \emptyset, \{\text{Tall, Tanned, Young, Lovely, Big, Fat, Bright}\} \rangle \end{aligned}$$

¹Note: Concept Explorer's author requests that users cite his Russian text, [Yev00], as a reference to the package.

An object is called the *own object* of a concept C when that concept is the smallest that includes the object in its extent, and the set of C 's own objects is denoted by $ownobj(C)$. The own object sets of all concepts discussed thusfar in the Ipanema example are empty; for example, $ownobj(c_1) = \emptyset$, $ownobj(\perp) = \emptyset$, etc. However, the concept marked c_4 in Figure 1.2.2 is an example of one which has a non-empty own object set; i.e. $ownobj(c_4) = \{\text{Alice}\}$. In general, Concept Explorer provides links from concepts to their own objects.

There is a clear duality between the role of attributes and objects which will not be spelled out in detail here. Thus, for example, Concept Explorer also links concepts to their so-called own attributes—defined similarly but dually to own objects. This duality is also known as a Galois connection between attributes and objects.

It is a property of a concept lattice that it is set-intersection closed with respect to both the intents and to the extents of concepts—i.e. the intersection of the intents of any two concepts will be an intent of some lattice concept, and dually for extents. Furthermore, the larger a concept, the larger its extent and the smaller its intent. This is clearly seen in the two extremes where $ext(\top)$ in the example lattice is the full set of objects, while $ext(\perp)$ is \emptyset , and dually in respect of their extents. This parallels our instinctive notion of abstraction (generalisation): the more abstract an object, the less specific its properties (thus less attributes in its intent), and the more entities (objects in its extent) it represents.

FCA concept lattices are therefore information-rich—arguably the most information rich representation possible of the relationship between objects and their attributes. However, this richness comes at a cost: the number of concepts in a concept lattice is, in the worst case, exponentially dependent on the number of objects and attributes. Specifically, if N is the minimum over the number of objects and the number of attributes, then the maximum number of concepts is 2^N . Suppose that N is the number of attributes. Then this worst-case scenario arises when there is a set of N objects, each of which is characterised by exactly $N - 1$ attributes, and each of which differs from all the other N objects in that set by exactly one attribute. As the number of entries in the context matrix declines, the number of concepts in the lattice falls to more tractable levels.

1.3 Problem Statements and Research Question

The thesis does not attempt to solve a new problem. It tries to find a new approach to solve previously solved problems in the fields of pattern matching and finite automata using FCA.

This, the thesis provides answers the following questions:

How can FCA be applied to develop algorithms that provide solutions for the following problems:

- Two dimensional pattern matching
- Multiple keyword pattern matching
- Failure automaton construction
- NFA to DFA conversion
- DFA minimisation

For every problem above, what is the advantage of the FCA based solution over existing algorithms if any, and does the FCA based solution provide an important academic point of departure for further research?

1.4 Research Method

The research questions are answered by formally introducing each of the above problems and then deriving at least one FCA based algorithm for that problem. Informal discussion on the advantages and disadvantages of developed algorithms and important contributions that may arise from the new algorithms are provided. Comparisons of time complexity of FCA based algorithms with existing algorithms are also provided. Thus the research method can be characterized as exploratory algorithmics.

1.5 Thesis Outline

This section provides an overview of the thesis chapters.

1.5.1 Preliminaries

This chapter provides formal preliminaries for FCA, Pattern Matching and Automata. This chapter provides the preliminary notations, definitions and properties used throughout the thesis. The reader is thus advised to read this chapter before reading the rest of the thesis.

1.5.2 2D Pattern Matching

In this chapter that is based on [VKW09], a new approach to two-dimensional pattern matching is proposed. The target domain is assumed to consist of 2D rectilinear shapes, such as is typically the case with components on a chip. Satellite images could also be approximated in this fashion. An encoding scheme will be worked out whereby any such rectilinear shape—i.e. pattern—can be described in terms of a finite set of attributes. Figure 1.5.1 shows a purpose built editor to describe and store such patterns in terms of these attributes. The set of patterns to be sought are then regarded as FCA objects described in terms of these attributes and stored in a context. Such a context then in turn yields a concept lattice in which the intent of each concept is a set of attributes held in common by a subset of the pattern objects that are sought. Figure 1.5.2 gives a partial view of an example line diagram used in a 2D search. An algorithm was developed which systematically traverses the search space and as attributes are encountered, a marker is appropriately moved from one concept to the next in the line diagram. It is shown that a hit (i.e. a 2D pattern match) can be inferred when certain concepts in the lattice are encountered.

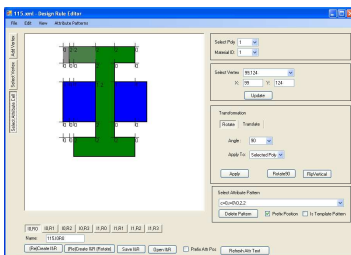


Figure 1.5.1: 2D Pattern Matching Editor

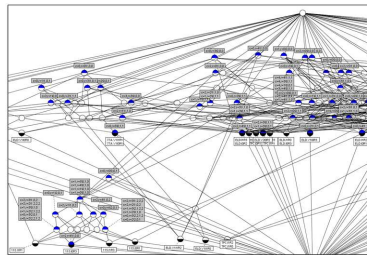


Figure 1.5.2: 2D Pattern Matching Lattice

| | $\langle 1, a \rangle$ | $\langle 2, a \rangle$ | $\langle 2, b \rangle$ | $\langle 3, b \rangle$ | $\langle 3, c \rangle$ | $\langle 4, c \rangle$ |
|------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| abc | × | | × | | × | |
| aabc | × | × | | × | | |
| abcc | × | | × | | × | × |

Figure 1.5.3: Position encoded context for $P = \{abc, aabc, abcc\}$.

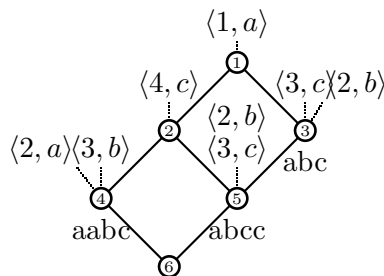


Figure 1.5.4: Line diagram of PEPL for $P = \{abc, aabc, abcc\}$.

1.5.3 Multiple Keyword Pattern Matching

In this chapter the thesis proposes a new way to match multiple keywords on a target string. The approach views a character string as an FCA object whose attributes are pairs of the form $\langle n, c \rangle$, where n is the position of the character c in the string, i.e. the attributes are position-encodings of the characters in the string. Using such a scheme, a set of patterns can be described as objects in a context such as shown in Figure 1.5.3. Thus, the string “abc” has attributes $\langle 1, a \rangle$, $\langle 2, b \rangle$ and $\langle 3, c \rangle$, etc. Once again, such a context yields a concept lattice, which we call a Position Encoded Pattern Lattice (PEPL), whose line diagram appears in Figure 1.5.4.

In this chapter two algorithms are proposed. The first is a somewhat naive algorithm that uses a PEPL to do multiple keyword pattern matching and a second is an improved algorithm that eliminates sets of words from P that do not match in a string s without ever backing up in s , i.e. t , the matching position in s , is monotonically increasing. In this sense the algorithm is an *online* algorithm, similar to the Aho-Corasick algorithm [AC75]. We also present a PEPL-based algorithm that avoids such revisits and thus becomes competitive with the Aho-Corasick algorithm in terms of space and time efficiency.

1.5.4 Failure DFAs

Taking the cue from the way in which failure arcs are defined and used in one of the variants of the Aho-Corasick algorithm, the notion of a failure deterministic finite automaton (FDFA) is introduced. This chapter shows how FCA can be used to derive from a given deterministic finite automaton (DFA) a language-equivalent FDFA [KWCV12].

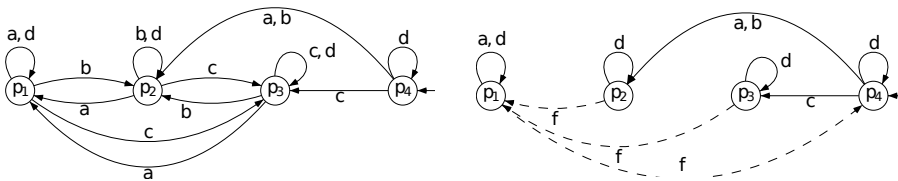


Figure 1.5.5: Initial DFA and an equivalent FDFA.

As a brief survey of the approach, consider the DFA on the left hand side of Figure 1.5.5. The structure on the right hand side of Figure 1.5.5 is an example of an FDFA. The FDFA has three failure transitions (represented by dashed arcs) and labelled f . The semantics of these failure transitions differ from those of normal DFA transitions. In the latter case, if an arc is labelled by the symbol at the head of the string under test, then that symbol is consumed and a transition is made to the arc's destination. In the former case, if the head of the string under test does *not* match the labels on any of the current state's outgoing DFA arcs, then the failure transition is made to the next state, *without consuming* the head of the string under test. Thus, to recognise the string $abca$, the following DFA transitions are made in Figure 1.5.5

$$p_4 \xrightarrow{a} p_2 \xrightarrow{b} p_2 \xrightarrow{c} p_3 \xrightarrow{a} p_1$$

However, in the case of the FDFA in Figure 1.5.5 the transitions made are as follows

$$p_4 \xrightarrow{a} p_2 \xrightarrow{f} p_1 \xrightarrow{f} p_4 \xrightarrow{b} p_2 \xrightarrow{f} p_1 \xrightarrow{f} p_4 \xrightarrow{c} p_3 \xrightarrow{f} p_1 \xrightarrow{a} p_1$$

It can easily be verified that the FDFA recognises the same language as the DFA, and is in this sense equivalent to it. Note that the FDFA has only eight arcs, and three failure transitions (represented by dashed arcs) and labelled f , while the DFA has sixteen arcs. It is a potential savings in arc space such

| | $\langle a, p_1 \rangle$ | $\langle a, p_2 \rangle$ | $\langle b, p_2 \rangle$ | $\langle c, p_3 \rangle$ | $\langle d, p_1 \rangle$ | $\langle d, p_2 \rangle$ | $\langle d, p_3 \rangle$ | $\langle d, p_4 \rangle$ |
|-------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| p_1 | × | | × | × | × | | | |
| p_2 | × | | × | × | | × | | |
| p_3 | × | | × | × | | | × | |
| p_4 | | × | × | × | | | | × |

Figure 1.5.6: The state/out-transition context of DFA in Figure 1.5.5

as this which motivates the introduction of failure transitions. The trade-off to be made is that an additional time cost to consume the entire string is incurred.

However, it is not immediately evident how to build an F DFA that is language-equivalent to a given DFA. The problem is to determine where to position failure transitions and which DFA transitions to remove. It turns out that one can rely on FCA to assist with these decisions. The approach starts with a so-called state / out-transition context in which DFA states are treated as objects and arc-label / arc-destination pairs are treated as the attributes of the objects as shown in Figure 1.5.6. This then leads to a state / out-transition lattice as shown in Figure 5.3.1.

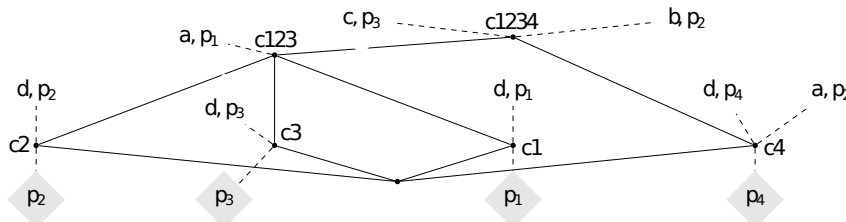


Figure 1.5.7: State/out-transition formal concept lattice of DFA in Figure 1.5.5

The algorithm described in this chapter and [KWCV12] relies on a so-called arc redundancy metric computed for each concept in the lattice. By “greedily” selecting concepts with the highest arc redundancy, sets of DFA transitions can be selected which are replaced by a single failure transition, thereby building a sequence of FDFAs each of which is language-equivalent to its predecessor. The algorithm terminates when it is no longer possible to reduce any more DFA transitions from the structure obtained to date. Although this algorithm does not guarantee optimality (in the sense of maximally reducing DFA transitions

to FDFA failure transitions), empirical data obtained to date suggests that the number of DFA arcs on randomly generated DFAs can be reduced by between 5% and 15%. Because the FDFA produced by this algorithm has the same set of states (and therefore topological layout of the state diagram) as the starting DFA, we have called it the DFA-homomorphic algorithm. Refinements to this algorithm are currently under investigation. An alternative algorithm that adds new FDFA states in accordance with concepts in the state/out-transition lattice is also presented. Because of this state correspondence with the lattice rather than the original DFA, this has been designated a lattice-homomorphic algorithm.

1.5.5 NFA to DFA reduction and ADFA pattern matching using FCA

In this chapter, ideas based on the intersection of FCA and Automata that could be explored in further research are discussed. In the first section, the outline of new algorithms based on FCA to reduce an NFA to an equivalent DFA is proposed. In the second section, new *Position Encoded Pattern Lattice* (PEPL) based algorithms for deriving an ADFA from a given set of keywords are proposed. In the same section, *Reverse Position Encoded Pattern Lattices* (RPEPLs) is introduced and an algorithm to derive an ADFA from a RPEPL of a set of keywords is proposed. The intuitions, algorithms and illustrative examples related to these ideas are presented. Specific questions or hypotheses to explore in further research are provided throughout the chapter.

1.5.6 Conclusion

This chapter reviews important results and conclusions of all chapters as well as possible areas of future research.

Chapter 2

Preliminaries

2.1 Introduction

This chapter provides the preliminary notations, definitions and properties used throughout the thesis. It is divided into the following sections:

- General
- Lattice Theory and Formal Concept Analysis
- Languages and Automata

2.2 General

Notation 2.2.1 (Quantifications). A basic understanding of the meaning of *quantifications* is assumed. We use the following notation due to [Wat95]:

$$(\oplus a : R(a) : f(a))$$

where \oplus is the *symbol* for an associative and commutative *quantification operation* (with unit e_{\oplus}), a is a *dummy variable* introduced, $R(a)$ is a *range predicate* on the dummy variable a , and $f(a)$ is a *quantified expression* on dummy variable a .

By definition, we have:

$$(\oplus a : false : f(a)) = e_{\oplus}$$

The following table lists some of the most commonly quantification operations, their quantification symbols, and their units:

| | | | | | | |
|------------------|--------------|-------------|-------------|------------|------------|----------|
| <i>Operation</i> | \vee | \wedge | \cup | min | max | $+$ |
| <i>Symbol</i> | \exists | \forall | \cup | MIN | MAX | Σ |
| <i>Unit</i> | <i>false</i> | <i>true</i> | \emptyset | $+\infty$ | $-\infty$ | 0 |

□

Example 2.2.2. For example, $(\Sigma x : x \in [1..3] : x) = 6$ and $(\Sigma x : x \in [1..3] : x \times 2) = 12$

Property 2.2.3 (Conjunction and disjunction in **MIN** quantifications). For predicates P , Q and integer function f we have

$$\begin{aligned} (\mathbf{MIN} i : P(i) \wedge Q(i) : f(i)) &\geq (\mathbf{MIN} i : P(i) : f(i)) \mathbf{max} (\mathbf{MIN} i : Q(i) : f(i)) \\ (\mathbf{MIN} i : P(i) \vee Q(i) : f(i)) &= (\mathbf{MIN} i : P(i) : f(i)) \mathbf{min} (\mathbf{MIN} i : Q(i) : f(i)) \end{aligned}$$

□

Notation 2.2.4 (Conditional conjunction/disjunction). We use **cand** and **cor** for *conditional conjunction* and *conditional disjunction* respectively. A conditional conjunction (disjunction) is one in which the second operand is evaluated if and only if this is necessary to determine the value of the conjunction (disjunction). □

Definition 2.2.5 (Set Classes). $\mathbf{set}(C)$ is defined as the set of all sets of elements from a domain C (or set class C). □

Definition 2.2.6 (Set element identifier). For the set S and element $e \in S$ the function $id \in \{S\} \times S \rightarrow \mathbb{N}$ such that $id(S, e)$ gives a unique number in the range $[0..|S|)$ as identifier of e . Note that when a set S is the only set in scope, the first argument may be omitted and $id(e) = id(S, e)$. □

Definition 2.2.7 (i^{th} element of a set). For the set S the notation S_i is used for the i^{th} element of S . Formally:

$$S_i = e | id(e) = i$$

Note that S_i does not imply any order in S in the elements of S other than the *incidental order* introduced by id . □

2.3 Lattice Theory and Formal Concept Analysis

Definition 2.3.1 (A lattice). A lattice is a partially ordered set denoted by (L, \leq) in which every pair of elements $\langle a, b \rangle$ has a *unique least upper bound* (or *supremum* $(L, \langle a, b \rangle)$) and a *unique greatest lower bound* (or *infimum* $(L, \langle a, b \rangle)$). A lattice L is complete if it is *supremum*- and *infimum* dense, i.e. *suprema* and *infima* exist for every subset of L . \square

Notation 2.3.2 (Maximum and minimum elements of a lattice). The maximum element of a lattice L is denoted by \top_L and the minimum element of L is denoted by \perp_L . \square

Notation 2.3.3 (Short form of supremum and infimum). For brevity *supremum*(*infimum*) for lattice L will be denoted *sup*(*inf*). \square

Definition 2.3.4 (descendants function). The function *descendants* gives all the nodes that are less than an element $e \in (L, \leq)$. Formally:

$$\text{descendants}(L, e) = \left(\bigcup c : c \leq e : \{c\} \right)$$

Note that when L exists in the scope of an algorithm or definition, the first argument of *descendants* may be omitted. \square

Definition 2.3.5 (Cover graph of a lattice). The lattice (L, \leq) , can be represented as a line (Hasse) diagram or *cover graph*, (L, \prec_L) . In the cover graph, nodes represent elements of L and $c \prec_L p$ means that a (parent) node representing $p \in L$ is connected by an edge to a (child) node representing $c \in L$. \prec_L is also called the *cover relation* on elements of L and is defined as follows:

$$c \prec_L p \equiv c, p \in L \wedge c \leq p \wedge (\nexists r : r \in L : c < r < p)$$

Note that when L exists in the scope of an algorithm or definition, the unsubscripted version of the cover relation, i.e. \prec may also be used.

□

Definition 2.3.6 (children function). The function *children* gives all the nodes directly “below” a node p in the cover graph of L . Formally:

$$\text{children}(L, p) = (\bigcup c : c \prec_L p : \{c\})$$

Note that when L exists in the scope of an algorithm or definition, the first argument of *children* may be omitted.

□

Definition 2.3.7 (Formal context). We define a *formal context* as a triple $\mathbb{K} = \langle G, M, I \rangle$ where G and M are sets and I is an incidence relation between G and M . The elements of G and M are called *objects* and *attributes* respectively. We say that the object $g \in G$ has the attribute $m \in M$ if gIm or, put differently, if $\langle g, m \rangle \in I$.

□

Definition 2.3.8 (Formal concept). We now consider the definition of a *formal concept*. We first define operators *att* and *obj* as follows:

$$\text{att}(A) = (\bigcup m : m \in M \wedge g \in A \wedge gIm : \{m\})$$

$$\text{obj}(B) = (\bigcup g : g \in G \wedge m \in B \wedge gIm : \{g\})$$

For $A \subseteq G$, $B \subseteq M$, a pair $\langle A, B \rangle$ such that $\text{att}(A) = B$ and $\text{obj}(B) = A$ is called a *formal concept*.

□

Definition 2.3.9 (Extent and intent of formal concepts). For a *formal concept* $\langle A, B \rangle$ the set A is called the *extent* and the set B the *intent* of the concept.

Helper operators to access the *extent* and the *intent* of a concept c in any concept lattice are denoted $\text{ext}(c)$ and $\text{int}(c)$ respectively.

□

Definition 2.3.10 (Partial order of formal concepts). Formal concepts are partially ordered by

$$\langle A_1, B_1 \rangle \leq \langle A_2, B_2 \rangle \Leftrightarrow A_1 \subseteq A_2 (\Leftrightarrow B_2 \subseteq B_1)$$

□

Definition 2.3.11 (Concept lattice). With respect to this partial order and the formal context $\mathbb{K} = \langle G, M, I \rangle$, the set of all *formal concepts*, denoted by $\mathfrak{B}(\langle G, M, I \rangle)$, forms a complete lattice called the *concept lattice* of the context.

□

Definition 2.3.12 (Sub-concept and super-concept). For concepts $\langle A_1, B_1 \rangle$ and $\langle A_2, B_2 \rangle$ that belong to \mathfrak{B} , such that $\langle A_1, B_1 \rangle \leq \langle A_2, B_2 \rangle$, $\langle A_1, B_1 \rangle$ is called the *sub-concept* of $\langle A_2, B_2 \rangle$ and $\langle A_2, B_2 \rangle$ is called the *super-concept* of $\langle A_1, B_1 \rangle$. It follows that in terms of the cover graph of \mathfrak{B} , the following holds:

$$\langle A_1, B_1 \rangle \prec_{\mathfrak{B}} \langle A_2, B_2 \rangle$$

□

Definition 2.3.13 (*Attribute Top* and *Object Bottom*). For the concept lattice $\mathfrak{B}(\langle G, M, I \rangle)$ where we define the *Attribute Top*, $\top \in M \rightarrow \mathfrak{B}$ and the *Object Bottom*, $\perp \in G \rightarrow \mathfrak{B}$:

$$\top(x) = c \in \mathfrak{B} \mid x \in \text{int}(c) \wedge (\nexists d \in \mathfrak{B} \mid x \in \text{int}(d) \wedge c \prec_B d)$$

$$\perp(y) = c \in \mathfrak{B} \mid y \in \text{ext}(c) \wedge (\nexists d \in \mathfrak{B} \mid y \in \text{ext}(d) \wedge d \prec_B c)$$

Informally, the *Attribute Top* of an attribute x , $\top(x)$, is the concept $c \in \mathfrak{B}$, such that $x \in \text{int}(c)$ and no other concept $d \in \mathfrak{B}$ exists, such that $x \in \text{int}(d)$ and d is a super-concept of c . $\top(x)$ can be regarded as the “highest concept”, c , in the Hasse diagram of the lattice \mathfrak{B} that contains attribute x as an element of its intent.

Informally, the *Object Bottom* of the object y , $\perp(y)$, is the concept $c \in \mathfrak{B}$, such that $y \in \text{ext}(c)$ and no other concept $d \in \mathfrak{B}$ exists, such that $y \in \text{ext}(d)$ and d is a sub-concept of c . It can also be said that it is the “lowest concept” c in the Hasse diagram of the lattice \mathfrak{B} such that the object y is an element of its extent.

Notation 2.3.14 (Symbols for the *Top* and *Bottom* concepts of a lattice). As a notational convention, the symbols \top and \perp are overloaded so that if either of these symbols are used without parenthesis or without arguments, they represent the top concept and bottom concept of a lattice respectively.

Definition 2.3.15 (*Own Objects* of a concept). For the concept lattice \mathfrak{B} we define the *Own Objects* of a concept as $ownobj \in \mathfrak{B}(\langle G, M, I \rangle) \rightarrow \mathcal{P}(G)$ such that :

$$ownobj(c) = \left(\bigcup x : \perp(x) = c : \{x\} \right)$$

Informally we used the relation \perp to find all the objects x in the extent of a concept c such that c is also the *Object Bottom* of x .

□

Definition 2.3.16 (*Own Attributes* of a concept). For the concept lattice \mathfrak{B} we define the *Own Attributes* of a concept as $ownatt \in \mathfrak{B}(\langle G, M, I \rangle) \rightarrow \mathcal{P}(M)$ such that :

$$ownatt(c) = \left(\bigcup x : \top(x) = c : \{x\} \right)$$

Informally we used the relation \perp to find all the objects x in the intent of a concept c such that c is also the *Attribute Top* of x .

□

Notation 2.3.17 (*Concept identifier*). If \mathfrak{L} is the set of concepts of a concept lattice, a concept $c \in \mathfrak{L}$ will often be referred to by its identifier in \mathfrak{L} , i.e. $id(c)$. Conversely, $\diamond i$ gives the concept associated with concept identifier i . Thus $\diamond id(c) = c$

2.4 Languages and Automata

Definition 2.4.1 (Alphabet, Words and Language). Given the alphabet V , a non-empty finite set of symbols:

- $V^+ = \left(\bigcup V^n : n \in [1 \dots \infty] : \{V^n\} \right)$ where V^n is a word of length n , is the set of non-empty words over V .

- $V^* = \{\varepsilon\} \cup V^+$ is the set of words over V (including the empty word ε).
- $L \subseteq V^*$ is a language over V .

□

Definition 2.4.2 (String reversal function R). Assuming alphabet V , we define string reversal function R recursively by $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$ (for $a \in V, w \in V^*$). We will use R on sets of strings as well. □

Definition 2.4.3 (Functions **pref**, **suff** and **fact**). For any given alphabet V , define **pref** $\in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$, **suff** $\in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ and **fact** $\in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ as

$$\begin{aligned} \mathbf{pref}(L) &= (\cup x, y : xy \in L : \{x\}) \\ \mathbf{suff}(L) &= (\cup y, z : yz \in L : \{z\}) \\ \mathbf{fact}(L) &= (\cup x, y, z : xyz \in L : \{y\}) \end{aligned}$$

Informally, **pref**(L) (**suff**(L), **fact**(L)) is the set of all strings which are (not necessarily proper) prefixes (suffixes, factors) of strings in L . □

Notation 2.4.4 (String arguments to functions **pref**, **suff** and **fact**). For string $w \in V^*$, we will write **pref**(w) (**suff**(w), **fact**(w)) instead of **pref**($\{w\}$) (**suff**($\{w\}$), **fact**($\{w\}$)). □

Definition 2.4.5 (Length of a string). The length of string p will be denoted by $|p|$ and its $(i + 1)^{st}$ element by p_i for $i \in [0, |p|)$. □

Definition 2.4.6 (String operators $\lceil, \lfloor, \lrcorner, \llcorner$). Assuming alphabet V , we define four infix operators $\lceil, \lfloor, \lrcorner, \llcorner \in V^* \times \mathbb{N} \rightarrow V^*$ as follows:

- $w \lceil k$ is a prefix p of w such that $|p| = k \mathbf{min} |w|$
- $w \lfloor k$ is a suffix s of w such that $|s| = (|w| - k) \mathbf{max} 0$
- $w \lrcorner k$ is a suffix s of w such that $|s| = k \mathbf{min} |w|$
- $w \llcorner k$ is a prefix p of w such that $|p| = (|w| - k) \mathbf{max} 0$

The four operators are pronounced ‘left take’, ‘left drop’, ‘right take’ and ‘right drop’ respectively. □

Property 2.4.7 (String operators $\lceil, \lfloor, \lrcorner, \llcorner$). For string operator $\lceil, \lfloor, \lrcorner$ and \llcorner ,

$$\begin{aligned} (w \lceil k)(w \lfloor k) &= w \\ (w \lfloor k)(w \lrcorner k) &= w \end{aligned}$$

□

Example 2.4.8 (String operators \uparrow , \downarrow , \lceil , \lfloor). $(hers)\uparrow 3 = her$, $(hers)\downarrow 1 = ers$, $(hers)\lceil 5 = hers$ and $(hers)\lfloor 10 = \varepsilon$. □

Definition 2.4.9 (Sub strings). We define the sub-string operator $[..] \in V^* \times \mathbb{N} \times \mathbb{N} \rightarrow V^*$ in terms of the operators \downarrow and \uparrow .

$$w[x..y] = w\downarrow x\uparrow y - x$$

□

Notation 2.4.10 (Single symbol sub-string). For string $w \in V^*$, we will write $w[x]$ instead of $w[x..x]$. □

Property 2.4.11 (Idempotence of **pref**, **suff** and **fact**). **pref**, **suff** and **fact** are idempotent. □

Property 2.4.12 (Relationship between **fact** and **suff**, **pref**). Function **fact** can also be defined in terms of the functions **suff** and **pref**:

$$\mathbf{fact}(L) = \mathbf{pref}(\mathbf{suff}(L))$$

and

$$\mathbf{fact}(L) = \mathbf{suff}(\mathbf{pref}(L)).$$

Proof: We will prove only the first equality. The proof of the second is similar.

$$\begin{aligned} & y \in \mathbf{pref}(\mathbf{suff}(L)) \\ = & \quad \{ \text{definition of } \mathbf{pref} \} \\ & (\exists z :: yz \in \mathbf{suff}(L)) \\ = & \quad \{ \text{property of } \mathbf{suff} \} \\ & (\exists z :: (\exists x :: xyz \in L)) \\ = & \quad \{ \text{nesting} \} \\ & (\exists x, z :: xyz \in L) \\ \equiv & \quad \{ \text{definition of } \mathbf{fact} \} \\ & y \in \mathbf{fact}(L) \end{aligned}$$

□

Property 2.4.13 (Duality of **pref** and **suff**). Functions **pref** and **suff** are each other's duals. This can be seen as follows:

$$\begin{aligned}
& x \in \mathbf{pref}(L^R) \\
\equiv & \quad \{ \text{property of } \mathbf{pref} \} \\
& (\exists y :: xy \in L^R) \\
\equiv & \quad \{ \text{property of operator } R \} \\
& (\exists y :: y^R x^R \in L) \\
\equiv & \quad \{ \text{change of bound variable: } y' = y^R \} \\
& (\exists y' :: y' x^R \in L) \\
\equiv & \quad \{ \text{property of } \mathbf{suff} \} \\
& x^R \in \mathbf{suff}(L) \\
\equiv & \quad \{ \text{property of operator } R \} \\
& x \in \mathbf{suff}(L)^R
\end{aligned}$$

□

Property 2.4.14 (Symmetry of fact). Function **fact** is symmetrical. This can be seen as follows:

$$\begin{aligned}
& \mathbf{fact}(L^R) \\
\equiv & \quad \{ \text{Property 2.4.12} \} \\
& \mathbf{pref}(\mathbf{suff}(L^R)) \\
\equiv & \quad \{ \text{(dual of) Property 2.4.13} \} \\
& \mathbf{pref}(\mathbf{pref}(L)^R) \\
\equiv & \quad \{ \text{Property 2.4.13} \} \\
& \mathbf{suff}(\mathbf{pref}(L))^R \\
\equiv & \quad \{ \text{Property 2.4.12} \} \\
& \mathbf{fact}(L)^R
\end{aligned}$$

□

Definition 2.4.15 (Prefix and suffix partial orderings). Partial orders $\leq_p, <_p$,

\leq_s and $<_s$ over $V^* \times V^*$ are defined as

$$\begin{aligned} u \leq_p v &\equiv u \in \mathbf{pref}(v) \\ u <_p v &\equiv u \in \mathbf{pref}(v) \setminus \{v\} \\ u \leq_s v &\equiv u \in \mathbf{suff}(v) \\ u <_s v &\equiv u \in \mathbf{suff}(v) \setminus \{v\} \end{aligned}$$

□

Definition 2.4.16 (Concatenation of languages). Language concatenation is an infix operator $\cdot \in \mathcal{P}(V^*) \times \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ (the dot) defined as

$$L \cdot L' = (\cup x, y : x \in L \wedge y \in L' : \{xy\})$$

The singleton language $\{\varepsilon\}$ is the unit of concatenation and the empty language \emptyset is the zero of concatenation □

Notation 2.4.17 (Concatenation of languages using juxtaposition). Juxtaposition is often used in stead of the infix operator \cdot (i.e. LL' is used in stead of $L \cdot L'$). □

Property 2.4.18 (Language intersection). If A and B are languages over alphabet V and $a \in V$, then

$$\begin{aligned} V^*A \cap V^*B \neq \emptyset &\equiv V^*A \cap B \neq \emptyset \vee A \cap V^*B \neq \emptyset \\ V^*aA \cap V^*B \neq \emptyset &\equiv V^*aA \cap B \neq \emptyset \vee A \cap V^*B \neq \emptyset \end{aligned}$$

□

Definition 2.4.19 (Non-Deterministic Finite Automaton). A non-deterministic finite automaton (*NFA*), is a 5-tuple $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$ where

- Q is a finite set of states.
- Σ is an alphabet.
- $\delta \in Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is a transition relation.
- $s \in Q$ is a start state.
- $F \subseteq Q$ is a set of final states.

□

Definition 2.4.20 (Path through an *NFA*). This definition is due to [Wat10].

Given *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$, for state p and $w \in \Sigma^*$,

$$[p \xrightarrow{w}]$$

is the sequence of states $p, \dots, \delta^*(p, v)$ where v is the longest prefix of w such that $\delta^*(p, v) \neq \perp$. We refer to this as the single “ w -path from state p .”

Notation 2.4.21 (Path through a *NFA*). This notation is due to [Wat10]. We use the standard parentheses notation to denote state sequences which are open at the beginning or end — for example $(p \xrightarrow{w})$ does not include p but *does* include the rest of $[p \xrightarrow{w}]$. In some contexts, we may pass a path $[p \xrightarrow{w}]$ as an argument to a predicate or function which expects a *set*, thereby implicitly treating the path as a set of states.

Property 2.4.22. This property is due to [Wat10].

Given *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$, we can give a recursive definition for $[p \xrightarrow{w}]$:

$$[p \xrightarrow{\varepsilon}] = p$$

and, for all $a \in \Sigma, w \in \Sigma^*$ (where \cdot is sequence concatenation and ε is the empty sequence which some authors write as $[\]$)

$$[p \xrightarrow{aw}] = p \cdot \begin{cases} [\delta(p, a) \xrightarrow{w}] & \text{if } a \in \Sigma_p \\ \varepsilon & \text{otherwise} \end{cases}$$

Definition 2.4.23 (Useless state). This definition is due to [Wat10].

For the *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$, a state p is useless if there is no path from the start state (s) to p , or there is no path from p to a final state $f \in F$.

□

Definition 2.4.24 (Out-transitions on a state). Given the *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$ the function $out(p) \in \mathcal{P}(Q) \rightarrow \mathcal{P}(\delta)$ gives all out-transitions from p . Formally:

$$out(p) = \left(\bigcup a : a \in \Sigma \wedge \delta(p, a) \neq \emptyset : \{ \langle p, a, \delta(p, a) \rangle \} \right)$$

□

Definition 2.4.25 (Extending the transition relation). For the *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$ we extend transition relation $\delta \in Q \times \Sigma \leftrightarrow Q$ to $\delta^* \in Q \times \Sigma^* \leftrightarrow Q$ as follows:

$$\delta^*(q, v) = \delta^*(p, w) \iff (\forall a : a \in \Sigma_q : \langle q, a, p \rangle \in \delta \wedge aw \in \Sigma^*)$$

Definition 2.4.26 (Language of a *NFA*). The language of the *NFA* $\mathcal{N} = \langle Q, \Sigma, \delta, s, F \rangle$ is given by the function $\mathcal{L}_{NFA} \in NFA \rightarrow \mathcal{P}(\Sigma^*)$, defined as: □

$$\mathcal{L}_{NFA}(\mathcal{N}) = \left(\bigcup f : f \in F : \{\delta^*(s, f)\} \right)$$

Definition 2.4.27 (Right language of a state). The right language of a state q , denoted $\vec{\mathcal{L}}(q)$ is defined by □

$$\vec{\mathcal{L}}(q) = \{w \mid \delta^*(q, w) \in F\}$$

Definition 2.4.28 (Deterministic Finite Automaton). A deterministic finite automaton, or *DFA*, is a 5-tuple $\mathcal{M} = \langle Q, V, \delta, q_0, F \rangle$ where □

- Q is a finite set of states.
- V is an alphabet.
- $\delta \in Q \times V \rightarrow Q$ is a transition relation.
- $q_0 \in Q$ is a start state.
- $F \subseteq Q$ is a set of final states.

Definition 2.4.29 (Acyclic Deterministic Finite Automaton). An acyclic deterministic finite automaton, or *ADFA*, is a *DFA* in which no path leads from a state to itself. □

Definition 2.4.30 (Minimal Deterministic Finite Automaton). A minimal deterministic finite automaton, or *MDFA*, is a *DFA* M_m such that no other *DFA* M exists that recognises the same language as M_m and has fewer states than M_m . □

Definition 2.4.31 (Minimal Acyclic Deterministic Finite Automaton). A minimal acyclic deterministic finite automaton, or *MADFA*, is an *MDFA* in which no path leads from a state to itself. □

□

Definition 2.4.32 (Mealy Machine). A Mealy Machine is a 5-tuple $\langle Q, V, \Gamma, \delta, \omega, q_0 \rangle$ where

- Q is a finite set of states.
- V is an input alphabet.
- Γ is an output alphabet.
- $\delta \in Q \times V \rightarrow Q$ is a transition relation.
- $\omega \in Q \times V \rightarrow \Gamma$ is an output relation.
- $q_0 \in Q$ is a start state.

□

Definition 2.4.33 (Moore Machine). A Moore Machine is a 5-tuple $\langle Q, V, \Gamma, \delta, \omega, q_0 \rangle$ where

- Q is a finite set of states.
- V is an input alphabet.
- Γ is an output alphabet.
- $\delta \in Q \times V \rightarrow Q$ is a transition relation.
- $\omega \in Q \rightarrow \Gamma$ is an output relation.
- $q_0 \in Q$ is a start state.

□

Part II

Formal Concept Analysis applied to pattern matching

Chapter 3

FCA based Two Dimensional Pattern Matching

3.1 Preamble

This chapter is based on the paper [VKW09]. The content has been significantly updated to provide greater clarity and coherence with the overall contents of this thesis.

3.2 Introduction

In this chapter an approach based on FCA to solve a specific matching problem in the field of microchip design is proposed. This matching problem is to find all positions in a *layout* where one or more *design rules* match. The next section describes layouts and design rules and provides some background to the application domain to which the proposed approach applies. Further sections describe the transformation of geometric properties of layouts and design rules to 2D images and introduce the outline of a matching algorithm that uses a concept lattice derived from 2D images of design rules as underlying data structure.

The experimental tools developed and results of this matching approach as applied to some existing design rules are discussed and some promising findings

on the efficiency of this approach are presented.

3.3 Background on the application domain

In the domain of microchip manufacturing, a step that is often followed to derive a new version of a microchip is to reduce the surface area of (or “compress”) the respective microchip. This step is often implemented by software that belongs to the category of software called *Electronic Design Automation* (EDA) software. EDA software is used throughout the microchip design and manufacturing process. Of special interest for this thesis is an artifact created and maintained using EDA software that represents the design of a microchip called a *layout*.

A layout normally contains the vector graphical representation of the design of a target microchip to be manufactured. More specifically, for the purpose of this thesis, a layout is essentially a very large set of rectilinear polygons that are spatially arranged within the physical 2D area of the target microchip. Thus, to reduce the 2D area of the next generation of a microchip by some reduction factor, a simple approach would be to reduce the 2D area of all polygons in the current version of the layout of the microchip by the respective reduction factor. However, for various reasons that are out of scope of this thesis, this simple approach does not work in practice. It turns out that at specific regions in the target layout, the reduction factor needs to be different in some fashion from the global target reduction factor.

To characterize such exceptional regions in the layout, so-called *design rules* are defined by design engineers. Every such design rule is used by layout area reduction software to search for regions in the target layout that are homomorphic with the respective design rule and to then apply a local area reduction factor for every found region in the target layout. There are various ways to represent a design rule. For the purpose of this thesis, a design rule is represented as a set of rectilinear polygons. Thus, matching a design rule in a target layout will require an algorithm that finds all positions in the target layout where there is a set of polygons in the target layout that is homomorphic to all the polygons in the design rule. In further sections it is shown how the representations of layouts and design rules are converted to 2D images in order to use FCA based 2D pattern matching to achieve this goal.

3.4 Preliminaries related to layouts and design rules

This section provides some preliminary definitions related to layouts and design rules used to derive a formal context and concept lattice from design rules in later sections.

3.4.1 Layouts and Design Rules In Terms of V-Sets

To simplify pattern matching of design rules in layouts, layouts and design rules are treated as structurally equivalent. For brevity we refer to a *layout* or *design rule* or any subset of a layout or design rule using the collective term *v-set*.

A *v-set* is a set of vertices.

A vertex v in such a set is defined as the 5-tuple $\langle x_0, x_1, p, q, r \rangle$ where x_0 (x_1) is a the first (second) dimension displacement in the 2D space called *The Vertex Plane*; p is the index (into some global table of polygons) of the polygon to which the vertex belongs; q is the numeric identifier of the vertex in the respective polygon; and r is the respective polygon's material identifier (an index into some global table of materials).

We also define utility functions to access elements of v as follows:

$$\begin{aligned} x(v) &= x_0 \\ y(v) &= x_1 \\ p(v) &= p \\ q(v) &= q \\ r(v) &= r \end{aligned}$$

If two distinct vertices v_1, v_2 in vertex set s are such that $x(v_1) = x(v_2)$ and

$y(v1) = y(v2)$, then it will be the case that $p(v1) \neq p(v2)$ — i.e. the vertices co-incide in the vertex plane but belong to different polygons.

Given v -set s , R_s is defined as the *smallest* rectangle in *The Vertex Plane* that covers s .

Formally

$$R_s = \langle \langle \mathbf{min}(X), \mathbf{min}(Y) \rangle, \langle \mathbf{max}(X), \mathbf{max}(Y) \rangle \rangle$$

where

$$\begin{aligned} X &= \left(\bigcup v : v \in s : \{x(v)\} \right) \\ Y &= \left(\bigcup v : v \in s : \{y(v)\} \right) \end{aligned}$$

Given the set \mathbb{V} as all vertices, a v -set $s \in \mathcal{P}(\mathbb{V})$ and \mathbb{N} , the set of non-negative integers, a function *polygon* is defined as a function on s that returns the subset of s containing only the vertices that belong to a specific polygon. It is defined as follows:

$$polygon \in \mathbb{N} \times \mathcal{P}(\mathbb{V}) \rightarrow \mathcal{P}(\mathbb{V})$$

satisfying

$$polygon(a, s) = \left(\bigcup v : v \in s \wedge a = p(v) : \{v\} \right)$$

To illustrate v -sets and their relation to polygons, consider polygons $polygon_1$ and $polygon_2$ in Figure 3.4.1.

Assume $polygon_1$ is made of material with id 0 (indicated by darker fill colour of $polygon_1$) and $polygon_2$ is made of material with id 1 (indicated by lighter fill colour of $polygon_2$). Also assume that x_0 (first element of a vertex tuple) starts at 0 on the left and increases to the right of the diagram and that x_1

(second element of a vertex tuple) starts at 0 at the bottom of the diagram and increases towards the top of the diagram. The units for x_0 and x_1 are not of importance in this illustration.

Now consider v_1 , the top-left vertex of $polygon_1$ and v_9 , the top-left vertex of $polygon_2$. According to the above definitions their values are as follows:

$$\begin{aligned} v_1 &= \langle 0, 90, 1, 0, 0 \rangle \\ v_9 &= \langle 0, 20, 2, 0, 1 \rangle \end{aligned}$$

To illustrate the utility functions above, they are applied to v_1 as follows:

$$\begin{aligned} x(v_1) &= 0 \\ y(v_1) &= 90 \\ p(v_1) &= 1 \\ q(v_1) &= 0 \\ r(v_1) &= 0 \end{aligned}$$

If we assign to s the complete set of vertices in the figure, then:

$$\begin{aligned} s &= \{v_1, v_2, v_3, \dots, v_{12}\} \\ &= \{\langle 0, 90, 1, 0, 0 \rangle, \langle 60, 90, 1, 1, 0 \rangle, \langle 60, 70, 1, 2, 0 \rangle, \dots, \langle 0, 5, 2, 3, 1 \rangle\} \end{aligned}$$

To illustrate the function $polygon$, the subset of vertices in the figure denoted $polygon_1$ is obtained by calling $polygon$ as follows:

$$\begin{aligned} polygon(1, s) &= \{v_1, v_2, v_3, \dots, v_8\} \\ &= \{\langle 0, 90, 1, 0, 0 \rangle, \langle 60, 90, 1, 1, 0 \rangle, \langle 60, 70, 1, 2, 0 \rangle, \dots, \\ &\quad \langle 0, 70, 1, 7, 0 \rangle\} \end{aligned}$$

The subset of vertices in the figure denoted $polygon_2$ is obtained by calling $polygon$ as follows:

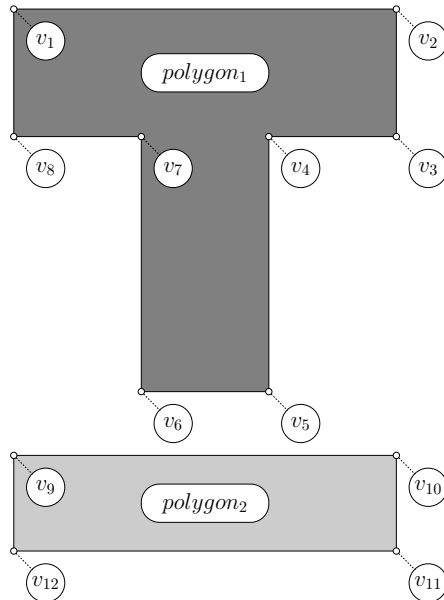


Figure 3.4.1: Example v-set for two polygons

$$\begin{aligned}
 \text{polygon}(2, s) &= \{v_9, v_{10}, v_{11}, v_{12}\} \\
 &= \{\langle 0, 20, 2, 0, 1 \rangle, \langle 60, 20, 2, 1, 1 \rangle, \langle 60, 5, 2, 2, 1 \rangle, \langle 0, 5, 2, 3, 1 \rangle\}
 \end{aligned}$$

3.4.2 From V-Sets to V-Images

One option to solve the problem addressed by this chapter is to match a v-set on another v-set. However the approach proposed here applies matching one *v-image* on another *v-image*. A v-image is a 2D data structure formed from a v-set. Many existing 2D pattern matching algorithms can thus be applied for matching on these v-images. Examples of such algorithms are by Baeza-Yates in [BYR93], by Bird in [Bir77], by Baker in [Bak78] and by Zhu and Takaoka in [RT89].

Definition 3.4.1 (A *v-image* derived from a *v-set*). A v-image is defined as follows in terms of a v-set:

- i is the v-image to be defined in terms of the v-set s that is covered by rectangle R_s in a vertex plane.
- P is the set of polygons defined by vertices in s , i.e.

$$P = \left(\bigcup \text{pol} : v \in s \wedge t = p(v) \wedge \text{pol} = \text{polygon}(t, s) : \{\text{pol}\} \right)$$

- Recall that X and Y were defined above respectively as the sets of first and second dimension displacements of all vertices in s . Suppose that w and h denote the respective sizes of these sets. A v-image is represented in a 2D matrix called *the image plane* that has w columns and h rows.
- Let x_n be the n^{th} largest value in X and y_m be the m^{th} largest value in Y , whereby $n \in [0, w)$ and $m \in [0, h)$. Then the cell $\langle n, m \rangle$ of the image plane is associated with the vertex in $\langle x_n, y_m \rangle$ of the vertex plane. As a result, each cell $\langle n, m \rangle$ of the image plane may be uniquely associated with a *rectangle* in the vertex plane whose top left-hand corner is $\langle x_n, y_m \rangle$. The cells of the image plane (2D matrix) may therefore be regarded as partitioning R_s into non-overlapping rectangles. Note that there is no need to assume that the physical sizes of these rectangles are equal.
- Note that there could be more than one vertex, say $v_1, v_2 \in s$ such that $\langle x(v_1), y(v_1) \rangle = \langle x(v_2), y(v_2) \rangle = \langle x_n, y_m \rangle$. However, v_1, v_2 would then differ from one another in terms of $p(v_1), p(v_2)$ (the identifiers of the respective polygons). They may also possibly differ in terms of $q(v_1), q(v_2)$ (the vertex identifier of the respective polygons) and $r(v_1), r(v_2)$ (the material type of the respective polygons). Since each $\langle x_n, y_m \rangle$ is uniquely associated with a cell $\langle n, m \rangle$ of the image plane, each such cell (representing a rectangle) might be associated with 0, 1 or more of the polygons represented by the vertices in s . The entry in cell $\langle n, m \rangle$ of the image plane (2D matrix) is a *set of material identifiers* of polygon(s) associated with this cell.
- Note also that not all vertices within R_s are necessarily associated with a polygon. Some may correspond to “empty space” within R_s . Since each vertex is associated with an index into a global materials table, the assumption is made that index 0 references a “no material” entry in this table. Consequently, entries of 0 in the image plane characterise upper left hand rectangles representing empty space.
- For example, in Figure 3.4.2 there are two “real” rectilinear objects —

a blue and a green one — whose materials are denoted by 1 and 2 respectively. R_s is not shown explicitly in this figure, but white spaces within the implied R_s also define rectilinear polygons. The figure does not explicitly show the complete non-overlapping rectangles, but marks their top left-hand corners and gives the associated material number(s). Note that the rectangle associated with the overlap of the two objects is marked with both a 1 and 2, because it is associated with two materials.

- Each non-overlapping rectangle, generically denoted by r , is associated with a *pixel*. A pixel is defined as the triple $\langle n, m, u \rangle$, where $\langle n, m \rangle$ is a cell in the image plane and u is the value of the pixel. The pixel's value is the triple $\langle x, y, t \rangle$ where $\langle x, y \rangle$ specifies the upper left-hand vertex of r and t is the set of materials associated with r .
- The v-image, i , is thus the set of pixels derived from the v-set s

□

3.4.3 Functions and operators related to v-images

Definition 3.4.2 (Utility functions on pixels). As with vertices, utility functions to access elements of a pixel $p = \langle n, m, u \rangle$ with $u = \langle x, y, t \rangle$ are defined as follows:

$$\begin{aligned} n(p) &= n \\ m(p) &= m \\ x(p) &= x \\ y(p) &= y \\ t(p) &= t \end{aligned}$$

□

Definition 3.4.3 (v-image derivation function). Let \mathbb{P} be the universal set of pixels and \mathbb{V} the universal set of v-sets. Given a v-set s , X and Y the respective sets of first and second dimension offsets of all vertices in s , the function $img \in \mathcal{P}(\mathbb{V}) \rightarrow \mathcal{P}(\mathbb{P})$ derives a v-image from s , satisfying

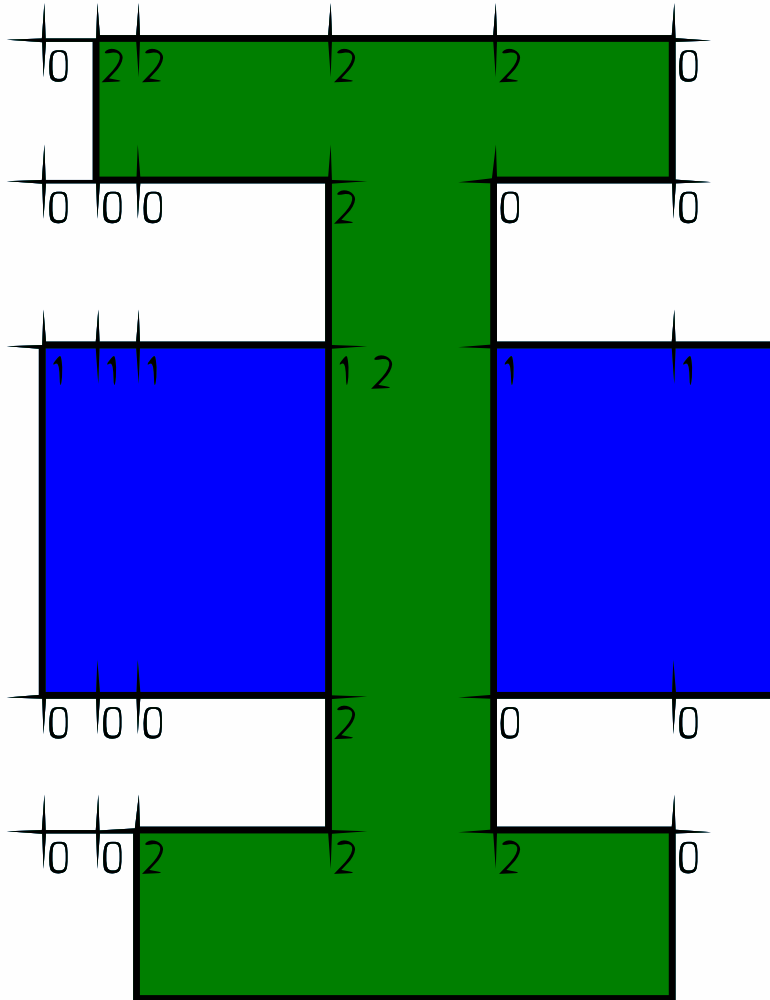


Figure 3.4.2: "Example of an image overlaid on a v-set."

$$\text{img}(s) = \left(\bigcup n, m : n \in [0, |X|) \wedge m \in [0, |Y|) : \{ \langle n, m, \langle x_n, y_m, \text{mat}(s, x_n, y_m) \rangle \} \right)$$

where x_n is the n^{th} largest value in X and y_m is the m^{th} largest value in Y and the function $\text{mat} \in \mathcal{P}(\mathbb{V}) \in \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$ gives the set of material identifiers of all vertices $v \in s$ at the same location, i.e.

$$\text{mat}(s, x_n, y_m) = \left(\bigcup v : v \in s \wedge x(v) = x_n \wedge y(v) = y_m : \{r(v)\} \right)$$

□

Definition 3.4.4 (Utility functions and operators on v-images). The following utility functions and operators on v-images are used in further definitions and sections:

- The width w of a v-image i is given by the function $\mathbf{w}(i)$
- The height h of a v-image i is given by the function $\mathbf{h}(i)$
- The size of a v-image i with width w and height h is given by the operator $\| \in \mathcal{P}(\mathbb{P}) \rightarrow \mathbb{N} \times \mathbb{N}$ satisfying $\|i\| = \langle w, h \rangle$
- Given v-image i , a pixel of i can be referenced using operator $[,] \in \mathcal{P}(\mathbb{P}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}$ satisfying $i[j, k] = p \mid p \in i \wedge n(p) = j \wedge m(p) = k$
- Given v-image i , a sub-v-image of i can be accessed using operator $[\langle, \rangle .. \langle, \rangle] \in \mathcal{P}(\mathbb{P}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{P}(\mathbb{P})$ satisfying $i[\langle l, t \rangle .. \langle r, b \rangle] = \left(\bigcup j, k : j \in [l..r] \wedge k \in [t..b] : \{i[j, k]\} \right)$

□

Definition 3.4.5 (Sub-v-image of a v-image of a given size). Given v-image i , a sub-v-image of i of a given size is returned by function $\mathbf{isub} \in \mathcal{P}(\mathbb{P}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{P}(\mathbb{P})$, satisfying

$$\mathbf{isub}(i, l, t, w, h) = i[\langle l, t \rangle .. \langle l + w, t + h \rangle]$$

□

Definition 3.4.6 (Set of sub-v-images of a given size from a v-image). Given the v-image i and a sub-v-image size $\langle w, h \rangle$, the set of (location, sub-v-image)

pairs within i is returned by function $\mathbf{isuba} \in \mathcal{P}(\mathbb{P}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{I})$, satisfying

$$\mathbf{isuba}(i, w, h) = \left(\bigcup l, t : l \in [0..w(i) - w] \wedge t \in [0..h(i) - h] : \{ \langle l, t, \mathbf{isub}(i, l, t, w, h) \rangle \} \right)$$

□

Definition 3.4.7 (Mirror of a v-image). The function $mirror \in \mathcal{P}(\mathbb{P}) \rightarrow \mathcal{P}(\mathbb{P})$ creates a mirror v-image of a given v-image, satisfying:

$$mirror(i) = \left(\bigcup p : p \in i : \{ \langle w(i) - 1 - n(p), m(p), \langle x(p), y(p), t(p) \rangle \rangle \} \right)$$

The mirror of a v-image is also referred to as the *inversion* of the v-image below.

□

Definition 3.4.8 (Rotation of a v-image). Given a pre-defined array of angles in radians $A = [0, \frac{3}{2}\pi, \pi]$ and $a \in [0..2]$, an index into A , the function $rotate \in \mathcal{P}(\mathbb{P}) \rightarrow \mathcal{P}(\mathbb{P})$ rotates a v-image i by $A[a]$ radians, satisfying:

$$rotate(i, a) = \left(\bigcup p : p \in i : \{ \langle rx(i, n(p), a), ry(i, m(p), a), \langle x(p), y(p), t(p) \rangle \rangle \} \right)$$

where rx

$$rx(i, x, a) = (x - w(i)/2) \times \cos(A[a]) - (y - h(i)/2) \times \sin(A[a]) + w(i)/2$$

and

$$ry(i, y, a) = (x - w(i)/2) \times \sin(A[a]) + (y - h(i)/2) \times \cos(A[a]) + w(i)/2$$

The angles in A represent three consecutive clockwise rotations of a v-image whose initial angle is assumed to be $\frac{1}{2}\pi$.

□

3.4.4 Functions related to matching on v-images

Definition 3.4.9 (Verifying a match of one v-image in another v-image). Given a v-image a , a target image b and a location $\langle n, m \rangle$ in the 2D matrix of b , $matchat \in \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$ is a function that verifies that the set of material types of every pixel $a[j, k]$ of a is equal to the set of material types of the corresponding target pixel $b[n + j, m + k]$ of b . Formally:

$$matchat(a, b, n, m) = (\forall j, k : j \in [0, \mathbf{w}(a)] \wedge k \in [0, \mathbf{h}(a)] : t(a[j, k]) = t(b[n + j, m + k]))$$

□

Definition 3.4.10 (Locations of matches of one v-image in another v-image). Given a v-image a and a target image b , $matches \in \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \rightarrow \mathbb{N} \times \mathbb{N}$ is a function that gives set set of pairs M such that for every $\langle a, \langle n, m \rangle \rangle \in M$, a matches b at location $\langle n, m \rangle$ in b . Formally:

$$matches(a, b) = (\bigcup n, m : matchat(a, b, n, m) : \{\langle a, \langle n, m \rangle \rangle\})$$

□

3.5 Concept lattices of v-sets for EDA pattern matching

3.5.1 High level algorithm to generate a concept lattice from a set of design rules

The hypothesis that the approach proposed in this chapter is investigating, is that a concept lattice derived from v-images derived from a set of *design rules* can be used as a supporting data structure for matching of all the respective design rules in a target layout.

Algorithm 1 derives the elements of a formal context $\mathbb{K} = \langle G, M, I \rangle$ from a set of design rules. See the definition of a formal context in Definition 2.3.7.

Definition 3.5.1 (String format function). Algorithm 1 uses a function **format** that takes a string as input and returns a string. The input string may contain expansion expressions. Every expansion expression starts with the symbol $\{$ and ends with the symbol $\}$. **format** substitutes every expansion expression with an appropriate string representation of the respective expression enclosed by $\{$ and $\}$. For example if a variable $i = 3$ exists in the scope that **format** is called in, then **format**(“he has $\{i\}$ cars”) will return the string “he has 3 cars”. \square

Algorithm 1.

{ *A function that generates a string of material types from a v-image* }

```

func mstr( $i$ )  $\rightarrow$ 
   $m := ""$ 
  for  $y \in [0..h(i)] \rightarrow$ 
     $r := ""$ 
    for  $x \in [0..w(i)] \rightarrow$ 
       $T := t(i[x, y])$ 
      select  $t' \in T$ 
       $u := \mathbf{format}(\{\mathit{t}'\})$ 
       $T := T \setminus \{t'\}$ 
      do  $T \neq \emptyset \rightarrow$ 
        select  $t' \in T$ 
         $u := \mathbf{format}(\{u\}.\{t'\})$ 
         $T := T \setminus \{t'\}$ 
      od
       $r := \mathbf{format}(\{r\}, \{u\})$ 
    rof
     $m := \mathbf{format}(\{m\} \setminus \{r\})$ 
  rof
  return  $m$ 
cnuf

```

{ $\langle w, h \rangle$ is assumed to be the desired attribute size }


```

{ D is assumed to be a set of v-sets that represents the set of }
{ input design rules }
G, M, I := ∅, ∅, ∅
{ Generate the set of objects from S }
for d ∈ D →
  i := img(d)
  { Process i, the v-image of d and the mirror of i }
  for i' ∈ {i, mirror(i)} →
    { Get all rotations of i' }
    G := G ∪ {i'}
    for a ∈ [0..2] →
      G := G ∪ {rotate(i', a)}
    rof
  rof
rof
{ Generate the attributes and the incidence relation from the objects }
for g ∈ G →
  Mg := ∅
  for l ∈ [0..w(i) - w] →
    for t ∈ [0..h(i) - h] →
      ms := mstr(isub(g, l, t, w, h))
      Mg := Mg ∪ {format("c = {l}, r = {t} \ {ms}")}
    rof
  rof
  { Mg now contains the attributes for object g }
  { Update the set of attributes and add the incidence relation entry }
  M := M ∪ Mg
  I := I ∪ {⟨g, Mg⟩}
rof
{ Result is stored in  $\mathfrak{D}$  }
 $\mathfrak{D}$  :=  $\mathfrak{B}$ (⟨G, M, I⟩)

```

As an example of an attribute $m \in M$ derived by Algorithm 1, consider the (sub) v-image represented by “shaded” section of the v-image in Figure 3.5.1

whose top left hand corner is in position $\langle 0, 0 \rangle$, ie $c = 0$ and $r = 0$. The shading includes a grey rectangle on the top left of the image. This would have been a white rectangle in the original image. The shading also includes an inverted L-shaped in purple that is part of the blue cross.

This (sub) v-image consists of six pixels in the image plane, three of them referenced in the first row of the corresponding 2D pixel matrix, and the next three in the second row. The pixel referenced in row 0, column 0 of the 2D matrix is the white rectangle (shaded to grey). The 0 entry in its top left hand corner indicates the material index of the pixel. Alongside it are two blue (shaded to purple) pixels whose top left hand corners each contain a 1 to indicate their respective material index. In the second row are three adjacent blue pixels (shaded to purple) each of whose top left hand corners also contain a 1 to indicate their respective material index.

Thus 0, 1, 1 indicates the material indices associated with the three pixels in row 1 and 1, 1, 1, the material indices associated with the three pixels in row 2. The attribute of this (sub) v-image is therefore $c = 0, r = 0 \setminus 0, 1, 1 \setminus 1, 1, 1$.

If a pixel is associated with more than one non-zero material index, then this attribute representation would change accordingly. For example, suppose the last pixel in the last row of the example was associated not only with the material index 1, but with material indices $\{1, 2, 3\}$. Then its material description in the attribute pattern would have appeared as 1.2.3 and the attribute would be $c = 0, r = 0 \setminus 0, 1, 1 \setminus 1, 1, 1.2.3$.

Figure 3.5.2 shows a section of a formal context derived from an experimental set of v-images.

3.5.2 Matching design rules on a layout

Definition 3.5.2 (Matching requirement). Given a layout represented by the v-set l and a set of design rules represented by the set of v-sets D , a matching algorithm will first derive the v-image i_l and a set of v-images I_D , such that

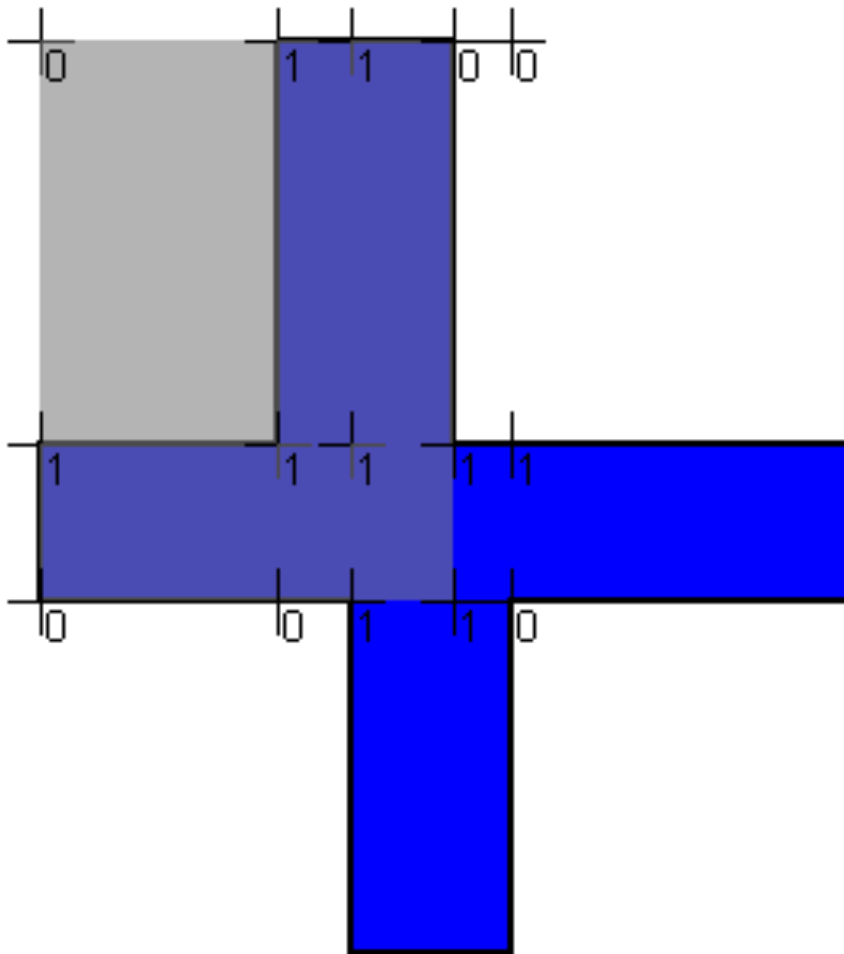


Figure 3.5.1: An attribute depicted by the shaded area of a v-image

3.5 Concept lattices of v-sets for EDA pattern matching

| | c=0,r=0\0.2,2 | c=0,r=1\0.0,0 | c=0,r=2\1.1,1 | c=0,r=3\0.0,0 | c=0,r=4\0.0,2 | c=1,r=0\2.2,2 | c=1,r=1\0.0,2 | c=1,r=2\1.1,1,2 | c=1 |
|------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|-----------------|-----|
| ▶ 115.I0R0 | X | X | X | X | X | X | X | X | X |
| 115.I0R1 | | | | | | | | | |
| 115.I0R2 | X | | | | | X | | | |
| 115.I0R3 | | | | | | | | | |
| 115.I1R0 | X | | | | | X | | | |
| 115.I1R1 | | | | | | | | | |
| 115.I1R2 | | X | X | X | | | X | X | X |
| 115.I1R3 | | | | | | | | | |
| TPC.I0R0 | | | | | | | | | |
| TPC.I0R1 | | | X | | | | | | |
| TPC.I0R2 | | | | | | | | | |
| TPC.I0R3 | | | X | | | | | | |
| TPC.I1R0 | | | | | | | | | |
| TPC.I1R1 | | | X | | | | | | |
| TPC.I1R2 | | | | | | | | | |
| TPC.I1R3 | | | X | | | | | | |
| ELD.V10R0 | | X | | | | | | | |
| ELD.V10R1 | | X | | | | | | | |
| ELD.V10R2 | | | | X | | | | | |
| ELD.V10R3 | | X | | | | | | | |
| ELD.V11R0 | | X | | | | | | | |
| ELD.V11R1 | | X | | | | | | | |
| ELD.V11R2 | | | | X | | | | | |
| ELD.V11R3 | | X | | | | | | | |
| 77A.V10R0 | | | | | | | | | |
| 77A.V10R1 | | | | X | | | | | |
| 77A.V10R2 | | | X | | | | | | |
| 77A.V10R3 | | | | X | | | | | |
| 77A.V11R0 | | | | | | | | | |
| 77A.V11R1 | | | | X | | | | | |
| 77A.V11R2 | | | X | | | | | | |
| 77A.V11R3 | | | | X | | | | | |

Figure 3.5.2: Example of a context created from a set of v-images

$$i_l = \text{img}(l)$$

and

$$I_D = \left(\bigcup d : d \in D : \{\text{img}(d)\} \right)$$

and then derive the output set O , such that

$$O = \left(\bigcup i_d : i_d \in I_D : \text{matches}(i, i_d) \right)$$

□

Definition 3.5.3 (Attribute string to location, v-image conversion). Let \mathbb{M} be all attribute names. Given an attribute name $m \in M$ generated by Algorithm 1, the function $\text{astoi} \in \mathbb{M} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathcal{P}(\mathbb{P})$ gives the location and v-image from which m was derived.

For example if $m = "c = 0, r = 1 \setminus 0, 0, 1 \setminus 0, 1, 0"$ then $\text{astoi}(m) = 0, 1, i$ such that i is the v-image that m was initially created from. The v-image i is depicted by Table 3.1.

| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |

Table 3.1: V-image returned by Function astoi

A formal definition of astoi can be implemented with string splits and substrings and is beyond the scope of this thesis.

□

Algorithm 2 records the matches in compliance with the matching requirement above. The following is a high level outline of the algorithm.

- The inputs to the algorithm are a concept lattice \mathfrak{D} and a target layout, whereby.
 - \mathfrak{D} is the output of Algorithm 1 which, in turn, take as input an attribute size $\langle w, h \rangle$ and a set of v-sets, D , representing the design rules of the problem at hand.

- The target layout is specified as a v-set l
- A target v-image i_l is generated from v-set l .
- Procedure $matchloc(x, y)$ is run for every location $\langle x, y \rangle$ in i_l .
- Procedure $matchloc(x, y)$ has the following outline: \top , the top concept of \mathfrak{D} is examined.
If the intent of the top concept is non-empty ($int(\top) \neq \emptyset$) then the function $matchattrs(int(\top))$ is called. If this function returns **true**, then matches are recorded into the output for every own object of \top . Thereafter $match(\top, x, y)$ is called.
If the intent of the top concept is empty ($int(\top) = \emptyset$) then $match(\top, x, y)$ is called directly.
- Procedure $match(p, x, y)$ records matches into the output for every own object of every child concept $c \in children(p)$ such that the result of calling the function $matchattrs(int(c) \setminus int(p), x, y)$ is **true** and recursively calls itself $match(c, x, y)$ to process c (and its descendants).
- Function $matchattrs(B, x, y)$ verifies that every attribute $b \in B$ matches i_l at location $\langle x, y \rangle$.

Algorithm 2.

```

{ The input layout is represented by the v-set  $l$  }
{ The input set of design rules is represented by the set of v-sets  $D$  }
{ The formal context  $\langle G, M, I \rangle$  and the corresponding concept lattice  $\mathfrak{D}$  is }
{ assumed to have been created using Algorithm 1 from  $D$  and some given }
{ attribute size  $\langle w, h \rangle$  }

```

$i_l, O := img(l), \emptyset$

```

for  $x \in [0..w(l)] \rightarrow$ 
  for  $y \in [0..h(l)] \rightarrow$ 
    { See implementation of function  $matchloc$  below }
     $matchloc(x, y)$ 
  rof
rof

proc  $matchloc(x, y) \rightarrow$ 

```

```

if  $int(\top) \neq \emptyset \rightarrow$ 
  { See implementation of function matchattrs below }
   $m := matchattrs(int(\top), x, y)$ 
  if  $m \rightarrow$ 
    for  $o \in ownobj(\top)$ 
       $O := O \cup \{ \langle o, \langle n, m \rangle \rangle \}$ 
    rof
    { See implementation of function match below }
     $match(\top, x, y)$ 
  ||  $\sim m \rightarrow$  skip
fi
||  $int(\top) = \emptyset \rightarrow$ 
   $match(\top, x, y)$ 
fi
corp

proc  $match(p, x, y) \rightarrow$ 
  for  $c \in children(p) \rightarrow$ 
    { See implementation of function matchattrs below }
     $m := matchattrs(int(c) \setminus int(p), x, y)$ 
    if  $m \rightarrow$ 
      for  $o \in ownobj(c)$ 
         $O := O \cup \{ \langle o, \langle n, m \rangle \rangle \}$ 
      rof
       $match(c, x, y)$ 
    ||  $\sim m \rightarrow$  skip
    fi
  rof
corp

func  $matchattrs(B, x, y) \rightarrow$ 
   $result := \mathbf{true}$ 
   $B' := B$ 
  do  $result \wedge B' \neq \emptyset \rightarrow$ 
    select  $b \in B'$ 
    { See definition of astoi in Definition 3.5.3 }
     $l, t, i_b := astoi(b)$ 

```

```

    { See definition of matchat in Definition 3.4.9 }
    result := matchat(ib, il, x + l, y + t)
    if result →
        B' := B' \ {b}
        || ~ result → skip
    fi
od
return result
cnuf

```

3.6 Experimental Tools and Initial Results

The above sections provide the formal basis and algorithms for matching multiple design rules on a target layout. Sections below describe experimental tools that were developed for the automatic creation of a formal context for a given set of design rules as well as an illustration of the matching of a set of design rules against a target layout using Algorithm 2.

3.6.1 Experimental Software Tools

Two experimental tools have been developed. The first tool is a “*Design Rule Editor*” that provides the following features:

- A graphical v-set editor — to create rectilinear polygons that form part of a microchip design rule with the ability to assign material identifiers to polygons.
- A v-image creation module that implements Function *img* defined above.
- A feature that creates orthogonal rotations and the inversion of a v-image for a v-set.
- A module that uses an attribute size as input to create all attributes for a v-image.
- A module to store all inversion / rotation variations and attributes cre-

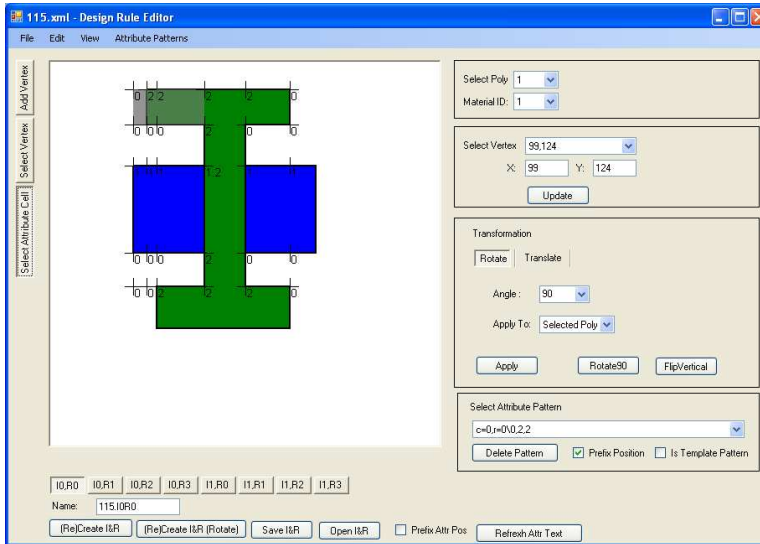


Figure 3.6.1: Main user interface of the Design Rule Editor.

ated during a session of usage of the “Design Rule Editor” in XML format.

Figure 3.6.1 shows a screen capture of the main user interface of the “Design Rule Editor”.

The second tool that was developed is a “Lattice Context Editor” and provides the following features:

- A rule editor file import mechanism that reads a file created with the “Design Rule Editor” into memory.
- A facility to save the context in XML format.
- A facility to export the context into the format accepted by the open source tool called “Concept Explorer”¹.

Figure 3.6.2 shows a screen capture of the main user interface of the “Lattice Context Editor”.

¹Note: Concept Explorer’s author requests that users cite his Russian text, [Yev00], as a reference to the package.

The screenshot shows the 'context.xml - Lattice Context Editor' window. It features a menu bar with 'File' and 'Add Patterns'. Below the menu is a grid with columns representing different context patterns: $c=0,r=0\setminus 1,1,1$, $c=0,r=1\setminus 1,1,0$, $c=0,r=2\setminus 1,0,0$, $c=0,r=3\setminus 1,0,0$, $c=0,r=4\setminus 0,0,0$, and $c=1,r=0\setminus 1,1,0$. The rows list various design rules, including 77A.V0I0R0 through 77A.V0I1R3, TPC.I0R0 through TPC.I1R3, and ELD.V1I0R0 through ELD.V1I1R0. 'X' marks indicate the presence of a pattern in a specific cell.

| | $c=0,r=0\setminus 1,1,1$ | $c=0,r=1\setminus 1,1,0$ | $c=0,r=2\setminus 1,0,0$ | $c=0,r=3\setminus 1,0,0$ | $c=0,r=4\setminus 0,0,0$ | $c=1,r=0\setminus 1,1,0$ |
|------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| 77A.V0I0R0 | X | X | X | X | X | X |
| 77A.V0I0R1 | | | | | X | |
| 77A.V0I0R2 | | | | | | |
| 77A.V0I0R3 | X | | X | | | |
| 77A.V0I1R0 | X | X | | X | | X |
| 77A.V0I1R1 | X | | | | X | |
| 77A.V0I1R2 | | | X | | | |
| 77A.V0I1R3 | X | X | X | | | |
| TPC.I0R0 | | | | | | X |
| TPC.I0R1 | | | | | | X |
| TPC.I0R2 | | | | | | |
| TPC.I0R3 | | | | | | |
| TPC.I1R0 | | | | | | |
| TPC.I1R1 | | X | | | | |
| TPC.I1R2 | | | | | | |
| TPC.I1R3 | | X | | | | |
| ELD.V1I0R0 | | | X | | | |
| ELD.V1I0R1 | | | | | | X |
| ELD.V1I0R2 | | | | | | |
| ELD.V1I0R3 | | | | | | |
| ELD.V1I1R0 | | | | | | |

Figure 3.6.2: Main user interface of the Lattice Context Editor.

3.6.2 Initial Results

3.6.2.1 Experimental Data

The results achieved using the experimental tools and algorithms mentioned above are discussed here in relation to a specific example data set. Using the *Design Rule Editor* the v-sets associated with some existing chip design rules as defined by engineers in the industry were created. Figure 3.6.3 shows these v-sets .

The top-left cell of Figure 3.6.4 shows the v-image created from a v-set representing a design rule and orthogonal rotations of the respective v-image in the next three cells in the top row. The bottom-left cell of Figure 3.6.4 shows the inversion of the v-image in the top-left cell and the next three cells of the

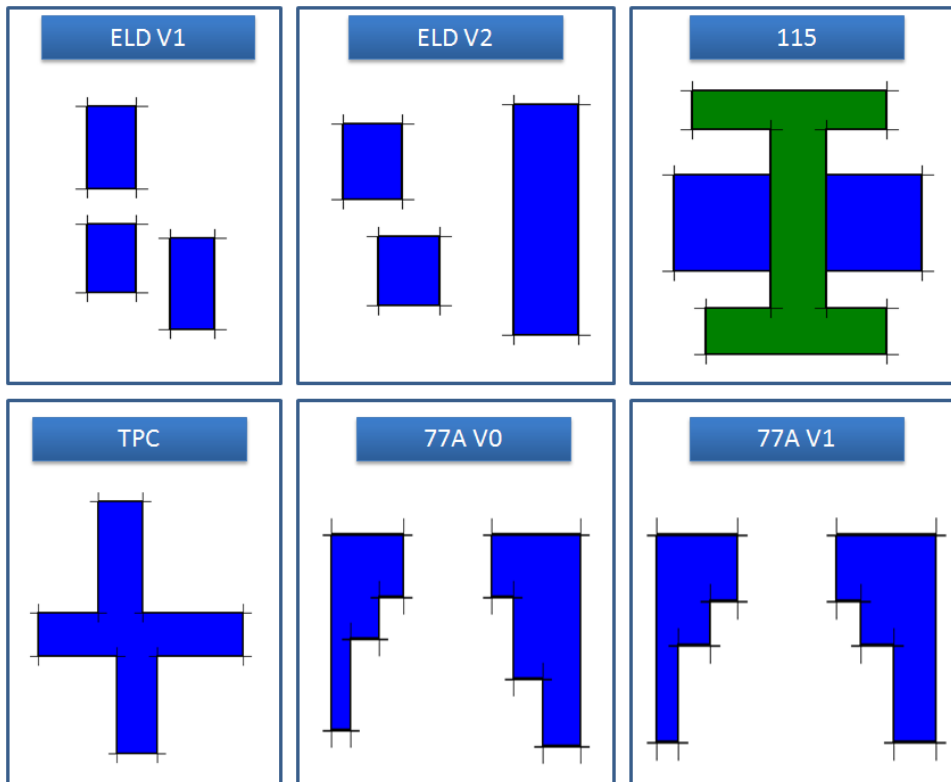


Figure 3.6.3: v-sets of chip design rules.

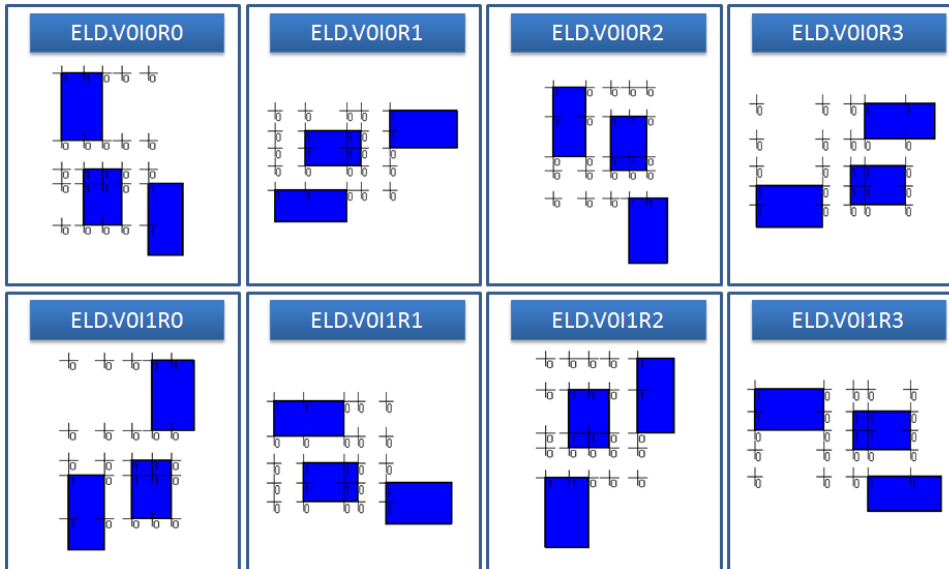


Figure 3.6.4: Rotations and Inversions of a v-image

bottom row show orthogonal rotations of v-image in bottom-left cell.

Figure 3.6.2 shows a section of the *formal context* derived from this data set.

A section of concept lattice line diagram associated with this context is shown in Figure 3.6.5.

3.6.2.2 Illustration of the matching process

To illustrate how the matching algorithm works, the concept lattice whose line diagram is partially shown in Figure 3.6.5 will be used. The algorithm will find all objects of the concept lattice depicted in Figure 3.6.5 that match at any location in the target v-image i_i represented in Table 3.2 (Recall that these objects are v-images.)

Table 3.2 shows the image plane — the 12×9 matrix — representing the

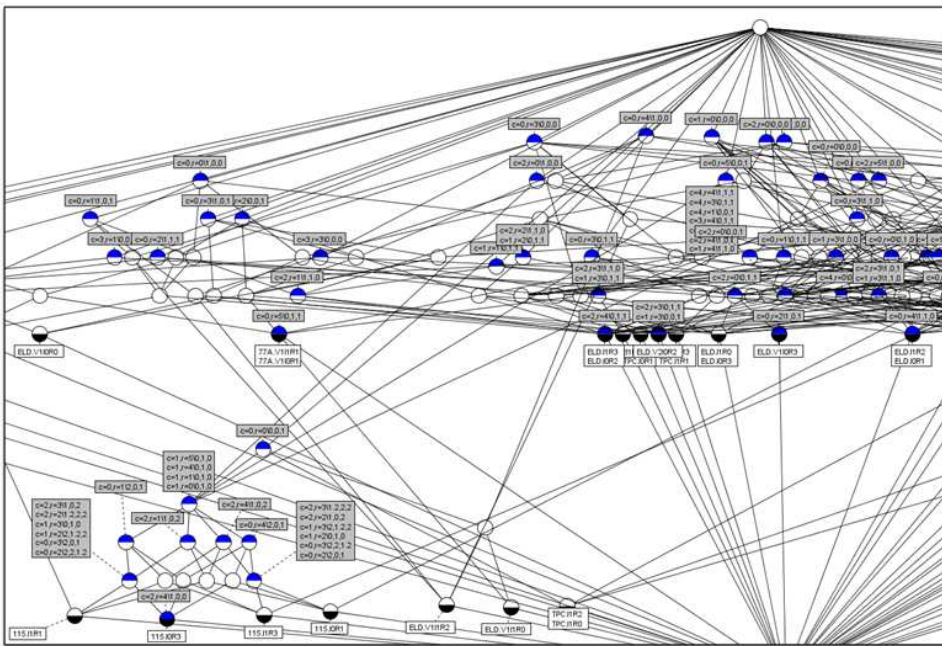


Figure 3.6.5: Lattice Diagram of Design Rules

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----------|---|----------|----------|----------|---|---|---|---|---|-----|----|----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | 0 |
| 4 | 0 | 2 | 0 | 1 | 0 | 2 | 0 | 2 | 0 | 1 | 0 | 2 |
| 5 | 0 | 2 | 2 | 1.2 | 2 | 2 | 0 | 2 | 2 | 1.2 | 1 | 2 |
| 6 | 0 | 2 | 0 | 1 | 0 | 2 | 0 | 2 | 0 | 1 | 0 | 2 |
| 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 8 | 1 | 1 | 1 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 0 | 0 |

Table 3.2: Target v-image to match

target v-image i_l . The columns of the image plane are numbered 0 to 11 and its rows are numbered 0 to 8. The entries 0, 1 or 2 in the image plane represent the materials of corresponding pixels. Some pixels are associated with empty space (0), some with only one material (1 or 2), and some with two materials (1,2). The heading of column 1, the heading of row 2 and the material types of $i_l[1, 2]$, $i_l[2, 2]$ and $i_l[3, 2]$ are shown in bold font to highlight the first attribute that matches against i_l at location $\langle 1, 2 \rangle$ when using Algorithm 2 to match with the concept lattice shown in Figure 3.6.5 against i_l .

Let us assume that traversal through the v-image i_l depicted in Table 3.2 is such that $x = 1$ and $y = 2$, where x and y are the variables referenced in the nested for loops of Algorithm 2. This location is highlighted in Table 3.2 by bold font column and row headers. The algorithm then makes the call $matchloc(1, 2)$. As $int(\top) = \emptyset$ for this concept lattice, Procedure $matchloc$ then calls $match(\top, 1, 2)$. The first child of \top whose intent difference with respect to \top matches in i_l is marked by a black circle in Figure 3.6.6. The box alongside shows that it has a single attribute that is shared by 8 objects (in its extent). The attribute references three pixels in a single row whose materials are 0, 0 and 1 respectively. This matching occurrence is highlighted by material types in bold font for pixels $i_l[1, 2]$, $i_l[2, 2]$ and $i_l[3, 2]$ in Table 3.2.

The recursive traversal (due to Procedure $match$) of the concept lattice through the descendants of the marked concept in Figure 3.6.6 is shown in Figure 3.6.7

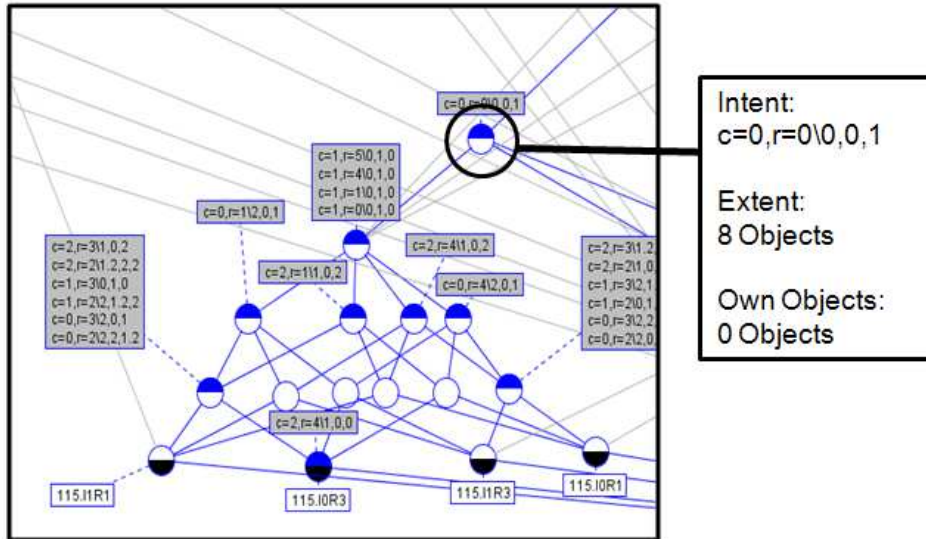


Figure 3.6.6: First matching concept

as a dark circle around every respective *visited concept*. Every visited concept is also annotated by a text box with a dark border that contains the visited concept's intent and the size of its set of own objects. The attributes in the shown intent that is the difference between the intent of the visited concept and the intent of its parent are shown in bold font. These are the attributes being matched in i_l using to Function *matchattrs*. The text box annotation of every visited concept is also labeled by a number in a circle in the range $[0..5]$. These numbers depict the order of visitation of the respective concepts. Every concept that does not match is marked as such by a cross symbol and the descendants of the respective concept that will not be visited are also marked by a cross symbol. As can be seen in Figure 3.6.7, the process reaches the concept c whose text box is labeled by number 5. All of the attributes shown in bold font matches at the current location $\langle x = 1, y = 2 \rangle$ in i_l and this concept has one own object "115.I1R3". Algorithm 2 thus records $\langle "115.I1R3", \langle 1, 2 \rangle \rangle$ into the output O .

A few promising observations can be made from this example. *Firstly*, after

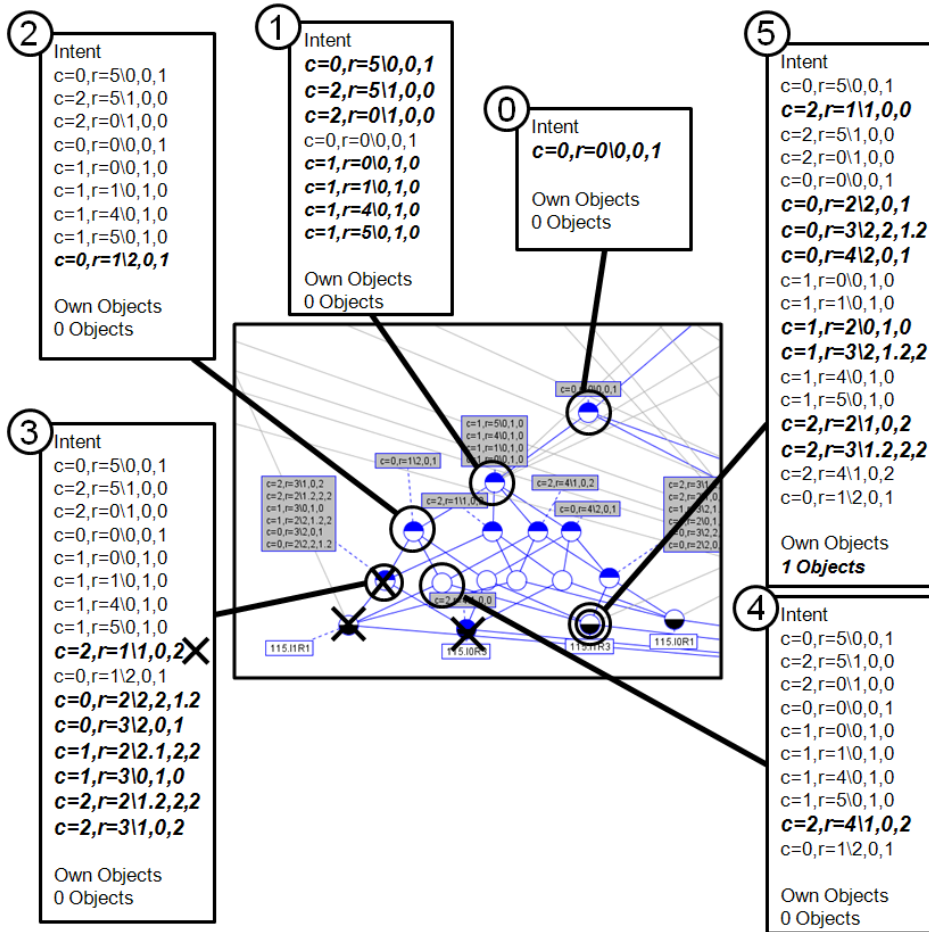


Figure 3.6.7: Matching applied to the descendants of the initial matching concept

finding the first matching concept, the number of attempted attribute matches are very close to the number of attributes of the successfully matched object “115.I1R3” that is the inversion and third rotation of the v-image derived from design rule “115” shown in Figure 3.6.3. This means that very little “unnecessary” matches were done. The optimisation of eliminating from the search space the descendants of concepts that do not match, contributes to the efficient behaviour shown by this approach. *Secondly*, the second concept visited during the matching process (highlighted in *Step 1* in figure 3.6.7) only contains objects in its extent that are variations of the “115” rule. In the microchip design context, the attributes pertaining to this concept could be transformed to a “generic” new design rule that can be searched for in future instead of all the objects in its extent or such a new design rule can be used in future as a replacement of all the rotations and inversions (objects) in its extent, thus simplifying the design rule set.

3.7 Conclusion

This chapter formally introduces v-sets and v-images that can represent EDA design rules and layouts. An algorithm to create a concept lattice from a set of v-images and an algorithm to match all objects that pertain to the respective concept lattice against a target v-image is also introduced. Experimental tools have been developed and used to test the hypothesis that formal concept analysis can be used for very efficient matching of design rules against a layout.

Initial observations on how the search space is traversed suggest that this technique may indeed be more efficient than the naive approach in terms of the number of matching operations at all locations in the target v-image.

Further work may include

- The proposed approach relies on a given attribute size. The impact of attribute size on time and space complexity of the matching process should be investigated.
- Comparison of time and space complexity of this approach to current research in multiple 2D pattern matching techniques.
- More experiments to investigate this technique’s efficiency in handling

non-matching cases

- Analysis of the expected size of Concept Lattices created from v-images derived from chip design rules.
- Implementing the matching algorithm in software to test its execution efficiency.
- Analysis of the characteristics of a data set that makes this approach more appropriate.

Chapter 4

Multiple Keyword Pattern Matching using Position Encoded Pattern Lattices

In this chapter formal concept analysis is used as the basis for two new multiple keyword string pattern matching algorithms. The algorithms addressed are built upon a so-called *position encoded pattern lattice* (PEPL). The algorithms presented are competitive in many ways to the (advanced) Aho-Corasick algorithm. The Aho-Corasick algorithm is heavily used in security applications thanks to its *online* behaviour, easy implementation, and exact/predictable running time analysis which depends only on the input text. The first algorithm to be presented is easily understood and relies directly on the PEPL for matching. Its worst case complexity depends on both the length of the longest keyword, and the length of the search text. Subsequently a finite-automaton-like structure, called an *APEPL Automaton*, is defined which is derived from an *Augmented* version of the PEPL, and which forms the basis for a second more efficient algorithm. In this case, worst case behaviour depends only on the length of the input text. The second algorithm's worst case performance is the same as the *matching phase* of the well-known (advanced) Aho-Corasick multiple-keyword pattern matching algorithm—widely regarded as the multiple keyword pattern matching algorithm of choice in contexts such as network intrusion detection. The first algorithm's performance is comparable to that of the *matching phase* of the lesser-known *failure-function* version

of Aho-Corasick.

4.1 Overview

Several decades of research in *keyword pattern matching* (in which the patterns are finite strings) have yielded many well-known algorithms, such as Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick, and Commentz-Walter. Overview articles are typically more accessible than the original literature—see [CR94, CR03, Smy03] for comprehensive overviews and [Wat95, CWZ10] for taxonomies and correctness proofs of such algorithms. The “advanced” (also known as “optimal”) Aho-Corasick (AC) algorithm [AC75], hence denoted “Advanced AC”, is most popular in applications domains such as network intrusion detection systems (NIDS) and anti-virus (AV) systems (see [Var04] for an overview of algorithms in network implementations), for a few reasons:

- Its worst-case running time complexity is linear in the size of the input stream, and independent of the number of keywords (patterns);
- Its best- and worst-case running time complexities are identical;
- It is *online*, meaning that it does not back-up in the input stream, thereby using a small buffer and a predictable amount of memory—useful properties in hardware implementations.

The key element of Advanced AC that facilitates this performance is the associated AC automaton, which encodes the set of patterns in a way that makes it possible to match against *all* the patterns simultaneously and online. (There are two other flavours of AC algorithm, both making use of a smaller partial automaton.)

The (*multiple*) *keyword pattern matching problem* (in which the patterns are finite strings, or ‘keywords’) consists of finding *all occurrences* (including overlapping ones) of the keywords within an *input* string. Typically, the input string is much larger than the set of keywords and the set of keywords are fixed, meaning they can be preprocessed to produce data-structures for later use while processing the input string. We also make these assumptions in this thesis, as this problem variant corresponds to many real-life applications in security, computational biology, etc [Var04].

In Section 4.2, it is shown how FCA can be used to construct a concept lattice from a position encoded set of patterns. Such a lattice is called a position encoded pattern lattice (PEPL). A first algorithm, called *PMatch*, is developed in Section 4.3, which takes such a PEPL together with a text string (stream) to be searched as input and produces the desired match occurrences as output. As an alternative, a so-called *APEPL Automaton* is defined in Section 4.4, based on the information in an *Augmented* PEPL. Section 4.5 defines the transition function of an *APEPL Automaton*, Section 4.6 defines the transition-output mapping of the *APEPL Automaton* and Section 4.7 defines the state-output mapping of an *APEPL Automaton*. A second algorithm given in Section 4.8 uses this automaton and the text string to be searched as input and also produces the desired match occurrences as output. In Section 4.9 we present a refactored version of the *APEPL Automaton algorithm*. The theoretical performance of the algorithms presented in Sections 4.8 and 4.9 are significant improvements on the algorithm derived in Section 4.3 and correspond to that of Advanced AC. In Section 4.10, we reflect on the implications of these results.

4.2 Position Encoded Pattern Lattices (PEPLs)

As given in Definition 2.4.5, The length of string p will be denoted by $|p|$ and its $(i + 1)^{st}$ element by p_i for $i \in [0, |p|)$.

Definition 4.2.1 (Match Occurrence at a Position). A *match occurrence* of pattern $p \in V^+$ in target string $s \in V^+$ at position t is denoted by the predicate $match : V^+ \times V^+ \times \mathbb{N} \rightarrow \mathbb{B}$ is defined as follows:

$$match(p, s, t) = (\forall k : k \in [0, |p|) : p_k = s_{t+k})$$

Informally *match* means that a single pattern p matches in target s at position t , if and only if $\forall k \in [0, |p|), p_k = s_{t+k}$.

□

Definition 4.2.2 (Set of Match Occurrences at a Position). The *set* of patterns P that matches at position t on target s can now be defined using the function $match\ set : \mathcal{P}(V^+) \times V^+ \times \mathbb{N} \rightarrow V^+ \times \mathbb{N}$ defined as follows:

$$\text{matchset}(P, s, t) = \left(\bigcup p : p \in P \wedge \text{match}(p, s, t) : \{\langle p, t \rangle\} \right)$$

□

Definition 4.2.3 (Position encoding of a pattern and a set of patterns). The *position encoding* of string w is the set of position-symbol pairs denoted by \overline{w} and is given by

$$\overline{w} = \left(\bigcup k : k \in [1, |w|] : \{\langle k, w_{k-1} \rangle\} \right)$$

The *position encoding* of a set of strings P is denoted \overline{P} and is given by

$$\overline{P} = \left(\bigcup w : w \in P : \overline{w} \right)$$

□

Example 4.2.4. For example, the position encoding of “pack” is $\overline{\text{pack}} = \{\langle 1, p \rangle, \langle 2, a \rangle, \langle 3, c \rangle, \langle 4, k \rangle\}$, and of “packet” it is $\overline{\text{packet}} = \{\langle 1, p \rangle, \langle 2, a \rangle, \langle 3, c \rangle, \langle 4, k \rangle, \langle 5, e \rangle, \langle 6, t \rangle\}$. In this case, the position encoding of the set of patterns $P = \{\text{pack}, \text{packet}\}$ and of “packet” happens to be the same, i.e. $\overline{P} = \overline{\text{packet}}$.

Definition 4.2.5 (Formal context and concept lattice based on the position encoding of a set of patterns). Given any set of patterns P , we can now constitute a formal context $\mathbb{K}_{\overline{P}}$ along the following lines. Regard the words in the set of patterns as a set of objects. Let the position-symbol pairs of the position encoding of the set of patterns serve as attributes of these objects: a given word has as its attributes all the position-symbol pairs that make up its position-encoding.

This context is defined as $\mathbb{K}_{\overline{P}} = \langle P, \overline{P}, \overline{I} \rangle$, where \overline{I} is the incidence relation between objects and attributes depicted in the cross table. The formal concept lattice to be derived from such a context will be called a *Position Encoded Pattern Lattice* (PEPL), denoted by $\overline{\mathfrak{P}}(\langle P, \overline{P}, \overline{I} \rangle)$ or, more concisely, by $\overline{\mathfrak{P}}$.

□

Example 4.2.6. As an example, consider the set of patterns $P = \{abc, aabc, abcc\}$. Table 4.2.1 shows the cross table that represents the position encoded formal context derived from P . The cover graph of the underlying PEPL is shown in

| $\langle P, \bar{P}, \bar{I} \rangle$ | $\langle 1, a \rangle$ | $\langle 2, a \rangle$ | $\langle 2, b \rangle$ | $\langle 3, b \rangle$ | $\langle 3, c \rangle$ | $\langle 4, c \rangle$ |
|---------------------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| abc | × | | × | | × | |
| aabc | × | × | | × | | |
| abcc | × | | × | | × | × |

Figure 4.2.1: Position encoded context for $P = \{abc, aabc, abcc\}$.

Figure 4.2.2. Note that concepts are marked using the \diamond operator defined in 2.3.17.

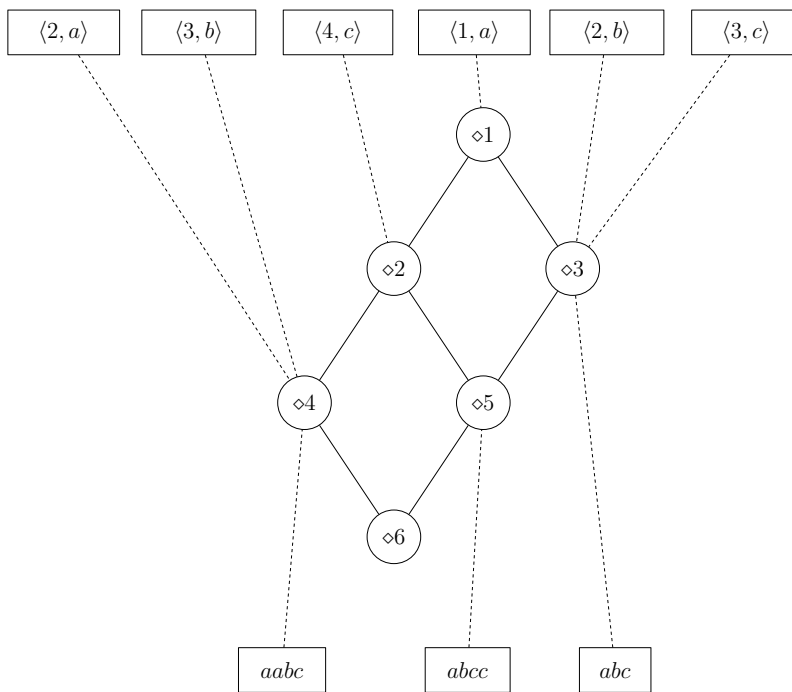


Figure 4.2.2: Cover graph of PEPL for $P = \{abc, aabc, abcc\}$.

Definition 4.2.7 (Checking an attribute positively against a target string (attribute matching)). If a search of text s is currently at position $s[t]$, and it

is found that $s[t + n - 1] = a$, then attribute $\langle n, a \rangle$ is said to *positively check against* s at t .

□

Property 4.2.8 (Property of the intent of a PEPL concept and its own objects as basis for matching). It is evident that the intent of a PEPL concept has the following property as it relates to multiple own objects:

If all the attributes in the intent have been positively checked against a search text s at position t , then the words that are in the set of the own objects of the concept match the text, starting at position t . Therefore, by the definition of the set of own objects of concept c , the following holds for the intent of c and how the result MO is updated for position t :

- $int(c) = \overline{ownobj(c)}$
- $MO := MO \cup ownobj(c) \times \{t\}$

i.e. the intent of c will correspond to the position encoding of the set $\overline{ownobj(c)}$ and the set of pairs formed by $ownobj(c) \times \{t\}$ will be added to the result MO . Note that this applies even in the degenerate case where $ownobj(c) = \emptyset$, for then $ownobj(c) \times \{t\} = \emptyset$.

□

Property 4.2.9 (Property of the intent of a PEPL concept and its single own object as basis for matching). Property 4.2.8 allows for the general case where a concept can have more than one own object¹.

However, in the case of PEPL, each concept has at most one own object w in the *singleton* set of own objects $\{w\}$. In this case, if all the attributes in the intent have been positively checked against a search text s at position t , the following holds for concept c :

- $int(c) = \overline{ownobj(c)} = \bar{w}$
- $MO := MO \cup \{\langle w, t \rangle\}$

i.e. the intent of c will correspond to the position encoding of w and the pair $\langle w, t \rangle$ will be added to the result MO .

¹The general cases arises when a concept lattice is based on a formal context that contains one or more sets of objects that share the same set of attributes.

□

Example 4.2.10. This is clearly the case for concepts $\diamond 3$, $\diamond 4$ and $\diamond 5$ with own objects “abc”, “aabc” and “abcc” respectively of the PEPL in Figure 4.2.2 .

4.3 PEPL-based Matching Using PMatch

Algorithm 3 described below is based on the insights of the previous section. Its top level procedure is called *PMatch*, which takes as input a PEPL $\overline{\mathfrak{P}}(\langle P, \overline{P}, \overline{I} \rangle)$ (or simply $\overline{\mathfrak{P}}$) and a text, s . It then finds in s all match occurrences of words in P , recording them in MO .

The definition of *PMatch* assumes constant $minlength(P)$ as the length of the shortest keyword in P . To avoid notational clutter, $\overline{\mathfrak{P}}$, s and MO are assumed to be globally accessible to all procedures.

PMatch calls *matchIntent* for each character in s where a match could possibly start (i.e. the tail is ignored). The condition in the associated for-loop is intended to signify that these probes are from left to right.

matchIntent takes a string position t , a concept, p and a set of attributes Δ , as parameters. It is assumed that Δ is the set *difference* between the intent of p and the intent of a parent of p in the lattice. A loop checks whether all the attributes in Δ indicate positional matches in the text s at an offset given by the current search position, t —i.e. the loop removes from Δ all attributes of the form $\langle i, \alpha \rangle$ such that $s[t + i - 1] = \alpha$. If this reduces Δ to the empty set, then a match occurrence is considered to have been found for the own object of c .

Moreover if Δ has been reduced to the empty set, then the algorithm recursively invokes *matchIntent* for all of p 's children.

Definition 4.3.1 (Contains Predicate). The post condition of *PMatch* and the function *matchIntent* relies on the predicate $contains(X, Y, t, s)$ that is defined as follows:

$$contains(X, Y, t, s) = (X = matchset(Y, s, t))$$

Informally, *contains* is true if set X contains all match occurrences of a set of strings Y that start at position t in a string s \square

Algorithm 3. *PEPL Based Matching*

```

proc  $PMatch(\overline{\mathfrak{P}}, s)$ 
   $MO, j := \emptyset, minlength(P);$ 
  { Traverse target string s from left to right }
  for  $(t \in [0, |s| - j + 1]) \rightarrow$ 
     $matchIntent(t, \top, int(\top))$ 
  rof
corp
  { post :  $(\forall t : t \in [0, |s| - j + 1] : contains(MO, P, t, s))$ 
    i.e. MO is the set of match occurrences of P in s }

  { pre :  $(\forall \langle i, \alpha \rangle : \langle i, \alpha \rangle \in int(p) \setminus \Delta : (s[t + i - 1] = \alpha))$  }
  proc  $matchIntent(t, p, \Delta)$ 
    do  $(\exists \langle i, \alpha \rangle : \langle i, \alpha \rangle \in \Delta : (s[t + i - 1] = \alpha)) \rightarrow$ 
       $\Delta := \Delta \setminus \{\langle i, \alpha \rangle\}$ 
    od;
    if  $(\Delta = \emptyset) \rightarrow MO := MO \cup ownobj(p) \times \{t\};$ 
      for all  $c \in children(p) \rightarrow$ 
         $matchIntent(t, c, int(c) \setminus int(p))$ 
      rof
     $\parallel (\Delta \neq \emptyset) \rightarrow \mathbf{skip}$ 
  fi
corp
  { post :  $contains(MO, ownobj(p), t, s)$  }

```

Example 4.3.2. To illustrate how Algorithm 3 works, consider the keywords to match $P = \{abc, aabc, abcc\}$ and the target $s = aaabcdabccd$. The formal context $\langle P, \overline{P}, \overline{I} \rangle$ is given in Figure 4.2.1 and the cover graph for the corresponding PEPL, $\overline{\mathfrak{P}}$, is in Figure 4.2.2. For convenience, the intents and own object sets of each concept are made explicit in Table 4.1. Table 4.2 provides a trace summary of calls to *matchIntent*. The first column shows t , the offset into s from which matching positions are calculated. The second column shows

| c | $int(c)$ | $ownobj(c)$ |
|--------------|--|-------------|
| $\diamond 1$ | $\{\langle 1, a \rangle\}$ | |
| $\diamond 2$ | $\{\langle 1, a \rangle, \langle 4, c \rangle\}$ | |
| $\diamond 3$ | $\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle\}$ | abc |
| $\diamond 4$ | $\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle\}$ | aabc |
| $\diamond 5$ | $\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle, \langle 4, c \rangle\}$ | abcc |
| $\diamond 6$ | $\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 2, b \rangle, \langle 3, b \rangle, \langle 3, c \rangle, \langle 4, c \rangle\}$ | |

Table 4.1: Details of concepts of the PEPL in Figure 4.2.2

the lattice concept visited, p . The third column Δ , is the intent difference between p and a corresponding parent of p . The first three columns therefore correspond to the parameters of the function $matchIntent$.

A column per symbol in the string $aaabcdabccd$ then follows.

The last column gives the element m to be added to MO , where m is defined as follows:

$$m = \begin{cases} \langle w, t \rangle & \text{if } \{w\} = ownobj(c) \neq \emptyset \wedge (\forall \langle i, \alpha \rangle : \langle i, \alpha \rangle \in \Delta : s[t + i - 1] = \alpha) \\ nil & \text{otherwise} \end{cases}$$

Note that since $minlength(P) = 3$ and $|s| = 11$, the trace ranges over $t \in [0, 9)$. Each row is a matching step of the algorithm—i.e. every row represents a call of the function $matchIntent$.

As an example, the first row indicates that the matching position $t = 0$, the visited concept $p = \diamond 1$ ² and the attribute set to match is $\Delta = \{\langle 1, a \rangle\}$.

The only element of the set is $\langle 1, a \rangle$, i.e. $i = 1$ and $\alpha = a$. This element “instructs” the algorithm to check position $t + i - 1 = 0$ in s for the symbol $\alpha = a$, which is indeed the case as indicated by the “T” (for the boolean value *true*) shown in the first column of the target string. Every “T” entry in the

²By convention, the concept in a lattice denoted by $\diamond 1$ is the same as \top

| t | p | Δ | a | a | a | b | c | d | a | b | c | c | d | ownobj(p) | m |
|---|--------------|--------------------|---|---|---|---|---|---|---|---|---|---|---|-------------|---------------------------|
| 0 | $\diamond 1$ | $\{(1,a)\}$ | T | | | | | | | | | | | \emptyset | nil |
| 0 | $\diamond 2$ | $\{(4,c)\}$ | | | | F | | | | | | | | nil | nil |
| 0 | $\diamond 3$ | $\{(2,b), (3,c)\}$ | | F | | | | | | | | | | nil | nil |
| 1 | $\diamond 1$ | $\{(1,a)\}$ | | T | | | | | | | | | | \emptyset | nil |
| 1 | $\diamond 2$ | $\{(4,c)\}$ | | | | | T | | | | | | | \emptyset | nil |
| 1 | $\diamond 4$ | $\{(2,a), (3,b)\}$ | | | T | T | | | | | | | | $\{abc\}$ | $\langle abc, 1 \rangle$ |
| 1 | $\diamond 3$ | $\{(2,b), (3,c)\}$ | | | F | | | | | | | | | nil | nil |
| 2 | $\diamond 1$ | $\{(1,a)\}$ | | | T | | | | | | | | | nil | nil |
| 2 | $\diamond 2$ | $\{(4,c)\}$ | | | | | | F | | | | | | nil | nil |
| 2 | $\diamond 3$ | $\{(2,b), (3,c)\}$ | | | | T | T | | | | | | | $\{abc\}$ | $\langle abc, 2 \rangle$ |
| 2 | $\diamond 5$ | $\{(4,c)\}$ | | | | | | F | | | | | | nil | nil |
| 3 | $\diamond 1$ | $\{(1,a)\}$ | | | | F | | | | | | | | nil | nil |
| 4 | $\diamond 1$ | $\{(1,a)\}$ | | | | | F | | | | | | | nil | nil |
| 5 | $\diamond 1$ | $\{(1,a)\}$ | | | | | | F | | | | | | nil | nil |
| 6 | $\diamond 1$ | $\{(1,a)\}$ | | | | | | | T | | | | | \emptyset | nil |
| 6 | $\diamond 2$ | $\{(4,c)\}$ | | | | | | | | | | T | | \emptyset | nil |
| 6 | $\diamond 4$ | $\{(2,a), (3,b)\}$ | | | | | | | | F | | | | nil | nil |
| 6 | $\diamond 5$ | $\{(2,b), (3,c)\}$ | | | | | | | | T | T | | | $\{abcc\}$ | $\langle abcc, 6 \rangle$ |
| 6 | $\diamond 3$ | $\{(2,b), (3,c)\}$ | | | | | | | | T | T | | | $\{abc\}$ | $\langle abc, 6 \rangle$ |
| 7 | $\diamond 1$ | $\{(1,a)\}$ | | | | | | | | F | | | | nil | nil |
| 8 | $\diamond 1$ | $\{(1,a)\}$ | | | | | | | | | F | | | nil | nil |

Table 4.2: Algorithm 3 trace: matching $\{abc, aabc, abcc\}$ in $aaabcdabcccd$

table indicates that an attribute in the set Δ has been successfully matched in the do-loop of *matchIntent*.

Once Δ has been reduced to \emptyset , *MO* has to be updated. Of course, if the concept has no own object—as is the case for the top concept ($\diamond 1$)—then nothing is added to *MO* (i.e. $m = \text{nil}$).

Subsequent calls to *matchIntent* without updating t , recursively deal with child concepts of the one currently under test. The second row of the table therefore logs the results of the call to *matchIntent* in respect of concept $\diamond 2$, the leftmost child of concept $\diamond 1$ in the diagram. In this case, the intent difference set is $\Delta = \{(4,c)\}$, and since $(\nexists \langle i, \alpha \rangle : \{(4,c)\} : (s[t + i - 1] = \alpha))$, (or, more explicitly, $s[0 + 4 - 1] = b$ and not c) *matchIntent* cannot reduce Δ to \emptyset . This is indicated by “F” (for false) as an entry in the relevant column of the table.

As shown in the third row in the table, control now returns to *matchIntent*, where the next child of concept $\diamond 1$, namely concept $\diamond 3$, is considered. Further rows of the table illustrate the execution steps of Algorithm 3 for the rest of the target string.

In every row, the range of positions $[0..t]$ visited in previous iterations is depicted by a longer cell that spans columns $[3..t+3]$ - assuming the table columns are numbered from 0 from left to right. This longer cell precedes the first cell in the column that contains the header $s[t]$, signifying the part of s that has already been processed, i.e. $s[0..t-1]$.

4.3.1 Performance of PMatch

PMatch eliminates sets of words from P that do not match in s without ever backing up in s , i.e. t is monotonically increasing. In this sense *PMatch* is an *online* algorithm, similar to Advanced AC. However, *PMatch* sometimes *revisits* symbols in s . Such revisits are reflected by the multiple entries in various columns representing symbols in *aaabcdabccd* in Table 4.2.

The execution complexity of the matching process *per position checked in s* is bounded by the size of the PEPL. Table 4.2 shows how all concepts are visited when $t = 6$. An (rather conservative) upper bound of the complexity of Algorithm 3 is therefore $(|\overline{\mathfrak{P}}| \times |s|)$. Advanced AC is of course more efficient than this. Not only does it check every symbol in s exactly once; it also avoids the application of the expensive set difference operator that is applied in *matchIntent* of Algorithm 3. Instead, Advanced AC simply makes an automaton transition and considers whether an accepting state has been entered. In the upcoming sections, we refine our algorithm to arrive at a PEPL-based algorithm with similar performance characteristics to Advanced AC.

4.4 APEPL Automata

For PEPL based matching to achieve the same order-of-magnitude performance as Advanced AC, this section defines a structure called an *Augmented PEPL Automaton* or *APEPL Automaton*. This automaton is derived by an algorithm that traverses a concept lattice for a so-called *augmented* language $P^\#$. We derive the formal context for this lattice by extending in a specific way the position encoded formal context for the set of keywords P . This extension is achieved by including additional words (as objects) and their position

encodings (as attributes) in the position encoded formal context for P .

The precise construction of this extended formal context requires some additional refinement which is provided in terms of the formal definitions provided below.

Definition 4.4.1 (Augmentation operator). For two strings p and y we define the operator denoted $\#$ as

$$p\#y = \begin{cases} \{(p)y\} & \text{if } y \neq \varepsilon \wedge p \neq \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

The rationale of this operator will become apparent in definitions below. Note that parentheses are used here to distinguish two strings that are deemed to have the same position encoding. Thus, $(p)y$ and py represent different instances of the same string. They are the same in that their position encoding is identical—the parentheses are ignored for encoding purposes. They are different instances in that they may serve as distinct objects in an APEPL. □

Definition 4.4.2 (Augmentation of a string). We define the prefix-augmentation of string y with respect to string x as

$$\langle x, y \rangle^\# = \left(\bigcup p, r : x = p \cdot y \cdot r : p\#y \right)$$

Thus, $\langle x, y \rangle^\#$ gives $p\#y$, for all valid decompositions of x in the form: $x = p \cdot y \cdot r$. □

Example 4.4.3. As an example, if $x = aab$ and $y = ab$ then the only decomposition of x that is relevant is $a \cdot ab \cdot \varepsilon$ so that

$$\begin{aligned} \langle x, y \rangle^\# &= a\#y \\ &= a\#ab \\ &= \{(a)ab\} \end{aligned}$$

Note that $\langle y, x \rangle^\# = \emptyset$.

Example 4.4.4. As another example, if $x = aaab$ and $y = a$ then the decompositions of x that are relevant are $\{a \cdot a \cdot ab, aa \cdot a \cdot b\}$ so that

$$\begin{aligned}\langle x, y \rangle^\# &= a\#y \cup aa\#y \\ &= a\#a \cup aa\#a \\ &= \{(a)a, (aa)a\}\end{aligned}$$

Definition 4.4.5 (String-augmentation of a language). We define the string-augmentation of language V with respect to string w as follows.

$$\langle V, w \rangle^\# = \left(\bigcup v : v \in V \wedge v \neq w : \langle v, w \rangle^\# \right)$$

□

Example 4.4.6. For example, if $V = \{aaab, ab, ababa\}$ and $w = a$ then

$$\begin{aligned}\langle V, w \rangle^\# &= \langle \{aaab, ab, ababa\}, a \rangle^\# \\ &= \langle aaab, a \rangle^\# \cup \langle ab, a \rangle^\# \cup \langle ababa, a \rangle^\# \\ &= \{a\#a, aa\#a\} \cup \emptyset \cup \{ab\#a, abab\#a\} \\ &= \{(a)a, (aa)a, (ab)a, (abab)a\}\end{aligned}$$

Definition 4.4.7 (Augmentation of a language). For a language P we define the augmentation of the language as

$$P^\# = P \cup \left(\bigcup p : p \in P : \langle P, p \rangle^\# \right)$$

□

Example 4.4.8. Thus, if $P = \{aaab, ab, ababa\}$ then

$$\begin{aligned}P^\# &= P \cup \langle \{ab, ababa\}, aaab \rangle^\# \cup \langle \{aaab, ababa\}, ab \rangle^\# \cup \langle \{aaab, ab\}, ababa \rangle^\# \\ &= P \cup \langle ab, aaab \rangle^\# \cup \langle ababa, aaab \rangle^\# \cup \langle aaab, ab \rangle^\# \cup \langle ababa, ab \rangle^\# \cup \\ &\quad \langle aaab, ababa \rangle^\# \cup \langle ab, ababa \rangle^\# \\ &= P \cup \emptyset \cup \emptyset \cup aa\#ab \cup ab\#ab \cup \emptyset \cup \emptyset \\ &= P \cup \{(aa)ab\} \cup \{(ab)ab\} \\ &= \{aaab, ab, ababa, (aa)ab, (ab)ab\}\end{aligned}$$

| $\mathbb{K}_{P^\#}$ | $\langle 1, a \rangle$ | $\langle 2, a \rangle$ | $\langle 2, b \rangle$ | $\langle 3, c \rangle$ | $\langle 4, c \rangle$ | $\langle 3, b \rangle$ |
|---------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| abc | × | | × | × | | |
| aabc | × | × | | | × | × |
| abcc | × | | × | × | × | |
| (a)abc | × | × | | | × | × |

Table 4.3: Context derived by augmenting $P = \{abc, aabc, abcc\}$

Definition 4.4.9 (Formal context and concept lattice based on the position encoding of an augmented set of patterns). Given set of patterns P , we can constitute a formal context denoted $\mathbb{K}_{P^\#}$ using objects from the augmented set of patterns $P^\#$ and attributes from $P^\#$.

□

Definition 4.4.10 (Augmented Position Encoded Pattern Lattice). Given a formal context $\mathbb{K}_{P^\#}$ the associated formal concept lattice, also called an Augmented Position Encoded Pattern Lattice(APEPL), is denoted by $\overline{\mathfrak{P}^\#}$.

□

Example 4.4.11. As an example, consider again the set of patterns $P = \{abc, aabc, abcc\}$ from which the augmentation $P^\#$ is derived as follows:

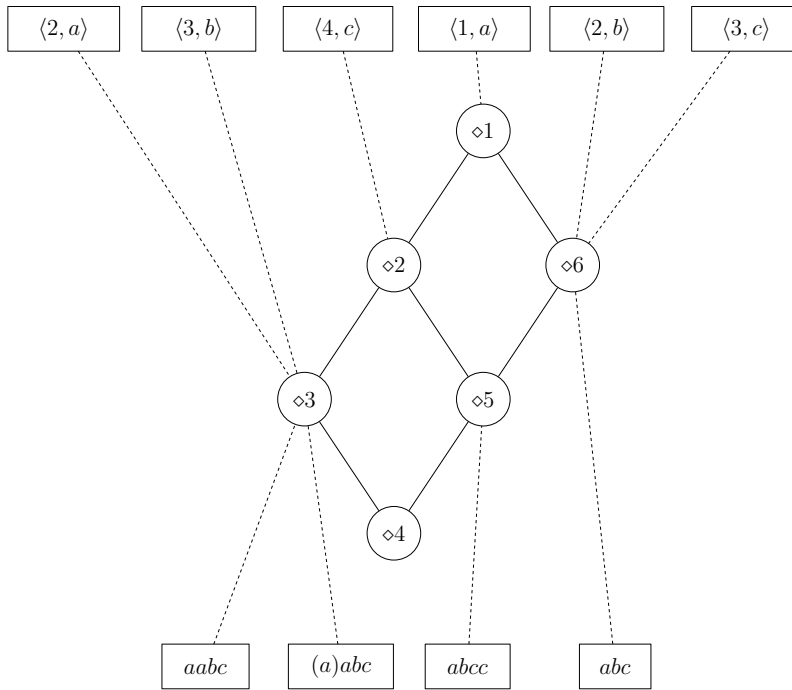
$$\begin{aligned}
P^\# &= P \cup \langle \{aabc, abcc\}, abc \rangle^\# \cup \langle \{abc, abcc\}, aabc \rangle^\# \cup \langle \{abc, aabc\}, abcc \rangle^\# \\
&= P \cup \langle aabc, abc \rangle^\# \cup \langle abcc, abc \rangle^\# \cup \langle abc, aabc \rangle^\# \cup \langle abcc, aabc \rangle^\# \cup \\
&\quad \langle abc, abcc \rangle^\# \cup \langle aabc, abcc \rangle^\# \\
&= P \cup a\#abc \cup \emptyset \cup \emptyset \cup \emptyset \cup \emptyset \cup \emptyset \\
&= P \cup \{(a)abc\} \\
&= \{abc, aabc, abcc, (a)abc\}
\end{aligned}$$

Table 4.3 depicts the context derived from the set of objects $P^\#$ and their corresponding position encoding as attributes. For this example, the Cover Graph for the APEPL for the context of $P^\#$ is shown in Figure 4.4.1.

Algorithm 5 defined in Section 4.8 uses $\overline{\mathfrak{P}^\#}$ to match a set of keywords against a target string. The algorithm relies on a further pre-processing step after $\overline{\mathfrak{P}^\#}$ has been generated from its context. This step creates an automaton, called an “Augmented Position Encoded Pattern Lattice Automaton” or simply an APEPL Automaton.

This automaton defined as follows:

Definition 4.4.12 (APEPL Automaton). Given $\overline{\mathfrak{P}^\#}$, the APEPL that has

Figure 4.4.1: PEPL derived from $P^\#$

been derived from set of patterns $P^\#$, the associated *APEPL automaton* is an eight-tuple $\langle Q, \Sigma, Y, R, \delta, v, \rho, q_0 \rangle$ such that:

- $Q \subseteq \overline{\mathfrak{P}^\#}$ is regarded as the automaton's set of states. It is a subset of the identifiers of all the concepts of the concept lattice $\overline{\mathfrak{P}^\#}$.
- $\Sigma = \overline{P^\#}$ is regarded as the automaton's alphabet, indicating position-symbol pairs.
- $Y = [0..|P^\#_{Max}|]$ is the set of values that a position in a target string can be updated to.
- $R = (\bigcup c : c \in Q : \{ownobj(c)\})$ is a set of sets of match output patterns.
- $\delta : Q \times \Sigma \rightarrow Q$ is the automaton's transition function. This function is

defined in Definition. 4.5.7 in the next section.

- $v : Q \times \Sigma \rightarrow Y$ maps pairs of a state and an input symbol to the corresponding position update value. This function is defined in Definition 4.6.2 below.
- $\rho : Q \times [1..|P^{\#}_{Max}|] \rightarrow R$ maps every state, match count pair to the corresponding match output pattern set. This function is defined in De. 4.7.1 below.
- $q_0 = \top$ is the automaton's start state, which is also the top concept of the APEPL.

□

Property 4.4.13 (APEPL Automaton). It is instructional to view the APEPL Automaton $\langle Q, \Sigma, Y, R, \delta, v, \rho, q_0 \rangle$ as a structure formed by combining the *Mealy Machine* $\langle Q, \Sigma, Y, \delta, v, q_0 \rangle$ and the *Moore Machine* $\langle Q, \Sigma, R, \delta, \rho, q_0 \rangle$.

See Definition 2.4.32 for the definition of a Mealy Machine and Definition 2.4.33 for the definition of a Moore Machine.

□

4.5 The transition function of an APEPL-Automaton

Before we can formally define the transition relation of an PEPL-Automaton we define some preliminary constructs.

Definition 4.5.1 (Position Head Function). Given a set of position-symbol pairs X and a position limit n , the function $poshead : \mathcal{P}(\Sigma) \times \mathbb{N} \rightarrow \mathcal{P}(\Sigma)$ is defined as follows:

$$poshead(X, n) = \left(\bigcup i, \alpha : \langle i, \alpha \rangle \in X \wedge i \leq n : \{ \langle i, \alpha \rangle \} \right)$$

Informally $poshead(X, n)$ gives all elements of X that have a position value less than or equal to n .

□

Notation 4.5.2 (*Nil concept*). To cater for “degenerate situations” where an operator on a concept lattice that returns a concept “fails”, the result will be

the universal “non-concept” identified by the value nil . It is also referred to as the *nil concept*. For example, $inf(\emptyset) = nil$.

Definition 4.5.3 (Attributes Under Test). Given a concept $c \in \overline{\mathfrak{P}^\#}$ and an attribute $\langle i, \alpha \rangle \in \overline{P^\#}$, the *Attributes Under Test* function $aut : \overline{\mathfrak{P}^\#} \times \overline{P^\#} \rightarrow \mathcal{P}(\overline{P^\#})$ is defined as follows:

$$aut(c, \langle i, \alpha \rangle) = poshead(int(c), i - 1) \cup \{\langle i, \alpha \rangle\}$$

aut appends an attribute $\langle i, \alpha \rangle$ to all elements of the intent of a concept c that have a position value less than i .

□

Definition 4.5.4 (Attribute Set Top). We generalize the definition of the *Attribute Top* of a concept lattice, denoted \top (defined in 2.3.13 for a *single* attribute) to also apply to a *set of attributes*. In this case, given the formal concept lattice concept set \mathfrak{L} formed from the formal context $\langle G, M, I \rangle$, the function \top is *overloaded* to have the additional signature : $\top \in \mathcal{P}(M) \rightarrow \mathcal{P}(\mathfrak{L})$ and is defined as follows:

$$\top(X) = \begin{cases} (\bigcup x, c : x \in X \wedge c = \top(x) : \{c\}) & \text{if } X \subseteq M \\ \emptyset & \text{otherwise} \end{cases}$$

Note that by convention, a symbol passed to this function that is in uppercase normally means a set of attributes, while a single attribute will be denoted by a lower case symbol.

□

Definition 4.5.5 (Position Decrement Function). Given a set of position-symbol pairs X , the function $pdec : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ returns the set of position-symbol pairs X' such that every element $\langle i - 1, \alpha \rangle \in X'$ is derived from the element $\langle i, \alpha \rangle \in X$ such that $i > 1$. Formally, $pdec : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ is defined as follows:

$$pdec(X) = (\bigcup i, \alpha : \langle i, \alpha \rangle \in X \wedge i > 1 : \{\langle i - 1, \alpha \rangle\})$$

Informally, the position value of every element in the returned set of pairs of $pdec(X)$ is a decremented version of the position value of the corresponding

element in the input set and any elements that have a position value less than 1 will not be included in the returned set. □

Example 4.5.6. To illustrate the Position Decrement Function, if $X = \{\langle 1, b \rangle, \langle 2, a \rangle, \langle 3, d \rangle, \langle 4, a \rangle\}$, then $pdec(X) = \{\langle 1, a \rangle, \langle 2, d \rangle, \langle 3, a \rangle\}$. Note that the element $\langle 1, b \rangle \in X$ does not have a counterpart in $pdec(X)$.

Definition 4.5.7 (APEPL Transition Function). The transition function δ of an APEPL Automaton $\langle Q, \Sigma, Y, R, \delta, v, \rho, q_0 \rangle$ maps a state $c \in Q$ and a symbol $\langle i, \alpha \rangle \in \Sigma$ to a target state c' .

Formally, $\delta : Q \times \Sigma \rightarrow Q$ is generally determined by the relationship:

$$\delta(c, \langle i, \alpha \rangle) = f(\text{aut}(c, \langle i, \alpha \rangle))$$

where $f : Q \times \Sigma \rightarrow Q$ is generally determined by the recursive relationship:

$$f(F) = \begin{cases} \top & \text{if } F = \emptyset \\ \text{inf}(\top(F)) & \text{if } F \neq \emptyset \wedge \text{inf}(\top(F)) \neq \text{nil} \\ f(pdec(F)) & \text{otherwise} \end{cases}$$

□

Example 4.5.8. As an example of a “normal” transition, consider again the APEPL $\mathfrak{P}^\#$ derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$. Then $\delta(\diamond 1, \langle 2, a \rangle)$ for this APEPL, yields the following transition derivation:

$$\begin{aligned} & \delta(\diamond 1, \langle 2, a \rangle) \\ = & \quad \{ \text{Definition 4.5.7 of } \delta \} \\ & f(\text{aut}(\diamond 1, \langle 2, a \rangle)) \\ = & \quad \{ 2^{nd} \text{ case of } f \text{ in Definition 4.5.7 applies as} \\ & \quad \text{inf}(\top(\text{aut}(\diamond 1, \langle 2, a \rangle))) \\ & \quad = \text{inf}(\top(\text{poshead}(\text{int}(\diamond 1), 1) \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\text{poshead}(\{\langle 1, a \rangle\}, 1) \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\{\langle 1, a \rangle\} \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\})) \\ & \quad = \text{inf}(\{\top(\langle 1, a \rangle), \top(\langle 2, a \rangle)\}) \end{aligned}$$

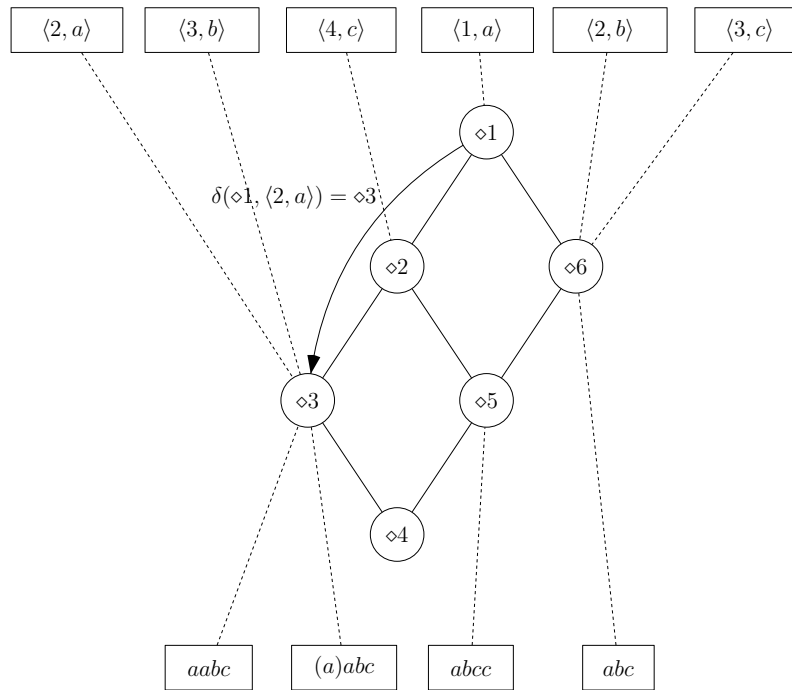


Figure 4.5.1: Transition $\delta(\diamond 1, \langle 2, a \rangle) = \diamond 3$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

$$\begin{aligned}
 &= \overline{\text{inf}(\{\diamond 1, \diamond 3\})} \\
 &= \diamond 3
 \end{aligned}$$

$\diamond 3$

See this transition superimposed over the cover graph of the PEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.5.1.

Example 4.5.9. As a first example of a “failure” transition, consider again the PEPL $\overline{\mathfrak{P}}^\#$ derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$. Then $\delta(\diamond 3, \langle 5, b \rangle)$ for this PEPL, yields the following transition derivation:

$$\begin{aligned}
 &\delta(\diamond 3, \langle 5, b \rangle) \\
 = &\quad \{ \text{Definition of } \delta \text{ in Definition 4.5.7.} \} \\
 &\mathfrak{f}(\text{aut}(\diamond 3, \langle 5, b \rangle)) \\
 = &\quad \{ 3^{\text{rd}} \text{ case of } \mathfrak{f} \text{ in Definition 4.5.7 applies as}
 \end{aligned}$$

$$\begin{aligned}
& \text{inf}(\top(\text{aut}(\diamond 3, \langle 5, b \rangle))) \\
&= \text{inf}(\top(\text{poshead}(\text{int}(\diamond 3), 4) \cup \{\langle 5, b \rangle\})) \\
&= \text{inf}(\top(\text{poshead}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle\}, 4) \cup \{\langle 5, b \rangle\})) \\
&= \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle, \langle 5, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \\
& \text{f}(\text{pdec}(\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle, \langle 4, b \rangle, \langle 5, b \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \text{f}(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle, \langle 4, b \rangle\}) \\
= & \quad \{ 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad \text{inf}(\top(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle, \langle 4, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{f}(\text{pdec}(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle, \langle 4, b \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \text{f}(\{\langle 1, c \rangle, \langle 2, b \rangle, \langle 3, b \rangle\}) \\
= & \quad \{ 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad \text{inf}(\top(\{\langle 1, c \rangle, \langle 2, b \rangle, \langle 3, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{f}(\text{pdec}(\{\langle 1, c \rangle, \langle 2, b \rangle, \langle 3, b \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \text{f}(\{\langle 1, b \rangle, \langle 2, b \rangle\}) \\
= & \quad \{ 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad \text{inf}(\top(\{\langle 1, b \rangle, \langle 2, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{f}(\text{pdec}(\{\langle 1, b \rangle, \langle 2, b \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \text{f}(\{\langle 1, b \rangle\}) \\
= & \quad \{ 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad \text{inf}(\top(\langle 1, b \rangle)) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{f}(\text{pdec}(\{\langle 1, b \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{\text{rd}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \text{f}(\{\emptyset\}) \\
= & \quad \{ 1^{\text{st}} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as } F = \emptyset \} \\
& \top
\end{aligned}$$

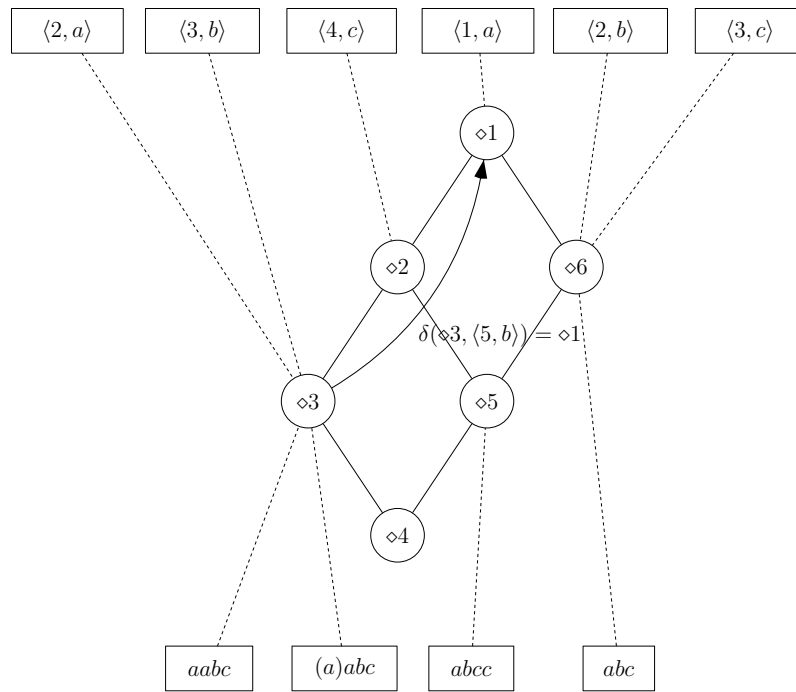


Figure 4.5.2: Transition $\delta(\diamond 3, \langle 5, b \rangle) = \diamond 1$ shown for PEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

$$= \quad \{ \text{Definition of } 3^{rd} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 and Substitution. } \}$$

$$\diamond 1$$

See this transition superimposed over the cover graph of the PEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.5.2.

As a second example of a “failure” transition on the same PEPL, it can be shown using the the following derivation process that $\delta(\diamond 3, \langle 3, a \rangle) = \diamond 3$.

$$\delta(\diamond 3, \langle 3, a \rangle)$$

$$= \quad \{ \text{Definition of } \delta \text{ in Definition 4.5.7. } \}$$

$$\begin{aligned}
& \mathbf{f}(aut(\diamond 3, \langle 3, a \rangle)) \\
= & \quad \{ 3^{rd} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad \mathit{inf}(\top(aut(\diamond 3, \langle 3, a \rangle))) \\
& \quad = \mathit{inf}(\top(\mathit{poshead}(\mathit{int}(\diamond 3), 2) \cup \{\langle 3, a \rangle\})) \\
& \quad = \mathit{inf}(\top(\mathit{poshead}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle\}, 2) \cup \{\langle 3, a \rangle\})) \\
& \quad = \mathit{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\} \cup \{\langle 3, a \rangle\})) \\
& \quad = \mathit{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle\})) = \mathit{inf}(\emptyset) = \mathit{nil} \} \\
& \mathbf{f}(\mathit{pdec}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle\})) \\
= & \quad \{ \text{Definition of } 3^{rd} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7.} \} \\
& \mathbf{f}(\{\langle 1, a \rangle, \langle 2, a \rangle\}) \\
= & \quad \{ 2^{nd} \text{ case of } \mathbf{f} \text{ in Definition 4.5.7 applies as} \\
& \quad = \mathit{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\})) \\
& \quad = \mathit{inf}(\{\top(\langle 1, a \rangle), \top(\langle 2, a \rangle)\}) \\
& \quad = \mathit{inf}(\{\diamond 1, \diamond 3\}) \\
& \quad = \diamond 3 \} \\
& \diamond 3
\end{aligned}$$

See this transition superimposed over the cover graph of the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.5.3.

4.6 The transition-output mapping of an APEPL Automaton

Before defining the transition-output mapping we define the transition-failure-output mapping.

Definition 4.6.1 (APEPL Transition-Failure-Output Mapping). The transition failure-output mapping $\mathit{vf} : \mathcal{P}(\Sigma) \times Y \rightarrow Y$ is defined as follows:

$$\mathit{vf}(F, j) = \begin{cases} j & \text{if } F = \emptyset \\ j + 1 & \text{if } F \neq \emptyset \wedge \mathit{inf}(\top(\mathit{pdec}(F))) \neq \mathit{nil} \\ \mathit{vf}(\mathit{pdec}(F), j + 1) & \text{otherwise} \end{cases}$$

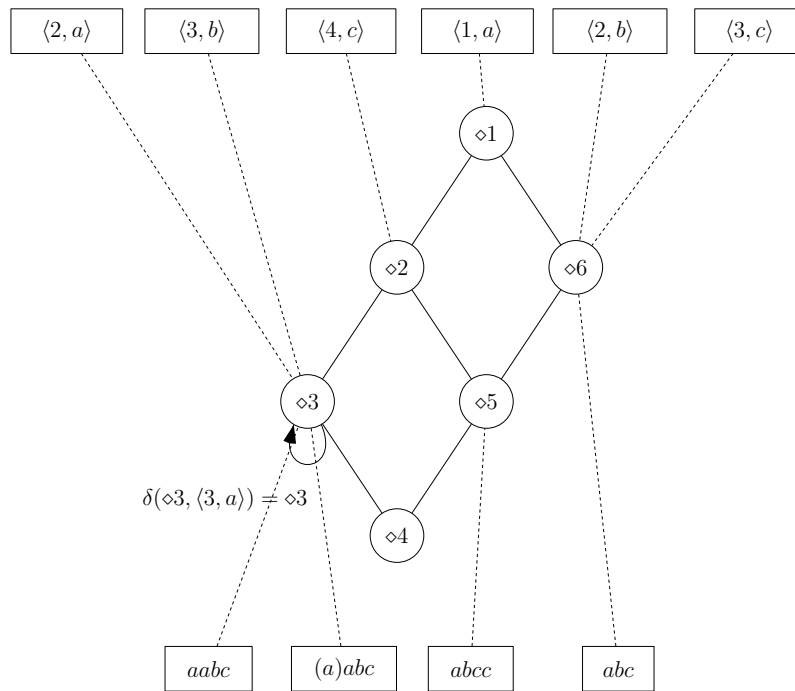


Figure 4.5.3: Transition $\delta(\diamond 3, \langle 3, a \rangle) = \diamond 3$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

□

Definition 4.6.2 (PEPL Transition-Output Mapping). The transition-output mapping $v : Q \times \Sigma \rightarrow Y$ is defined as:

$$v(c, \langle i, \alpha \rangle) = \begin{cases} 0 & \text{if } \text{inf}(\top(\text{aut}(c, \{\langle i, \alpha \rangle\}))) \neq \text{nil} \\ \text{vf}(\text{aut}(c, \langle i, \alpha \rangle), 0) & \text{otherwise} \end{cases}$$

□

Example 4.6.3. As an example of a “normal” transition output, consider again the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ shown in Figure 4.4.1. Then $v(\diamond 1, \langle 2, a \rangle)$ for this APEPL, yields the following transition output derivation:

$$\begin{aligned} & v(\diamond 1, \langle 2, a \rangle) \\ = & \quad \{ 1^{\text{st}} \text{ case of Definition 4.6.2 of } v \text{ applies as} \\ & \quad \text{inf}(\top(\text{aut}(\diamond 1, \langle 2, a \rangle))) \\ & \quad = \text{inf}(\top(\text{poshead}(\text{int}(\diamond 1), 1) \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\text{poshead}(\{\langle 1, a \rangle\}, 1) \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\{\langle 1, a \rangle\} \cup \{\langle 2, a \rangle\})) \\ & \quad = \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\})) \\ & \quad = \text{inf}(\{\top(\langle 1, a \rangle), \top(\langle 2, a \rangle)\}) \\ & \quad = \text{inf}(\{\diamond 1, \diamond 3\}) \\ & \quad = \diamond 3 \} \\ & 0 \end{aligned}$$

The derivation is almost identical to the derivation of the actual transition given in Example 4.5.8 above as the conditions of the cases in the definitions of δ and v respectively are identical.

See this transition output superimposed over the cover graph of the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.6.1.

Example 4.6.4. As a first example of a “failure” transition output, consider again the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$. Then $v(\diamond 3, \langle 5, b \rangle)$ for this APEPL, yields the following transition derivation:

$$\begin{aligned} & v(\diamond 3, \langle 5, b \rangle) \\ = & \quad \{ 2^{\text{nd}} \text{ case of Definition 4.6.2 applies as} \end{aligned}$$

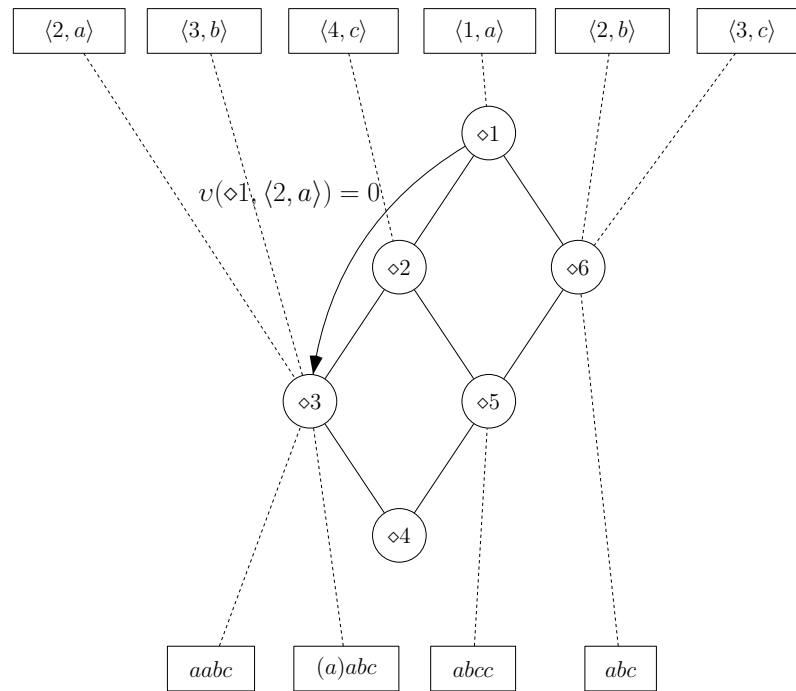


Figure 4.6.1: Transition *output* $v(\diamond 1, \langle 2, a \rangle) = 0$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

$$\begin{aligned}
& \text{inf}(\top(\text{aut}(\diamond 3, \langle 5, b \rangle))) \\
&= \text{inf}(\top(\text{poshead}(\text{int}(\diamond 3), 4) \cup \{\langle 5, b \rangle\})) \\
&= \text{inf}(\top(\text{poshead}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle\}, 4) \cup \{\langle 5, b \rangle\})) \\
&= \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle, \langle 5, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \\
& \text{vf}(\text{aut}(\diamond 3, \langle 5, b \rangle), 0) \\
&= \quad \{ \text{Definition 4.5.3 of } \text{aut.} \} \\
& \text{vf}(\text{poshead}(\text{int}(\diamond 3), 4) \cup \{\langle 5, b \rangle\}, 0) \\
&= \quad \{ \text{Definition 4.5.1 of } \text{poshead.} \} \\
& \text{vf}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle, \langle 5, b \rangle\}, 0) \\
&= \quad \{ 3^{\text{rd}} \text{ case of Definition 4.6.1 applies as} \\
& \quad \text{inf}(\top(\text{pdec}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle, \langle 5, b \rangle\}))) \\
& \quad = \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle, \langle 4, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{vf}(\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle, \langle 4, b \rangle\}, 1) \\
&= \quad \{ 3^{\text{rd}} \text{ case of Definition 4.6.1 applies as} \\
& \quad \text{inf}(\top(\text{pdec}(\{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle, \langle 4, b \rangle\}))) \\
& \quad = \text{inf}(\top(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{vf}(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle\}, 2) \\
&= \quad \{ 3^{\text{rd}} \text{ case of Definition 4.6.1 applies as} \\
& \quad \text{inf}(\top(\text{pdec}(\{\langle 1, b \rangle, \langle 2, c \rangle, \langle 3, b \rangle\}))) \\
& \quad = \text{inf}(\top(\{\langle 1, c \rangle, \langle 2, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{vf}(\{\langle 1, c \rangle, \langle 2, b \rangle\}, 3) \\
&= \quad \{ 3^{\text{rd}} \text{ case of Definition 4.6.1 applies as} \\
& \quad \text{inf}(\top(\text{pdec}(\{\langle 1, c \rangle, \langle 2, b \rangle\}))) = \text{inf}(\top(\{\langle 1, b \rangle\})) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{vf}(\{\langle 1, b \rangle\}, 4) \\
&= \quad \{ 3^{\text{rd}} \text{ case of Definition 4.6.1 applies as} \\
& \quad \text{inf}(\top(\text{pdec}(\{\langle 1, b \rangle\}))) = \text{inf}(\top(\emptyset)) = \text{inf}(\emptyset) = \text{nil} \} \\
& \text{vf}(\emptyset, 5) \\
&= \quad \{ 1^{\text{st}} \text{ case of Definition 4.6.1 applies as } F = \emptyset \}
\end{aligned}$$

5

See this transition output superimposed over the cover graph of the APEPL

4.6 The transition-output mapping of an APEPL Automaton

93

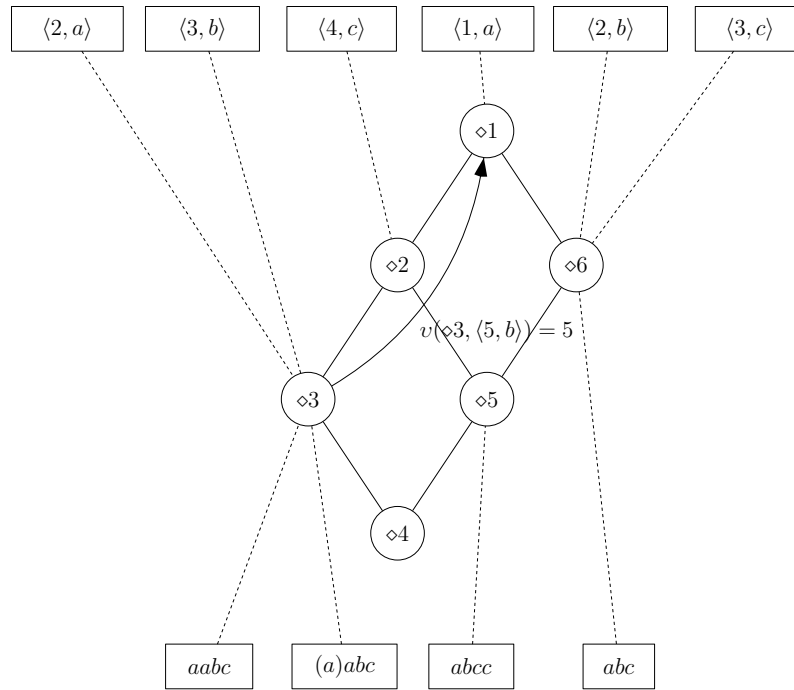


Figure 4.6.2: Transition output $v(\diamond 3, \langle 5, b \rangle) = 5$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.6.2.

As a second example of a “failure” transition output on the same APEPL, it can be shown that $v(\diamond 3, \langle 3, a \rangle) = 1$ as follows:

$$\begin{aligned}
 & v(\diamond 3, \langle 3, a \rangle) \\
 = & \quad \{2^{\text{nd}} \text{ case of Definition 4.6.2 applies as} \\
 & \quad \text{inf}(\top(\text{aut}(\diamond 3, \langle 3, a \rangle))) \\
 & \quad = \text{inf}(\top(\text{poshead}(\text{int}(\diamond 3), 2) \cup \{\langle 3, a \rangle\})) \\
 & \quad = \text{inf}(\top(\text{poshead}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, b \rangle, \langle 4, c \rangle\}, 2) \cup \{\langle 3, a \rangle\})) \\
 & \quad = \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\} \cup \{\langle 3, a \rangle\}))
 \end{aligned}$$

$$\begin{aligned}
&= \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle\})) \\
&= \text{inf}(\{\top(\langle 1, a \rangle), \top(\langle 2, a \rangle), \top(\langle 3, a \rangle)\}) = \text{inf}(\emptyset) = \text{nil} \\
&\text{vf}(\text{aut}(\diamond 3, \langle 3, a \rangle), 0) \\
&= \quad \{ \text{Definition 4.5.3 of } \text{aut.} \} \\
&\text{vf}(\text{poshead}(\text{int}(\diamond 3), 2) \cup \{\langle 3, a \rangle\}, 0) \\
&= \quad \{ \text{Definition 4.5.1 of } \text{poshead.} \} \\
&\text{vf}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle\}, 0) \\
&= \quad \{ 2^{\text{nd}} \text{ case of Definition 4.6.1 applies as} \\
&\quad \text{inf}(\top(\text{pdec}(\{\langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle\}))) \\
&\quad = \text{inf}(\top(\{\langle 1, a \rangle, \langle 2, a \rangle\})) \\
&\quad = \text{inf}(\{\top(\langle 1, a \rangle), \top(\langle 2, a \rangle)\}) \\
&\quad = \text{inf}(\{\diamond 1, \diamond 3\}) \\
&\quad = \diamond 3 \} \\
&1
\end{aligned}$$

See this transition superimposed over the cover graph of the PEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.6.3.

4.7 The state-output mapping of an APEPL Automaton

Definition 4.7.1 (APEPL State-Output Mapping). The state-output mapping relation $\rho : Q \times [1..|P^\#_{Max}|] \rightarrow R$ is defined as:

$$\rho(c, n) = \begin{cases} \text{ownobj}(c) & \text{if } \text{ownobj}(c) \neq \emptyset \wedge n = |\text{int}(c)| \\ \emptyset & \text{otherwise} \end{cases}$$

□

Example 4.7.2. As an example of a state output, consider again the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$. Then $\rho(\diamond 3, 4)$ for this APEPL, yields the following transition output derivation:

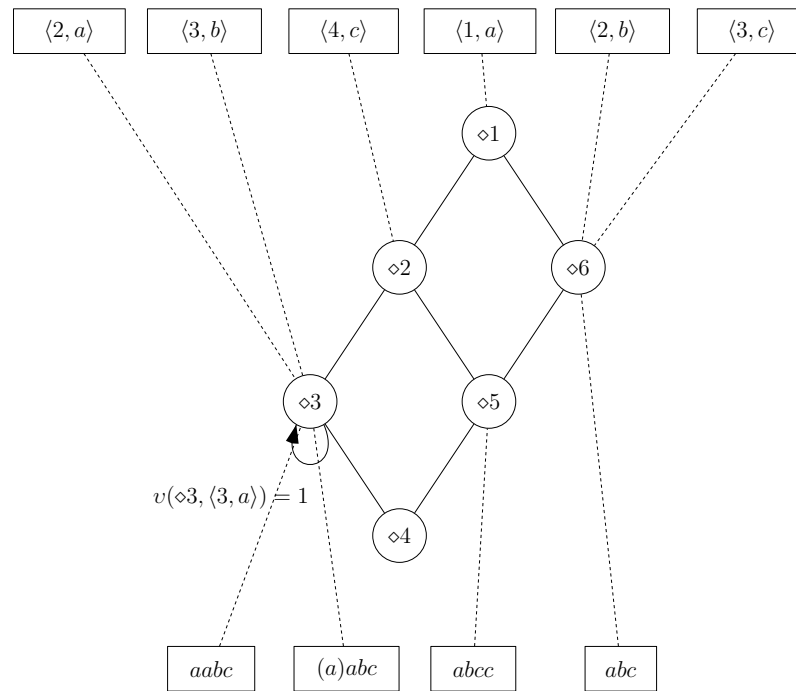


Figure 4.6.3: Transition *output* $v(\diamond 3, \langle 3, a \rangle) = 1$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

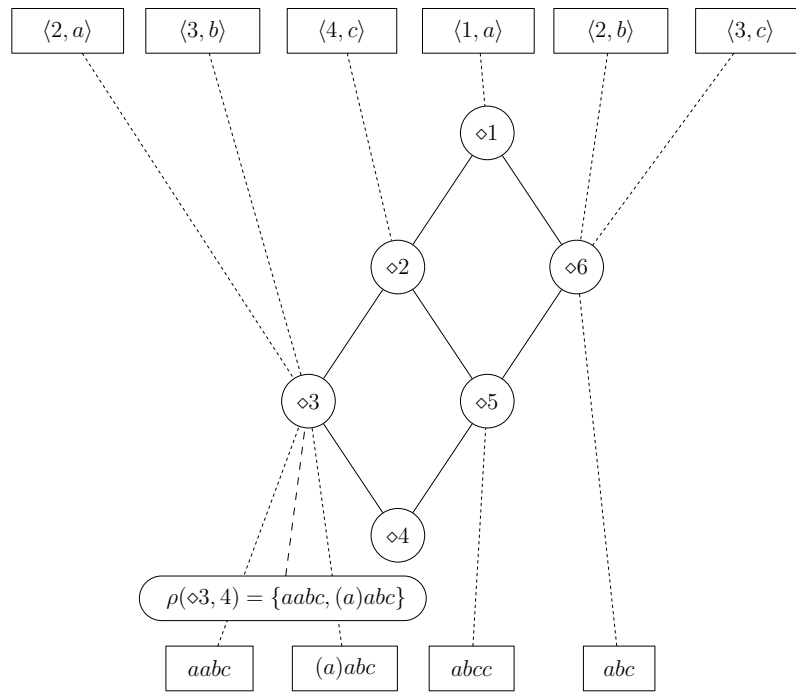


Figure 4.7.1: State *output* $\rho(\diamond 3, 4) = \{aabc, (a)abc\}$ shown for APEPL based on $P^\# = \{abc, aabc, abcc, (a)abc\}$

$$\begin{aligned}
 & \rho(\diamond 3, 4) \\
 = & \quad \{ 1^{\text{st}} \text{ case of Definition 4.7.1 of } v \text{ applies as} \\
 & \quad \text{ownobj}(\diamond 3) \neq \emptyset \wedge 4 = |\text{int}(\diamond 3)| \} \\
 & \text{ownobj}(\diamond 3) \\
 = & \quad \{ \text{Definition 2.3.15 of } \text{ownobj} \} \\
 & \{ aabc, (a)abc \}
 \end{aligned}$$

The output superimposed over the cover graph of the APEPL derived from $P^\# = \{abc, aabc, abcc, (a)abc\}$ in Figure 4.7.1.

Example 4.7.3. For the set of patterns $P^\# = \{abc, aabc, abcc, (a)abc\}$ used in examples above, the *APEPL-Automaton* is (partially) shown in Figure 4.7.2, superimposed over the partial cover graph for $\overline{\mathfrak{P}}^\#$. Note that in order to avoid clutter, the following adjustments were made to the depiction of the automaton:

- The attributes and objects have been omitted from the line diagram of the lattice.
- A number of transitions have been omitted. For example, many of transitions to \top have been left out.
- The labels of transition *relations*, transition *outputs* and state outputs have been made more concise as follows:
 - *Transitions* Where the transition *relation* label for the transition from the state represented by concept c to the state represented by concept c' has the form $\delta(c, \langle i, \alpha \rangle) = c'$ in Figures 4.5.1, 4.5.2 and 4.5.3, and a transition *output* label for the same transition has the form $v(c, \langle i, \alpha \rangle) = x$ in Figures 4.6.2 and 4.6.3, the corresponding transition has the form $\langle i, \alpha \rangle/x$ in this example.
 - *States* Where the oval shaped annotation on a state d that represents a concept that has own objects Z has a label of the form $\rho(d, y) = Z$ in Figure 4.7.1, the annotation of the state d in the figure in this example has the label y on the dotted line connecting the state corresponding to d and the oval shape and the oval shape has the label Z .

4.8 Matching Using an APEPL-Automaton

We show that an APEPL automaton could be used to test whether a given sequence of its alphabet are in the regular set of patterns that it describes.

It is assumed below that a function, *getFA*, is available which delivers an APEPL Automaton when provided with an APEPL. The transition relation and transition output relation of the automaton are assumed to be represented as efficient data structures such as hashed transition tables - i.e. we assume an $\mathcal{O}(1)$ mapping from a state, symbol pair to a state and an $\mathcal{O}(1)$ mapping

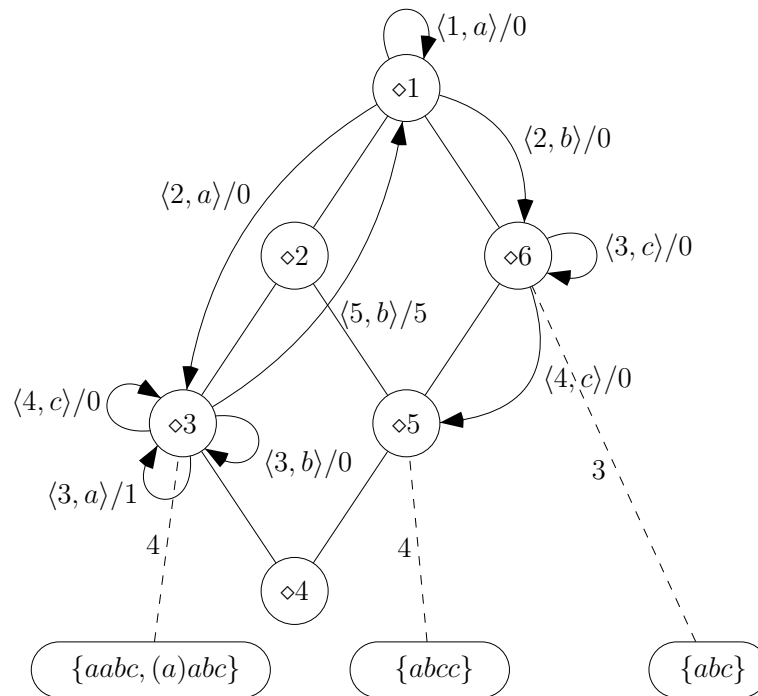


Figure 4.7.2: PEPL Automaton superimposed on a PEPL Cover Graph

from a state, symbol pair to a jump value.

For example, it can easily be seen in Figure 4.7.2 that, starting from state $\diamond 1$, successive transitions on elements of the string $\overline{abc} = \{\langle 1, a \rangle, \langle 2, b \rangle, \langle 3, c \rangle\}$ lead to the state $\diamond 6$ and the result of invoking the state-output relation on state $\diamond 6$, i.e. $\rho(\diamond 6, 3) = \{abc\}$, affirming that this sequence is indeed part of the set of patterns described by the automaton and affirming that abc is in the original

set of patterns, P .

However, the APEPL is not primarily intended to be used in this way. Instead, the APEPL is used in Algorithm 5 to find all match opportunities in P in a text s . The meaning of the variables used in the algorithm are as follows:

- t_0 is the next base index into s . Note that t_0 fulfills the role that t played in Alg. alg:lat-pm-2 above.
- c is the state of the APEPL automaton that is currently being visited. It also corresponds to a concept of the APEPL.
- i is the next position offset (relative to t_0 and t) to check. It also corresponds to the length of the longest prefix already matched
- j is the count by which to adjust t_0 when a mismatch is encountered
- t is the next index of s to be probed

The algorithm's **do** loop processes symbols of s , updating variables c and i to keep track of partial matches in that part of s already processed. This is expressed as loop *invariant* $Inv(c, i, t) \equiv$

- MO contains all matches in $s_{[0, t-i+1]}$. (These are the matches *already processed*.)
- And $s_{[t-i+1, t]}$ matches the first $i - 1$ characters of all patterns in $ext(c)$. (These are the partial matches *in progress*.)

Algorithm 4. *PEPL Automaton Based Matching*

```

proc PAutMatch( $\overline{\mathfrak{P}^\#}$ ,  $s$ )
   $MO := \emptyset$ ;
   $FA := getFA(\overline{\mathfrak{P}^\#})$ ;
   $c, i, t_0, t := \diamond 1, 1, 0, 0$ ;
  { invariant:  $Inv(c, i, t_0)$  }
  do ( $t < |s|$ )  $\rightarrow c, j := \delta(c, \langle i, s[t] \rangle), v(c, \langle i, s[t] \rangle)$ ;
    if ( $\rho(c, i) \neq \emptyset$ )  $\rightarrow MO := MO \cup (\{t\} \times \rho(c, i))$ 
    | ( $\rho(c, i) = \emptyset$ )  $\rightarrow$  skip
    fi;
    if ( $j \neq 0 \wedge c = \diamond 1$ )  $\rightarrow i := 1$ 
    | ( $j = 0 \wedge c \neq \diamond 1$ )  $\rightarrow i := i + 1$ 

```

```

    || (j ≠ 0 ∧ c ≠ ◊1) → skip
    || (j = 0 ∧ c = ◊1) → skip
  fi;
  t0 := t0 + j; { Increment base position in target }
  { by jump value }
  t := t0 + i - 1;
od
corp { post : MO is the set of match occurrences of P in s }

```

To illustrate matching as executed by Algorithm 5, consider the steps logged in Table 4.4 when matching the set of patterns $abc, aabc, abcc$ against the target string $aaabcdabccd$. Each entry (cell) in a data (non-header) row shows the pertinent assignments and conditions (with bound parameters and variables) that constitute execution (of one iteration) of the loop (*do*) statement body in Algorithm 5. The first entry (cell) in this row shows the assignment to c and j , i.e. a transition on the *PEPL Automaton* and the corresponding transition output. The second entry shows the cumulative value of MO , the set of matched occurrences. The last entry shows how the index offset variable i , index base variable t_0 and absolute index variable t is updated before the end of the loop statement body .

As an example, consider the first data row of Table 4.4. This row represents the first iteration of the matching process. The first entry in the row shows a transition is made from the start state ($c = \top = \diamond 1$) to the same state (see loop in transition diagram shown in Figure 4.7.2). The second entry shows that the size of intent of the reached state is equal to 1, which is also equal to the current value of the index offset variable i . However as the third entry shows, no objects are added to the output set MO as the concept $\diamond 1$ has no own objects and therefore according to Definition def:pepl-state-out-mapping the state-output relation yields $\rho(c, i) = \rho(\diamond 1, 1) = \emptyset$. Finally the variable i is incremented, the value of the base index does not change as the value of the jump variable j is zero and the value of the absolute index variable t increments from 0 to 1 accordingly.

Consequent rows show how this process continues until the patterns $aabc, (a)abc$ are recorded when the variables i and t are both (coincidentally) equal to 4 and state $\diamond 3$ is reached in the 5th data row of the table. Recall that according to the definition of ρ , a match is recorded for a state that is represented by a concept such that the concept that has a non-empty set of own objects and

the size of such concept's intent (as shown in the second entry) is the same as the value of variable i .

| $c, j := \delta(c, \langle i, s[t] \rangle (= \alpha)), v(c, \langle i, s[t] \rangle)$ | $MO := MO \cup (\{t\} \times \rho(c, i))$ | $i, t_0, t :=$ |
|--|--|----------------|
| $c, j := \delta(\circ 1, \langle 1, s[0] \rangle (= \mathbf{a})), v(\circ 1, \langle 1, s[0] \rangle) = (\circ 1, \mathbf{0})$ | \emptyset | 2, 0, 1 |
| $c, j := \delta(\circ 1, \langle 2, s[1] \rangle (= \mathbf{a})), v(\circ 1, \langle 2, s[1] \rangle) = (\circ 3, \mathbf{0})$ | \emptyset | 3, 0, 2 |
| $c, j := \delta(\circ 3, \langle 3, s[2] \rangle (= \mathbf{a})), v(\circ 3, \langle 3, s[2] \rangle) = (\circ 3, \mathbf{1})$ | \emptyset | 3, 1, 3 |
| $c, j := \delta(\circ 3, \langle 3, s[3] \rangle (= \mathbf{b})), v(\circ 3, \langle 3, s[3] \rangle) = (\circ 3, \mathbf{0})$ | \emptyset | 4, 1, 4 |
| $c, j := \delta(\circ 3, \langle 4, s[4] \rangle (= \mathbf{c})), v(\circ 3, \langle 4, s[4] \rangle) = (\circ 3, \mathbf{0})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle\}$ | 5, 1, 5 |
| $c, j := \delta(\circ 3, \langle 5, s[5] \rangle (= \mathbf{d})), v(\circ 3, \langle 5, s[5] \rangle) = (\circ 1, \mathbf{5})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle\}$ | 1, 6, 6 |
| $c, j := \delta(\circ 1, \langle 1, s[6] \rangle (= \mathbf{a})), v(\circ 1, \langle 1, s[6] \rangle) = (\circ 1, \mathbf{0})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle\}$ | 2, 6, 7 |
| $c, j := \delta(\circ 1, \langle 2, s[7] \rangle (= \mathbf{b})), v(\circ 1, \langle 2, s[7] \rangle) = (\circ 6, \mathbf{0})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle\}$ | 3, 6, 8 |
| $c, j := \delta(\circ 6, \langle 3, s[8] \rangle (= \mathbf{c})), v(\circ 6, \langle 3, s[8] \rangle) = (\circ 6, \mathbf{0})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle, \langle 8, \{abc\} \rangle\}$ | 4, 6, 9 |
| $c, j := \delta(\circ 6, \langle 4, s[9] \rangle (= \mathbf{c})), v(\circ 6, \langle 4, s[9] \rangle) = (\circ 5, \mathbf{0})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle, \langle 8, \{abc\} \rangle, \langle 9, \{abcc\} \rangle\}$ | 5, 6, 10 |
| $c, j := \delta(\circ 5, \langle 5, s[10] \rangle (= \mathbf{d})), v(\circ 5, \langle 5, s[10] \rangle) = (\circ 1, \mathbf{5})$ | $\{\langle 4, \{aabc, (a)abc\} \rangle, \langle 8, \{abc\} \rangle, \langle 9, \{abcc\} \rangle\}$ | 1, 11, 11 |

Table 4.4: Execution trace showing $|aaabcdabccd| = 10$ positions visited when matching $abc, aabc, abcc$ on $aaabcdabccd$

4.9 Refactored PEPL-Automaton algorithm

It is possible to refactor Alg. 5 to eliminate the need for t_0 , the base offset into the target string as well as the second *if* statement in the body of the main loop. The new algorithm is given below. The loop invariant is formulated more formally and illustrated with a diagram.

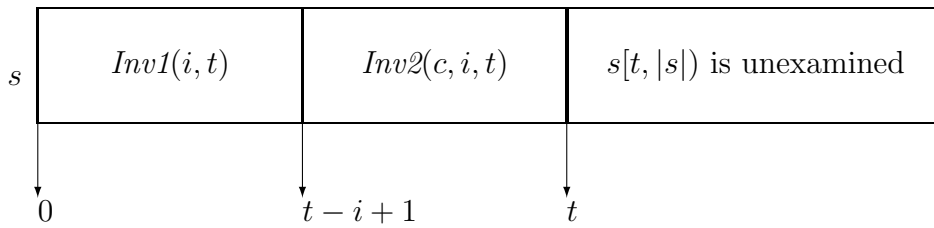


Figure 4.9.1: Visual depiction of $Inv(c, i, t) \triangleq Inv1(i, t) \wedge Inv2(c, i, t)$

$$\begin{aligned}
 Inv1(i, t) &\triangleq \forall k : k \in [0, t - i + 1) : \\
 &\quad (\langle k, p \rangle \in MO \iff ((p \in P) \wedge (p[0, |p|] = s[k, k + |p|]))) \\
 Inv2(c, i, t) &= \rho(c, i) \neq \emptyset \wedge (\forall p : p \in \rho(c, i) : p[0, i - 1] = s[t - i + 1, t)) \\
 Inv(c, i, t) &= Inv1(i, t) \wedge Inv2(c, i, t)
 \end{aligned}$$

Algorithm 5. *PEPL Automaton Based Matching - Refactored*

```

proc  $PAutMatch(\overline{\mathfrak{P}}^\#, s)$ 
   $MO := \emptyset;$ 
   $FA := getFA(\overline{\mathfrak{P}}^\#);$ 
   $c, i, t := \diamond 1, 1, 0;$ 
  { invariant:  $Inv(c, i, t)$  }
  do  $(t < |s|) \rightarrow$ 
     $c, j := \delta(c, \langle i, s[t] \rangle), v(c, \langle i, s[t] \rangle);$ 
    if  $(\rho(c, i) \neq \emptyset) \rightarrow MO := MO \cup (\{t\} \times \rho(c, i))$ 
     $\parallel (\rho(c, i) = \emptyset) \rightarrow$  skip
    fi;
     $t, i := t + 1, i + 1 - j$ 
    {  $Inv(c, i, t)$  }
  od
  {  $Inv(c, i, t) \wedge (t = |s|)$  }
corp
{ post:  $MO$  is the set of match occurrences of  $P$  in  $s$  }

```

4.10 Conclusion

In this chapter common information about multiple keywords is encoded into a APEPL, to form the basis for discovering positional information about matching instances of those keywords in a linearly streamed text. The two new pattern matching algorithms are shown to have theoretical running-time comparable to the Aho-Corasick family of algorithms.

We have arrived at a particularly efficient algorithm, thanks to two observations about $PAutMatch$. Firstly, transitions in the APEPL-Automaton (δ) and jump updates (using transition output relation v) can be done in constant time using lookup tables. Secondly, the **if** statement can be made in constant time, and consists of simple integer arithmetic to advance through the lattice and target s , and an update of MO (only if a match has been found). The latter can be done using a precomputed lookup table, as is done in the Advanced AC. These two characteristics are also found in the Advanced AC, and is unavoidable in pattern matching algorithms, giving us the same exact (worst- and best-case) running time of $|s|$.

The example in Table 4.4 illustrates how, in contrast to the example in Table 4.2, each symbol in the string *aaabcdabccd* is visited exactly once to match the keywords $\{abc, aabc, abcc\}$.

Ongoing work involves benchmarking the new algorithms against the Aho-Corasick and other multiple keyword pattern matching algorithms. We are also finding ways in which FCA can be effectively used in other stringology contexts [KWCV12].

Part III

Formal Concept Analysis applied to automata

Chapter 5

Failure Deterministic Finite Automata construction based on FCA

5.1 Introduction

In this chapter we show FCA applied to the problem of constructing Failure Deterministic Finite Automata. We present two algorithms for this task. The first algorithm presented in Section 5.4 is based on a DFA-homomorphic approach and the second algorithm presented in Section 5.5 is based on a Lattice-homomorphic approach.

Definition 5.1.1 (Undefined Value). In general, the symbol *nil* will be used to indicate an undefined value. Thus for the deterministic finite automaton (DFA) $\mathcal{D} = (Q, \Sigma, \delta, F, s)$, the transition relation $\delta(q, a) = \text{nil}$ means state $q \in Q$ has no out-transition on symbol $a \in \Sigma$. \square

Additionally, av denotes a string in Σ^+ if $a \in \Sigma$ and $v \in \Sigma^*$. It will also be convenient to rely on the functions *head*, *tail* and the *extended transition function* δ^* defined as follows:

Definition 5.1.2 (Head and tail of a string). $\text{head} \in \Sigma^+ \rightarrow \Sigma$ and $\text{tail} \in$

$\Sigma^+ \rightarrow \Sigma^+$ where

$$\text{head}(av) = a$$

$$\text{tail}(av) = v$$

□

Definition 5.1.3 (DFA extended transition function). $\delta^* \in Q \times \Sigma^* \rightarrow Q$ where

$$\delta^*(q, w) = \begin{cases} q & \text{if } w = \varepsilon \\ \delta^*(\delta(q, \text{head}(w)), \text{tail}(w)) & \text{otherwise} \end{cases}$$

□

Definition 5.1.4 (Language of a DFA). The function δ^* can now be used to define $\mathcal{L}(\mathcal{D})$, the language of \mathcal{D} . Specifically,

$$\mathcal{L}(\mathcal{D}) = \{w \mid \delta^*(s, w) \in F\}$$

□

Furthermore, the widely known classical Algorithm 6 can be used to test $x \in \mathcal{L}(\mathcal{D})$ for an arbitrary finite-length string $x \in \Sigma^*$.

Algorithm 6. *Test for string membership of a DFA's language*

```

{ pre ( $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ )  $\wedge$  ( $x \in \Sigma^*$ )  $\wedge$  ( $|x| < \infty$ ) }
 $y, q := x, s;$ 
{ invariant :  $y$  is untested and the current state is  $q$  }
do ( $(y \neq \varepsilon)$  cand ( $\delta(q, \text{head}(y)) \neq \text{nil}$ ))  $\rightarrow$ 
     $q, y := \delta(q, \text{head}(y)), \text{tail}(y)$ 
od;
{ ( $y$  is untested and the current state is  $q$ )  $\wedge$  ( $(y = \varepsilon)$  cor }
{ ( $\delta(q, \text{head}(y)) = \text{nil}$ ) }
 $\text{accept} := ((y = \varepsilon) \wedge (q \in F))$ 
{ post ( $\text{accept} \Leftrightarrow x \in \mathcal{L}(\mathcal{D})$ ) }

```

Variations of this simple algorithm are conventionally used to check for string membership of the regular language defined by a given DFA. The algorithm

clearly takes time $\mathcal{O}(|x|)$ if we assume that computing $\delta(q, a)$ is constant-time. It is $\mathcal{O}(|\Sigma| \times |Q|^2)$ in terms of space efficiency, because δ has to be stored—generally as a table or as a labelled directed (transition) graph. Applications of the algorithm vary widely, and it is not uncommon that the underlying DFA may involve millions of states and transitions. Consequently, research efforts have been directed at improving on the algorithm’s space or time efficiency. Examples include the DFA minimization algorithms [Wat95], hard-coding and cache manipulation strategies [KN07], stretching and jamming of alphabet representation [dBCKW10], construction of super-automata (or approximate automata) [CKW09], various strategies for storing sparse matrices [TY79, FKS84, DDH84, DH95], and other strategies to reduce representation sizes [DW11].

Here we propose a strategy for improving on the space efficiency of DFAs by relying on a formalism that we will call a *failure* deterministic finite automaton (FDFA). The formalism derives from the failure functions found in classical pattern matching algorithms [BM77, AC75, KJHMP77]. Recall, for example, the Aho-Corasick algorithm which takes a finite set of patterns $X \subseteq \Sigma^*$ and identifies all locations in a text $S \in \Sigma^*$ at which some pattern from X occurs. It comes in two versions. Following [Wat95, Chapter 4], we refer to the first version as AC-OPT and to the second as AC-FAIL. AC-OPT essentially builds a (minimal) DFA from the regular expression Σ^*X . A variant of Algorithm 6, based on this DFA, then consumes S , identifying all occurrences of elements of X in S . However, since the DFA is typically space intensive, AC-FAIL is an alternative devised to remove arcs that do not contribute to the definition of elements of X , replacing them judiciously with arcs derived from a so-called failure function. Graphically, the resulting structure is a conventional trie DFA [Fre60] of the words in X , decorated by various failure arcs. Algorithm 6 is adapted to traverse this structure, and to identify elements of X in S . The output of AC-FAIL is identical to AC-OPT but the total number of trie and failure arcs is significantly less than the number of arcs in AC-OPT. Benchmarks reported in [Wat95, Chapter 13] suggest that the gain in space efficiency comes at the cost of about 20% reduction in processing speed. Crochemore and Hancart [CH97] illustrate how the failure arc placement can sometimes be further optimised, but no benchmark information is provided about the time-impact of these optimisations and they do not give a construction for DFAs in general.

Our contribution shows how these ideas can be generalised by building an FDFA from any (complete) DFA. To this end, the next section formalises the notion of an FDFA and its language, showing how to adapt Algorithm 6 to recognise words in such a language. This is followed by a section that introduces formal concept lattices [CR04b]. These can be leveraged to derive FDFAs from a given DFA, as described in Section 5.4 and Section 5.5. The remainder of the chapter then identifies some areas that require further study.

5.2 Failure Deterministic Finite Automata

Definition 5.2.1 (FDFA). $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is a failure DFA (FDFA) if $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is a DFA and $\mathfrak{f} : Q \rightarrow Q$ is a partial function. We shall call \mathcal{D} the *embedded DFA* of \mathcal{F} .

We will refer to \mathfrak{f} as the *failure function* of \mathcal{F} . Its domain is indicated by $\text{dom } \mathfrak{f}$. If $q \in \text{dom } \mathfrak{f}$, then q will be called a *failure state*. If $q \notin \text{dom } \mathfrak{f}$ then this will be denoted by $\mathfrak{f}(q) = \text{nil}$ (i.e. $\mathfrak{f}(q)$ is not defined). \square

Definition 5.2.2 (Alphabet of a state). We use $\Sigma_q = \{a : \delta(q, a) \neq \text{nil}\}$ to denote the set of alphabet symbols labeling out-transitions of state q of an FDFA, and \mathbb{Z}_q for $\Sigma \setminus \Sigma_q$. Assuming $L \subseteq \Sigma^*$ and $u \in \Sigma$, then $u \cdot L = \{uw : w \in L\}$. Of course, if $L = \emptyset$ then $u \cdot L = \emptyset$. \square

Definition 5.2.3 (Right language of an FDFA's state). The right language of state q in FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$, denoted by $\vec{\mathcal{L}}(\mathcal{F}, q)$, is defined as follows:

$$\vec{\mathcal{L}}(\mathcal{F}, q) = \vec{\mathcal{L}}_\delta(\mathcal{F}, q) \cup \vec{\mathcal{L}}_{\mathfrak{f}}(\mathcal{F}, q)$$

where

$$\vec{\mathcal{L}}_\delta(\mathcal{F}, q) = \left(\bigcup_{b \in \Sigma_q} b \cdot \vec{\mathcal{L}}(\mathcal{F}, \delta(q, b)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$\vec{\mathcal{L}}_{\mathfrak{f}}(\mathcal{F}, q) = \begin{cases} \vec{\mathcal{L}}(\mathcal{F}, \mathfrak{f}(q)) \cap (\mathbb{Z}_q \Sigma^*) & \text{if } \mathfrak{f}(q) \neq \text{nil} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that $\bigcup_{b \in \Sigma_q} b \cdot \vec{\mathcal{L}}(\mathcal{F}, \delta(q, b))$ is \emptyset if $\Sigma_q = \emptyset$ (because \emptyset is the unit of \cup , which applies in \cup -quantification over an empty range). \square

Thus, the right language of an FDFFA in state q , written $\vec{\mathcal{L}}(\mathcal{F}, q)$, consists of three components:

- the set of all strings that can be generated from that state by making a conventional DFA transition to the next state on one of the out-transition symbols in Σ_q , together with
- ε if q is a final state, together with
- those words in $\vec{\mathcal{L}}_{\mathfrak{f}}(\mathcal{F}, q)$ (the right language of the next state as determined by the failure function at q) that begin with a symbol *not* in Σ_q , because any word beginning with a symbol in Σ_q would already have caused a conventional DFA transition from q .

(Such a recursive definition of right language is well-formed, as it is essentially a set of right-linear grammar equations whose solution is the right languages of the states.)

Definition 5.2.4 (Language of an FDFFA). The language of an FDFFA \mathcal{F} with start state s is denoted by $\mathcal{L}(\mathcal{F})$ and is defined as $\vec{\mathcal{L}}(\mathcal{F}, s)$. \square

Definition 5.2.5 (FDFFA Equivalences). An FDFFA or DFA \mathcal{D} is said to be equivalent to the FDFFA \mathcal{F} iff $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{D})$. This will be denoted by $\mathcal{F} \equiv \mathcal{D}$. \square

Clearly, the embedded DFA of an FDFFA is not, in general, equivalent to the FDFFA. However, equivalence holds in the degenerate case, i.e. when $\mathfrak{f} = \emptyset$. In this sense, a DFA may be regarded as a special case of an FDFFA—it is an FDFFA that has a degenerate failure function. Note that for a given FDFFA, there could be many equivalent DFAs and vice-versa. Indeed, a regular language \mathcal{R} induces equivalence classes of DFAs and FDFFAs, namely $\mathcal{E}_{\mathcal{D}}(\mathcal{R}) = \{\mathcal{D} \mid \mathcal{D} \text{ is a DFA} \wedge \mathcal{L}(\mathcal{D}) = \mathcal{R}\}$ and $\mathcal{E}_{\mathcal{F}}(\mathcal{R}) = \{\mathcal{F} \mid \mathcal{F} \text{ is an FDFFA} \wedge \mathcal{L}(\mathcal{F}) = \mathcal{R}\}$ respectively. In fact, since every DFA can be seen as a degenerate FDFFA, $\mathcal{E}_{\mathcal{D}}(\mathcal{R}) \subseteq \mathcal{E}_{\mathcal{F}}(\mathcal{R})$. In Section 5.4, an algorithm is proposed which derives an $\mathcal{F} \in \mathcal{E}_{\mathcal{F}}(\mathcal{R})$ from a given $\mathcal{D} \in \mathcal{E}_{\mathcal{D}}(\mathcal{R})$.

Definition 5.2.6 (Failure path). A sequence of states, $\langle p_0, p_1, \dots, p_n \rangle$ of length $n > 0$ is a *failure path* from p_0 to p_n , written $p_0 \xrightarrow{\mathfrak{f}} p_n$, iff $\forall i \in [0, n) : \mathfrak{f}(p_i) = p_{i+1}$. \square

We also use $p \xrightarrow{\mathfrak{f}} q$ as a predicate asserting that there is such a failure path.

Definition 5.2.7 (Failure path alphabet). If $p_0 \xrightarrow{f} p_n = \langle p_0, p_1, \dots, p_n \rangle$ is a failure path, then we use $\Sigma_{p_0 \xrightarrow{f} p_n}$ to denote its *failure alphabet* $\Sigma_{p_0} \cap \Sigma_{p_1} \cap \dots \cap \Sigma_{p_n}$. \square

Intuitively, this is known as the failure alphabet because at each single failure step p_k to p_{k+1} , the only alphabet symbols which may cause such a failure step are those in Σ_{p_k} .

Definition 5.2.8 (Failure cycle). A failure path $p_0 \xrightarrow{f} p_0 = \langle p_0, p_1, \dots, p_0 \rangle$ is called a *failure cycle*. \square

Definition 5.2.9 (Divergent failure cycle). A failure cycle $p_0 \xrightarrow{f} p_0$ such that $\Sigma_{p_0 \xrightarrow{f} p_0} \neq \emptyset$ (i.e. its failure alphabet is non-empty) is a *divergent failure cycle*. \square

The term divergent comes from its use in process algebras to describe a system that is trapped into non-productive state changes, which is exactly what is implied by the term divergent in our context.

The failure function description in [CH97] also allows for failure arcs in a general DFA setting, although no algorithm for constructing FDFAs in general is presented, and failure arc cycles are essentially prohibited there. In fact, that requirement is stronger than necessary: only divergent failure cycles are problematic. Our FDFA definition does not preclude (divergent) failure cycles; it also allows for useless states and transitions, now including useless failure arcs (i.e. failure arcs from states q s.t. $\Sigma_q = \Sigma$). We do not consider such cases in detail here, but list some example cases (other than the case of divergent failure cycles) below.

- Consider an FDFA with two states, p and q , with p being the initial state, and with transitions on a from both states (their destinations do not matter) and a failure transition from p to q . In such a case, the transition on a from q makes no contribution to the FDFA's language, even if it follows paths to final states.
- Consider any FDFA which has a state p with $\Sigma_p = \Sigma$. There is nothing in the FDFA definition preventing a failure transition originating in p , even though such a transition is useless.

For the case of FDFAs with divergent failure cycles, an algorithm including

cycle detection can be conceived of. Nevertheless, from here on, we assume any FDFA construction we present will not introduce divergent failure cycles, and will not introduce any useless states and transitions either, preventing all of the above cases from arising.

Based on transition data from an FDFA, \mathcal{F} , the classic algorithm for string recognition given in Algorithm 6 can be modified to recognise strings from $\mathcal{L}(\mathcal{F})$. The general idea is as follows:

Continue the main/outer loop for as long as the current state, q , has an out-transition labelled by the symbol currently under test (i.e. $\delta(q, \text{head}(y)) \neq \text{nil}$), or the failure function, f , is defined at state q (i.e. $f(q) \neq \text{nil}$).

Within the loop, if there is no out-transition labelled by the symbol of the input string currently under test, then transition to a new state as determined by the failure function, but do not consume the symbol. This might be thought of as *failing* to make a normal state transition, and instead, making a transition determined by the failure function.

Algorithm 7 formalises these steps. It assumes FDFA $\mathcal{F} = (Q, \Sigma, \delta, f, F, s)$ is given, and shows how to determine whether a finite string, $x \in \Sigma^*$ is in $\mathcal{L}(\mathcal{F})$. To highlight the symmetry with Algorithm 6, we rely on the multiple guarded command format for the repeat loop in Dijkstra's Guarded Command Language¹.

Algorithm 7. *Test for string membership of an FDFA's language*

```

{ pre  $(x \in \Sigma^+) \wedge (|x| < \infty)$  }
 $y, q := x, s;$ 
{ invariant:  $y$  is untested and the current state is  $q$  }
do  $(y \neq \varepsilon)$  cand  $(\delta(q, \text{head}(y)) \neq \text{nil}) \rightarrow q, y := \delta(q, \text{head}(y)), \text{tail}(y)$ 
|  $(y \neq \varepsilon)$  cand  $(\delta(q, \text{head}(y)) = \text{nil}) \wedge (f(q) \neq \text{nil}) \rightarrow q := f(q)$ 

```

¹In this form, the loop comprises of several guarded commands of the form $G \rightarrow S$ where G is a boolean expression and S is a command. All guards are evaluated at the start of each iteration, and a statement is non-deterministically selected for execution from amongst all the guards which evaluate to **true**. If no guard evaluates to **true** then the loop terminates [DF88, pp. 44 and further].

```

od;
{ y is untested and the current state is q
   $\wedge ((y = \varepsilon) \text{ cor } ((\delta(q, \text{head}(y)) = \text{nil}) \wedge (\mathfrak{f}(q) = \text{nil})))$  }
accept :=  $((y = \varepsilon) \wedge (q \in F))$ 
{ post (accept  $\Leftrightarrow x \in \mathcal{L}(\mathcal{F})$ ) }

```

If we know that $\mathcal{F} \equiv \mathcal{D}$ where \mathcal{D} is a DFA, then Algorithm 7 shows that an FDFA can be used to determine string membership of $\mathcal{L}(\mathcal{D})$. It does so at the cost of an additional test in the loop, to verify whether or not there is a failure transition to be made if no conventional transition can be made. It therefore operates in $\mathcal{O}(|x|)$ time in the best case, but in the worst case it has to traverse the path of an entire failure cycle before having a symbol of x consumed. Since the longest possible non-divergent cycle is $|Q| - 1$, the algorithm's worst case performance is described by $\mathcal{O}(|x| \times (|Q| - 1))$. As noted before, the corresponding DFA string membership algorithm operates in $\mathcal{O}(|x|)$.

However, there is a potential savings in arc storage if an FDFA is used instead of a DFA. To illustrate this claim, consider the DFA depicted in Figure 5.2.1. It will serve throughout as the example DFA to be transformed into an FDFA. It has a total of sixteen arcs. (Doubly labelled arcs are counted twice, because storage is required to represent each transition.)

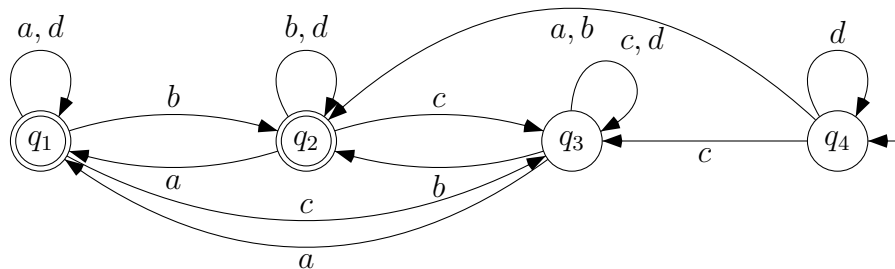


Figure 5.2.1: Initial DFA: $|\delta| = 16$, $|\mathfrak{f}| = 0$

We shall show below that the FDFA depicted in Figure 5.2.2 is equivalent to the DFA depicted in Figure 5.2.1. It has only eight normal transition arcs, and three failure function transitions (represented by dotted arcs).

This saving in arcs is possible because a conventional DFA is sometimes redundant, in the sense that there may be transitions to the same state from

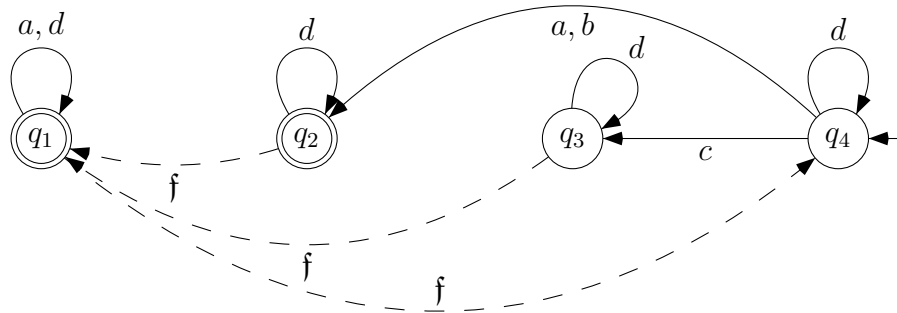


Figure 5.2.2: FDFFA equivalent to the DFA in Figure 5.2.1: $|\delta| = 8$, $|\mathbf{f}| = 3$

several destinations, all on the same transition². For example, in Figure 5.2.1, all states make a transition to state q_1 on a , to state q_2 on b and to state q_3 on c . The FDFFA in Figure 5.2.2 is designed to handle transitions that are unique at each state, and to *fail over* to another state if the transition to be made on a set of symbols is shared with other states. Thus, in state q_2 , a transition on d is determined locally (in fact, the destination is coincidentally q_2 itself), whereas on all other symbols, a failure transition is made to state q_1 , since the behaviour from state q_2 on those transitions is exactly the same as the behaviour from state q_1 for the same transitions: each goes to q_1 on symbol a , to q_2 on symbol b and to q_3 on symbol c . Similar remarks apply to state q_3 . In the case of state q_1 , transitions on a and d are handled locally, but fail-over to state q_4 occurs on other symbols. State q_4 handles all transitions locally—there are no fail-over transitions.

An example of an FDFFA for which the worst case bound of $\mathcal{O}(|x| \times (|Q| - 1))$ can be obtained is depicted in Figure 5.2.3. For $x = ab$, processing will take the FDFFA to state q_f using two symbol transitions and two failure transitions.

Since both δ and \mathbf{f} need to be stored, an FDFFA needs at most $\mathcal{O}(|Q|^2 \times (|\Sigma| + 1))$ space. However, the actual storage will be much less than this worst case estimate in as much as δ can be minimized when constructing the FDFFA. The challenge taken up here, therefore, is to derive from a DFA (seen here as a degenerate FDFFA) say $\mathcal{F}' = (Q', \Sigma, \delta', \emptyset, F', s')$, an equivalent FDFFA, say $\mathcal{F} = (Q, \Sigma, \delta, \mathbf{f}, F, s)$, such that $|\delta'| - (|\delta| + |\mathbf{f}|)$ is as large as possible. Because

²Minimization of DFAs relies on such redundancy, but only works when two states have *equal* right languages. FDFFA space savings occur also when a state's right language *contains* the right language of another state.

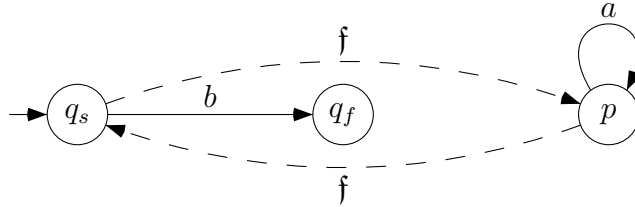


Figure 5.2.3: FDFA with which worst case bound of $\mathcal{O}(|x| \times (|Q| - 1))$ on processing a string x is achievable.

of the benchmarking results reported in [Wat95], we conjecture that the time penalty will be at most ca. 20%. In general $|\delta'| - (|\delta| + |\mathfrak{f}|)$ should be large to effect space savings, but how large depends on the extent to which such space economies will degrade time-performance when running Algorithm 7 and how critical such tradeoffs are in a given application context³. These are matters for further study. Algorithm 8 expresses this derivation in an abstract fashion.

Algorithm 8. *Transforming a DFA into an FDFA*

$$\{ \text{pre } \mathcal{F}' = (Q', \Sigma, \delta', \emptyset, s', F') \}$$

$$\text{Infer } (Q, \delta, \mathfrak{f}, s, F) \text{ from } (Q', \delta', \emptyset, s', F') \text{ such that } (\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}'))$$

$$\{ \text{post } \mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, s, F) \wedge (\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}')) \}$$

A specialisation/restriction is to require that the various states, Q , s and F are not changed. This means that these states can be regarded as constants in the algorithm, while δ and \mathfrak{f} are variables whose values change from their initial values δ' and \mathfrak{f}' . The algorithm thus preserves the originating DFA's shape, and will, for this reason, be called a DFA-homomorphic algorithm. As yet another restriction, we also require that the right language of every state remains unchanged as part of the change. These restrictions lead to the following slightly more explicit algorithm:

³Note that the storage requirements for an entry in \mathfrak{f} are less than the storage requirements for an entry in δ or δ'

Algorithm 9. *Transforming a DFA into an FDFA*

$$\begin{aligned} & \{ \text{pre } \mathcal{F}' = (Q, \Sigma, \delta', \emptyset, s, F) \} \\ & \quad \delta, \mathfrak{f} := \delta', \emptyset; \\ & \quad \text{Change } (\delta, \mathfrak{f}) \text{ such that } \forall q : Q : \vec{\mathcal{L}}(\mathcal{F}, q) = \vec{\mathcal{L}}(\mathcal{F}', q) \\ & \{ \text{post } \mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, s, F) \wedge (\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}')) \} \end{aligned}$$

One way of changing δ and \mathfrak{f} while preserving the right languages of all states, is expressed in Theorem 5.2.11 below.

Definition 5.2.10. For $p, q \in Q$ and $X \subseteq \Sigma$, $\text{FailPred}(p, q, X)$ is defined by

$$(\Sigma_p = \Sigma) \wedge (\forall a \in X : (\delta(p, a) = \delta(q, a))) \wedge (\mathfrak{f}(p) = \text{nil}) \wedge (q \xrightarrow{\mathfrak{f}} p \Rightarrow (\Sigma_{q \xrightarrow{\mathfrak{f}} p} \cap X = \emptyset))$$

□

Theorem 5.2.11 (A transformation that preserves right languages). Let \mathcal{F} be an FDFA such that there exist $p, q \in Q$ and $X \subseteq \Sigma$ for which $\text{FailPred}(p, q, X)$ holds. Then deleting from δ all transitions from p on each symbol in X , and adding a failure arc from p to q leaves the right languages of all states of \mathcal{F} unchanged. □

Corollary 5.2.12. The FDFA resulting from such a transformation is equivalent to the initial FDFA, $|\delta|$ has decreased by $|X|$ and $|\mathfrak{f}|$ has increased by 1.

Applying Theorem 5.2.11 leads to the following refinement of Algorithm 9.

Algorithm 10. *Transforming a DFA into an FDFA*

```

{ pre  $\mathcal{F}' = (Q, \Sigma, \delta', \emptyset, s, F)$  }
   $\delta, \mathfrak{f} := \delta', \emptyset$ 
  do  $(\exists p, q, X : p, q \in Q, X \subseteq \Sigma :$ 
     $(\Sigma_p = \Sigma) \wedge (\mathfrak{f}(p) = nil)$ 
     $\wedge (\forall a \in X : (\delta(p, a) = \delta(q, a)))$ 
     $\wedge ((q \xrightarrow{\mathfrak{f}} p \Rightarrow (\Sigma_{q \xrightarrow{\mathfrak{f}} p} \cap X = \emptyset))) \rightarrow$ 
    for each  $(a \in X) \rightarrow \delta := \delta \setminus \{\langle p, a, \delta(p, a) \rangle\}$  rof;
     $\mathfrak{f}(p) := q$ 
     $\{ \vec{\mathcal{L}}(\mathcal{F}, p) = \vec{\mathcal{L}}(\mathcal{F}', p) \}$ 
  od
{ post  $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, s, F) \wedge (\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}'))$  }

```

Note that the do-loop in this algorithm could be terminated at any point deemed suitable in the given context. The essential task in any concrete implementations of Algorithm 10 is to identify p, q and X that conform to Theorem 5.2.11. The next section introduces relevant ideas from formal concept analysis, a field that attempts to capture and represent commonalities between different objects in so-called formal concept lattices. Subsequently we will show how a concept lattice may be used to comprehensively identify all instances of p, q and X in \mathcal{F} .

5.3 State / Out-Transition Formal Concept Lattices

A *formal concept lattice* can be defined in a domain of discourse consisting of a set of objects, and a set of attributes that the various objects possess. In such a domain of discourse, a *concept* is considered to be a pair of two sets: a set of objects, called the concept's *extent*; and a set of attributes, called the concept's *intent*. All objects in the concept's extent have in common all and only the attributes in the intent. Furthermore, the extent is maximal over the objects, in the sense that there may not be any objects outside of the concept's extent which also possesses all the attributes in the intent.

In the theory known as formal concept analysis, such concepts are considered to be partially ordered: if c_i and c_j are two arbitrary concepts in the domain of discourse, and if $ext(c)$ denotes the extent of concept c , then $c_i \leq c_j \Leftrightarrow ext(c_i) \subseteq ext(c_j)$. Equality holds if and only if $i = j$. Furthermore, it can be shown that there is a duality in the role of objects and attributes, such that if $int(c)$ denotes the intent of concept c , then $c_i \leq c_j \Leftrightarrow int(c_j) \subseteq int(c_i)$.

The starting point in formal concept analysis is to represent the relationship between objects and attributes in a given domain of discourse in terms of a cross table known as a *context*. An example of a context to be discussed later is shown in Table 5.1. The rows represent the objects p_1, \dots, p_4 and the columns represent attributes designated $\langle a, p_1 \rangle, \langle a, p_2 \rangle, \langle b, p_2 \rangle, \dots, \langle d, p_4 \rangle$. (The reason for these rather strange attributes will be discussed later.) An entry in a cell indicates that the relevant object has the indicated attributes. Thus, for example, object p_4 has attributes $\{\langle a, p_2 \rangle, \langle b, p_2 \rangle, \langle c, p_3 \rangle, \langle d, p_4 \rangle\}$.

It can be shown that the partial ordering discussed above, over all possible concepts implied by such a context, constitutes a lattice. Various lattice construction algorithms have been devised to extract all possible concepts from a given context and to arrange them in a graph structure that reflects their parent/child relationships [KO02, KOWvdM09]. A tool called Concept Explorer [Yev06] was used to generate from the context in Table 5.1, the line diagram shown in Figure 5.3.1. The diagram shows the ordering of concepts in the concept lattice. Concepts have been labelled $c_1, \dots, c_4, c_{123}, c_{1234}$. The following

Table 5.1: The DFA's state/out-transition context

| | $\langle a, p_1 \rangle$ | $\langle a, p_2 \rangle$ | $\langle b, p_2 \rangle$ | $\langle c, p_3 \rangle$ | $\langle d, p_1 \rangle$ | $\langle d, p_2 \rangle$ | $\langle d, p_3 \rangle$ | $\langle d, p_4 \rangle$ |
|-------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| p_1 | 1 | | 1 | 1 | 1 | | | |
| p_2 | 1 | | 1 | 1 | | 1 | | |
| p_3 | 1 | | 1 | 1 | | | 1 | |
| p_4 | | 1 | 1 | 1 | | | | 1 |

are examples of concept extents: $ext(c_1) = \{p_1\}$ and $ext(c_{123}) = \{p_1, p_2, p_3\}$. Similarly, examples of concept intents include $int(c_4) = \{\langle d, p_4 \rangle, \langle a, p_2 \rangle, \langle c, p_3 \rangle, \langle b, p_2 \rangle\}$; $int(c_{123}) = \{\langle a, p_1 \rangle, \langle c, p_3 \rangle, \langle b, p_2 \rangle\}$. Thus, concept c_{123} indicates that objects p_1, p_2 and p_3 (its extent) have all and only the attributes $\langle a, p_1 \rangle, \langle c, p_3 \rangle$ and $\langle b, p_2 \rangle$ (its intent) in common.

Concept c_{123} illustrates the fact that the extent of a concept is the union of

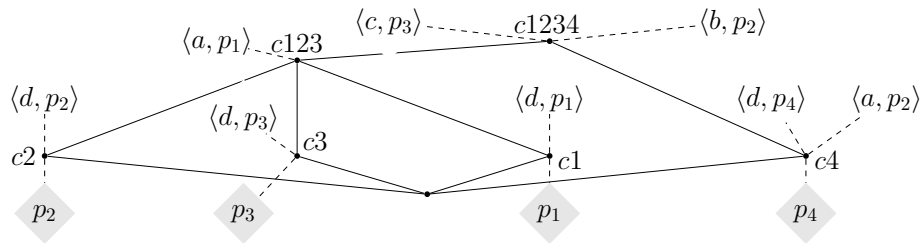


Figure 5.3.1: The DFA's state/out-transition formal concept lattice

the extents of all its children, together with any of its so-called “own objects”. In this case concept $c123$ does not have any own objects. Its children are $c1$, $c2$ and $c3$, and their respective extents correspond to their own objects, which are explicitly shown in the diagram—i.e. their extents are $\{p_1\}$, $\{p_2\}$ and $\{p_3\}$ respectively.

Dually, concept $c123$ also illustrates the fact that the intent of a concept is the union of the intents of all its parents, together with any of its so-called “own attributes”. In this case concept $c123$ has $\langle a, p_1 \rangle$ as its single own attribute, and its only parent, $c1234$, adds its intent, $\{\langle c, p_3 \rangle, \langle b, p_2 \rangle\}$, to the intent of $c123$.

Information in a DFA's transition graph can be represented in a context, and hence as a formal concept lattice. Here we propose one particular way and we call the resulting lattice a *state/out-transition (formal concept) lattice*. We will denote the state/out-transition lattice of DFA D by $\mathcal{SO}(\mathcal{D})$.

The set of objects in $\mathcal{SO}(\mathcal{D})$ is simply the set of states in D , namely Q . Each attribute is a pair consisting of the label of an out transition from some state, and the corresponding destination state. Formally, $\langle b, p \rangle$ is an attribute in $\mathcal{SO}(\mathcal{D})$ if and only if $\exists q : q \in Q : \delta(q, b) = p$. In this case, $\langle b, p \rangle$ is to be regarded as an attribute of the object q . By inspection it can be seen that the context in Table 5.1 has been derived from the DFA in Figure 5.2.1 in precisely this way. The line diagram in Figure 5.3.1 is therefore a visual representation of $\mathcal{SO}(\mathcal{D})$.

The space and time requirements building the lattice's context table are determined by the size of δ , i.e. they are $\mathcal{O}(|Q|^2 \times |\Sigma|)$. An SO-lattice is a constrained lattice in the sense that its objects are constrained to each have exactly one attribute from each of $|\Sigma|$ classes, each class having $|Q|$ attributes.

In [KO98] it is shown that number of concepts for such a lattice is bound from above by $\min((1 + |\Sigma|)^{|Q|}, \frac{|Q|}{(1 + |\Sigma|)} 2^{1 + |\Sigma|})$. For convenience, we shall denote this expression by $\mathcal{LB}(\Sigma, Q)$. This means that for a fixed alphabet, an upper bound of the lattice size eventually becomes linearly dependent on the number of states.

5.4 A DFA-Homomorphic Algorithm

A key point to note about any concept c in $\mathcal{SO}(\mathcal{D})$ is the following. Per definition of a concept, all states in $\text{ext}(c)$ share all and only the out-transitions in $\text{int}(c)$. (Of course, each state in $\text{ext}(c)$ may have zero or more out-transitions that are not represented in $\text{int}(c)$.) For convenience, let $m = |\text{ext}(c)|$ and $n = |\text{int}(c)|$. Therefore, based on Theorem 5.2.11, the out-transitions may be rearranged in \mathcal{D} 's transition graph as follows, to produce an equivalent FDFA:

- Select any $q \in \text{ext}(c)$ to be the destination of a failure arc, and let $\text{ext}(c') = \text{ext}(c) \setminus \{q\}$.
- For each $p \in \text{ext}(c')$ remove all outgoing arcs represented in $\text{int}(c)$. Thus, the number of arcs removed from \mathcal{D} is $n(m - 1)$.
- For each $p \in \text{ext}(c')$ install an outgoing failure transition to t . Thus, the number of arcs added to \mathcal{D} is $(m - 1)$.

As a result of these steps, $n(m - 1) - (m - 1) = (n - 1)(m - 1)$ arcs will be removed from the initial structure. Moreover, the process of constructing failure arcs and removing arcs may be repeated on any interim FDFA obtained by the above process. The result will always be a new FDFA provided that the specific precondition requirements of Theorem 5.2.11 at state p are not violated.

For simplicity, we will henceforth assume that we are working with a complete DFA, namely that $\forall p : Q : \Sigma_p = \Sigma$. We can easily transform an arbitrary DFA to such complete DFA by introducing a sink (non-final) state, and inserting transitions to that state from any state that is incomplete. This process of introducing additional transitions into an arbitrary DFA should be taken into account for when considering any claims about reductions attained by transformations to an FDFA.

We will call $(n - 1)(m - 1)$ for a given node, c , the *arc redundancy* of c and denote it by $ar(c)$. For example, since both extent and intent of $c123$ contain 3 objects, its arc redundancy is $(3 - 1) \times (3 - 1) = 4$, as shown in Figure 5.3.1. Also shown in Figure 5.3.1 is the arc redundancy of $c1234$ as 3.

Note that if the above steps to construct a failure arc are applied to a concept c whose arc redundancy is 0, there will be no decline in the overall number of arcs of the resulting FDFA. Conversely, the maximum decline is obtained if one selects from all the concepts, the one for which $ar(c)$ is maximal. This suggests the following greedy-like algorithm for constructing FDFA \mathcal{F} from DFA \mathcal{D} , assuming that $\mathcal{SO}(\mathcal{D})$ is available. The algorithm computes and maintains a set, AR , of concepts with non-zero arc redundancy, as well as the set O of states which do not originate failure transitions, i.e. O is defined by $\text{dom } \mathbf{f} = Q \setminus O$. The algorithm assumes a function $maxcar : \mathcal{P}(\mathcal{SO}(\mathcal{D})) \rightarrow \mathcal{SO}(\mathcal{D})$ which selects from AR the concept, c with the maximum arc redundancy. In the version of the algorithm below, q is arbitrarily selected from $ext(c)$ to act as the target for failure arcs. Later, this selection will be slightly refined. The source of failure arcs is selected from those remaining in $ext(c)$ and treated, one by one, in the outer **for each** loop. The treatment is conditional on no divergent failure cycles between q and p coming into existence. The test for this will be explained later.

Algorithm 11.

```

AR, f := ∅, ∅
{ Compute concepts' arc redundancy }
for each (c ∈ SO(D)) →
    ar(c) := ((|ext(c)| - 1) × (|int(c)| - 1));
    if (ar(c) > 0) → AR := AR ∪ {c}
    || (ar(c) ≤ 0) → skip
    fi
rof;
{ AR is set of concepts with non-zero arc redundancy }
O := Q;
{ Invariant: (dom f = Q \ O) ∧ (Concepts in AR have not been processed) }
do ((O ≠ ∅) ∧ (AR ≠ ∅)) →
    c := maxcar(AR);
    AR := AR \ {c};
    let q ∈ ext(c);
    P := ext(c) \ {q};
    for each (p ∈ P ∩ O) →
        if ¬(q  $\xrightarrow{f}$  p) COR (Σq $\xrightarrow{f}$ p ∩ (dom int(c)) = ∅) →
            for each ((a, r) ∈ int(c)) →
                δ := δ \ {⟨p, a, r⟩}
            rof;
            f(p) := q;
            O := O \ {p}
        || (q  $\xrightarrow{f}$  p) CAND (Σq $\xrightarrow{f}$ p ∩ (dom int(c)) ≠ ∅) → skip
        fi
    rof
od
{ Invariant ∧ ((O = ∅) ∨ (AR = ∅)) }

```

Applying the greedy-like algorithm to the DFA in Figure 5.2.1, and making use of the state/out-transition lattice shown in Figure 5.3.1 will yield the FDFA shown in Figure 5.4.1 after the first iteration of the outer **do**-loop.

To see that this is so, note that since the size of the extent of each of the concepts c_1, c_2, c_3 and c_4 is 1, the arc redundancy of each of these concepts is

0, and therefore the concepts $c1$, $c2$, $c3$ and $c4$ do not appear in AR . In addition, note that $ar(c123) = (3-1) \times (3-1) = 4$ and $ar(c1234) = (2-1) \times (4-1) = 3$. As a result, upon entering the loop, $AR = \{c123, c1234\}$.

Thus, in the first iteration $maxcar(AR)$ returns concept $c123$ and the algorithm removes $c123$ from AR . Choosing $q = p_1$ (any element of $P = ext(c123) \setminus \{q\} = \{p_1, p_2, p_3\}$ could have been chosen) as the destination of all failure nodes in this iteration, the **for each** loop removes the following 6 arcs (δ mappings) from the DFA in Figure 5.2.1:

$$\{\langle p_2, a, p_1 \rangle, \langle p_2, b, p_2 \rangle, \langle p_2, c, p_3 \rangle, \langle p_3, a, p_1 \rangle, \langle p_3, b, p_2 \rangle, \langle p_3, c, p_3 \rangle\}$$

Thereafter, it inserts two failure transitions into δ , namely $\{\langle p_2, f, p_1 \rangle, \langle p_3, f, p_1 \rangle\}$. As a result, the number of arcs has been reduced by 4—as predicted by the arc redundancy of $c123$.

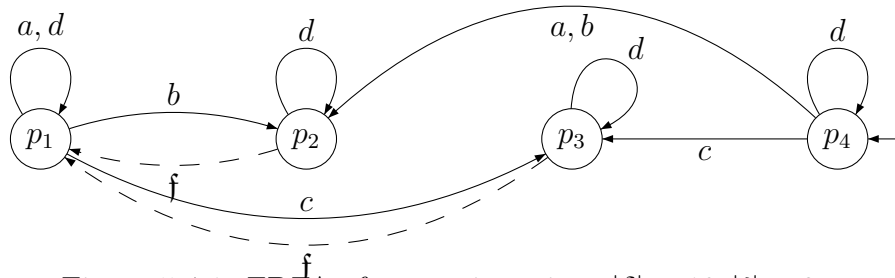


Figure 5.4.1: FDFA after one iteration. $|\delta| = 10$, $|f| = 2$

After the second iteration of the outer **do**-loop the FDFA in Figure 5.4.2 is obtained. Tracing the algorithm steps to confirm this, note that upon entering the loop for a second time, $AR = \{c1234\}$. $maxcar(AR)$ therefore returns concept $c1234$ (where $AR = \{p_1, p_2, p_3, p_4\}$). Choosing $q = p_4$ as the destination of all failure arcs in this iteration, (again an arbitrary choice from AR would do) the **for each** loop removes the following arcs from the DFA in Figure 5.4.1: $\langle p_1, b, p_2 \rangle$ and $\langle p_1, c, p_3 \rangle$. Instead, it inserts failure transition $\{\langle p_1, f, p_4 \rangle\}$, thus reducing the number of arcs by 1. Note that out transitions from p_2 and p_3 are not considered, since these two states already have a failed transition assigned to them in the previous iteration and are therefore no longer in O .

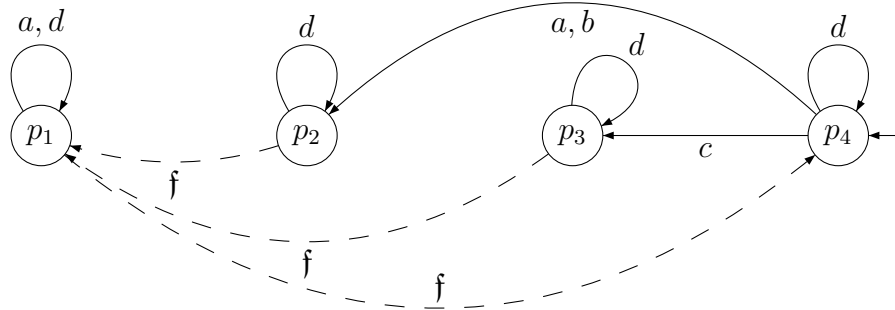
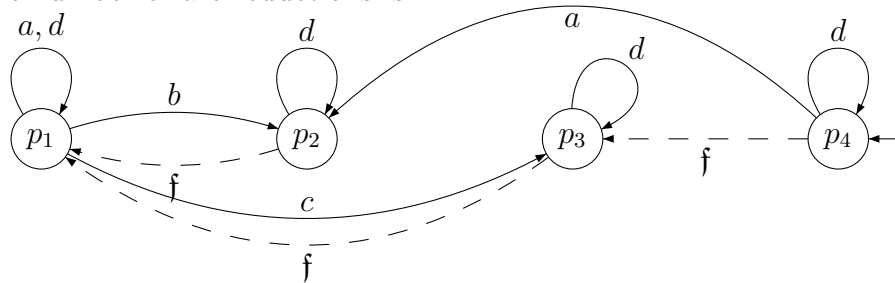
Figure 5.4.2: FDFFA after two iterations $|\delta| = 8, |\mathbf{f}| = 3$

Figure 5.4.3 shows an alternative FDFFA that would be derived if $q = p_3$ was chosen as the destination of all failure nodes instead of p_4 . The algorithmic steps to arrive at this form will be traced later. Note at this point, however, that the number of arc reductions is 1.

Figure 5.4.3: Alternative FDFFA after two iterations $|\delta| = 8, |\mathbf{f}| = 3$

5.4.1 Recomputing arc redundancy

The reduction in the number of arcs in the second iteration is by 1, both in Figure 5.4.2 and in Figure 5.4.3, which does *not* correspond to the initially computed value of $ar(c1234)$, namely 3. This is to be expected, because the algorithm computes concepts' arc redundancy only once—at the start of the algorithm. However, whenever states are removed from O (as in the above case, where p_2 and p_3 were removed from O after the first iteration) the arc redundancies of concepts in P may change in respect of those concepts whose extents contain removed states. As a result, the economising on arcs will be less than predicted by the initially computed arc redundancy metric.

In larger examples than the one given here, this might mean that the *maxcar* no longer chooses as “greedily” as it might have. To forestall this potential inefficiency (which does not affect overall accuracy, as long as Theorem 5.2.11 is observed), concept arc redundancy could be recomputed on the fly whenever the algorithm installs a failure arc from p to q (and thus removes p from O).

At that point, each concept, say c'' , whose extent contains state p could be located, and its arc redundancy appropriately adjusted in the manner explained below. Additionally, the way in which the target state for failure arcs, q , is selected in Algorithm 11 has to be modified: no longer should an arbitrary state in $ext(c)$ be selected; instead preference should be given to failure states (i.e. states already in $dom \mathbf{f}$). This is because a failure state is not allowed to serve as a *source* of an additional failure arc. Thus, if a non-failure state is selected as the target in a situation where a failure state could have served as such, then the number of arcs to be removed will be suboptimal because those of the failure states have to remain intact.

Reference to Figures 5.4.4, 5.4.5, 5.4.6 and 5.4.7 illustrates this point. Figure 5.4.4 shows a number of states in an FDFA that give rise to two concepts, $c12$ and $c2345$ in the associated state/out-transition lattice, as illustrated in Figure 5.4.5. The arc redundancies $ar(c12) = 2$ and $ar(c2345) = 3$. Transforming on the basis of concept $c2345$ leads to Figure 5.4.6 in the first step.

Note that in the next step towards reducing arcs, state p_2 cannot be chosen as a source of a failure arc to state p_1 , because it is already a failure state and FDFAs are deterministic and \mathbf{f} is a function. However, failure state p_2 can indeed be chosen as the *target* of a failure arc that emanates from state p_1 . Such a choice leads to Figure 5.4.7.

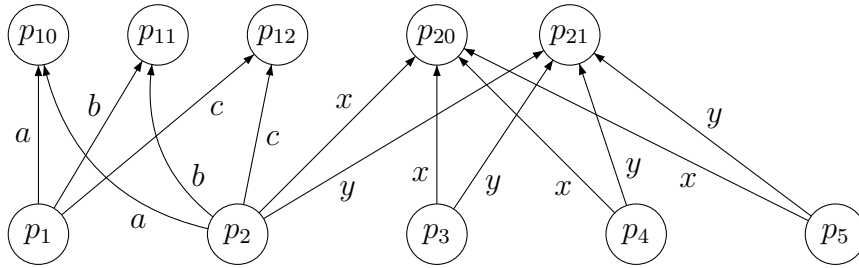


Figure 5.4.4: Part of a DFA

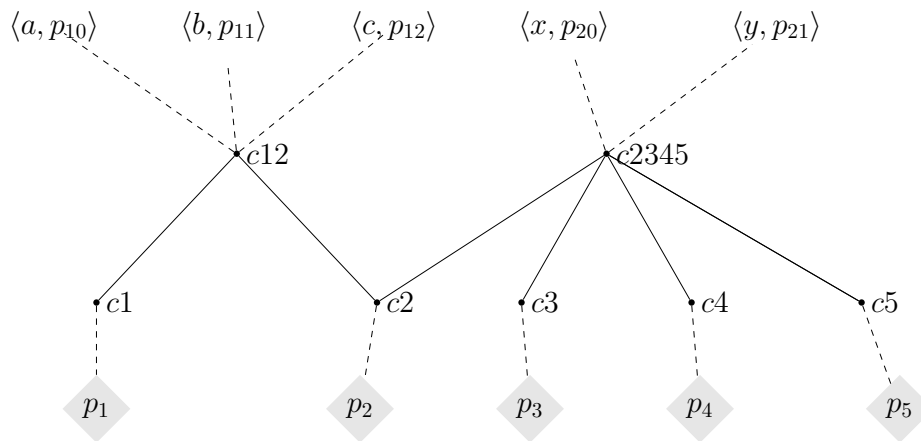


Figure 5.4.5: Two concepts in the state/out-transition concept lattice derived from Figure 5.4.4

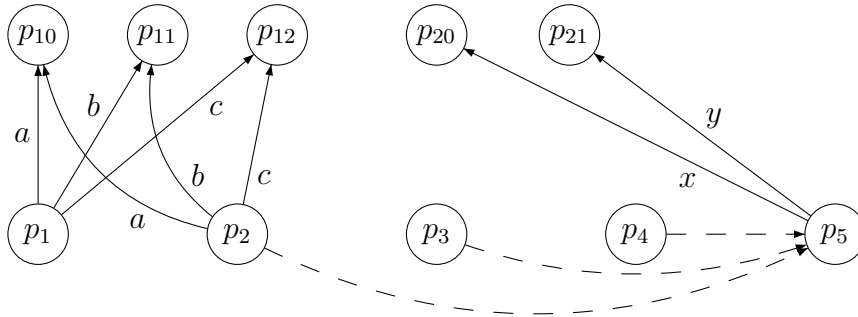


Figure 5.4.6: Transformation to FDFA after first step

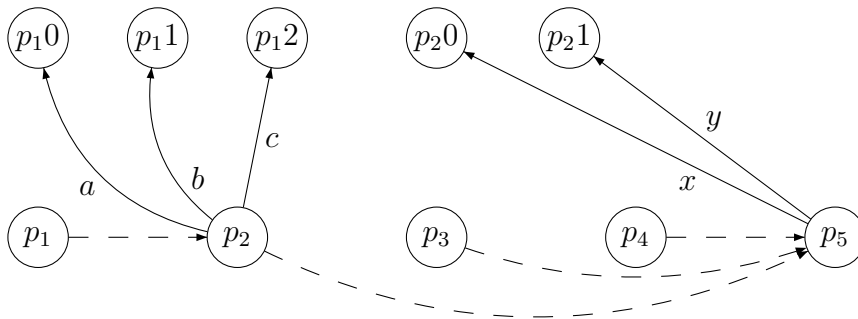


Figure 5.4.7: Using failure node as a failure arc target results in optimal solution.

The foregoing in mind, let us now return to the matter of identifying each concept, say c'' , whose extent contains a newly created failure state p . The question arises: Does this affect the computed arc redundancy of c'' , and if so, how should it be updated? In reference to Figure 5.4.5 this translates into the question: how does the $ar(c123)$ change, once p_2 becomes a failure arc?

In general, updating the arc redundancy of c'' has to be based on the number of failure states in $ext(c'')$. The expression $|ext(c'') \cap \text{dom } f|$ denotes this number.

Suppose that this value (initially 0) has changed to 1 because $p \in \text{ext}(c'')$ has become a failure state. This corresponds to the status of failure state p_2 in Figure 5.4.6. Then p may still serve as the *target* of failure arcs when c'' is considered. The computation of $\text{ar}(c'')$ therefore remains unchanged from its previous value. In our running example of Figure 5.4.5, $\text{ar}(c_{12})$ remains at 2.

If, however, $|\text{ext}(c'') \cap \text{dom } \mathbf{f}| > 1$, then $\text{ar}(c'')$ needs to be updated. There is now one less candidate in the extent of $\text{ext}(c'')$ than before which can have its arcs that are referenced in $\text{int}(c'')$ replaced by a failure arc. As a consequence, $\text{ar}(c'')$ has to decline by $|\text{int}(c'')| - 1$. If, as a result, $\text{ar}(c'')$ is no longer positive, then it should be removed from the set P .

Thus, the following can be inserted into Algorithm 11, just after the update of O , namely just after $O := O \setminus \{p\}$ if the elements of AR are to reflect not merely the concept arc redundancy values at the start of the algorithm, but throughout the execution of the algorithm.

Algorithm 12.

```
{ Adjust the arc redundancy of relevant concepts }
for each ( $c'' \in AR$ ) →
  if ( $p \in \text{ext}(c'')$ )  $\wedge$  ( $|\text{ext}(c'') \cap \text{dom } \mathbf{f}| > 1$ ) →
     $\text{ar}(c'') := \text{ar}(c'') - (|\text{int}(c'')| - 1)$ ;
    if ( $\text{ar}(c'') \leq 0$ ) →  $AR \setminus \{c''\}$  || ( $\text{ar}(c'') > 0$ ) → skip fi
  || ( $p \notin \text{ext}(c'')$ )  $\vee$  ( $|\text{ext}(c'') \cap \text{dom } \mathbf{f}| \leq 1$ ) → skip
fi
rof
```

In addition, the command **let** $q \in \text{ext}(c)$ in Algorithm 11 should be replaced by the following:

Algorithm 13.

```
if ( $|\text{ext}(c) \cap \text{dom } \mathbf{f}| \geq 1$ ) → let  $q \in (\text{ext}(c) \cap \text{dom } \mathbf{f})$ 
|| ( $|\text{ext}(c) \cap \text{dom } \mathbf{f}| = 0$ ) → let  $q \in \text{ext}(c)$ 
fi
```

Tracing the effect of these changes in Figures 5.4.1, 5.4.2 and 5.4.3, we note in the first iteration, t was arbitrarily chosen as p_1 . Thereafter, first p_2 (say) and then p_3 are installed as failure states. Since both are contained in $\text{ext}(c_{1234}) =$

$\{p_1, p_2, p_3, p_4\}$, we note that the arc redundancy of $c1234$ might change after each installation. After installing p_2 , $|ext(c1234) \cap f| = |\{p_2\}| = 1$ and so it is not necessary to update $ar(c1234)$. After installing p_3 , $|ext(c1234) \cap f| = |\{p_2, p_3\}| = 2$ and hence $ar(c1234)$ has to decrease by $(int(c1234)1) = (2-1) = 1$. Its revised value is therefore 2.

5.4.2 The role of cycles

This reduced arc redundancy of 2 ought to correspond exactly to the number of arcs to be removed in the next iteration. At that stage, we anticipate that a failure arc from each of p_1 and p_4 will be installed to either p_2 or p_3 , these latter two states being preferred target states, since they are already failure states. Figure 5.4.3 shows the status after installing a failure arc from p_4 to p_3 . (Note that p_2 could have been chosen.) Suppose we now try to install a failure arc from p_1 to p_3 and remove the relevant δ arcs, $\langle p_1, b, p_2 \rangle$ and $\langle p_1, c, p_4 \rangle$. The result would be a failure arc cycle between p_1 and p_3 . There would be no way of consuming b or c .

Notice that a cycle test has been built into Algorithm 11 but not discussed above. It has been designed to prevent the installation of failure arcs that that will result in a divergent failure cycle. For the time being, the work to be done is specified declaratively—i.e. algorithmic work to be done to keep track of cycles and their associated failure path alphabets has not been spelled out as part of the algorithm. This task is well known in general data structure theory. In the present context, there is a pleasing fact that a cycle cannot be longer than $|Q|$, and there cannot be any paths spiralling off from one cycle to start another—all cycles have to be disjoint, since f is a function. This will considerably simplify the task of identifying divergent cycles.

In the example being discussed, the cycle test discovers that $(P(p_3, p_1))$ is indeed true—there is a failure path from p_3 to p_1 . The alphabet of the path $\Sigma_{P(p_3, p_1)}$ is $\{a, b, c\}$, i.e. all symbols not consumed at p_3 . Furthermore, since $int(c)$ consists of pairs of observations, namely $\{\langle b, p_2 \rangle, \langle c, p_3 \rangle\}$ it can be viewed as a function whose domain is $\{b, c\}$. Since $(\Sigma_{P(p_3, p_1)} \cap \text{dom } int(c)) = \{b, c\} \neq \emptyset$, we would have a dangerous cycle if a failure arc were to be installed from p_1 to p_3 . The algorithm thus has to terminate in the state indicated in Figure 5.4.3.

The matter of whether or not to recompute arc redundancy is therefore in question. The above has illustrated that the metric may be distorted in the presence of cycles, even if recomputed as indicated above. Nevertheless, it should be noted that Algorithm 11 always runs through all elements of set P —the concepts initially determined as having arc redundancy. If, during the course of processing, some of those concept arc redundancies become inaccurate or drop to 0, the algorithm does no harm. It might do nothing more than eliminate the concept from P and treat the next concept. Any transformation it makes will be based on Theorem 5.2.11, and will thus guarantee the language equivalence. Empirical tests will be needed to discover contexts in which it is worth recomputing arc redundancy, and vice-versa.

5.4.3 Complexity

Recall from Section 5.3 that the number of concepts in lattice $\mathcal{SO}(\mathcal{D})$ is bound from above by $\mathcal{LB}(\Sigma, Q)$, giving a very rough upper bound for $|AR|$ in Algorithm 11. We expect the actual bound to be much lower than this, since it is not clear a state/out-transition lattice can reach the upperbound mentioned, and many concepts may have no arc redundancy and thus not end up in AR . Nevertheless, using that bound, and as $O \subseteq Q$, the outer **do** loop is executed at most $\mathcal{LB}(\Sigma, Q)$ times. The outer **for each** loop is executed at most $|Q|$ times, as both P and O are subsets of Q . The complexity of the guards of the **if** statement is bounded by the maximum of $|Q|$ (for failure path tracing) and $|\Sigma|$ (for checking intersection), while the inner **for each** loop has complexity at most $|\Sigma|$. Combining this gives $\mathcal{LB}(\Sigma, Q) \times |Q| \times \max(|Q|, |\Sigma|) \times |\Sigma|$ as a very coarse upper bound on Algorithm 11's time complexity.

5.5 A Lattice-Homomorphic Algorithm

In this section we present a second algorithm that constructs an FDFA from the cover graph relation of the state/out-transition lattice for a DFA.

5.5.1 Lattice-Homomorphic Algorithm Preliminaries

We rely on the following definitions to derive this algorithm.

Definition 5.5.1 (State equivalence in a DFA). According to Watson in [Wat10], given the DFA \mathcal{D} , and two states p and q in \mathcal{D} , if $\vec{\mathcal{L}}(\mathcal{D}, p) = \vec{\mathcal{L}}(\mathcal{D}, q)$, then p and q are equivalent. This property of p and q can also be expressed as $E(p, q)$.

□

Lemma 5.5.2 (State merging). For states p, q in DFA \mathcal{D} , if $E(p, q)$ then another DFA \mathcal{D}' can be formed such that $\mathcal{L}(\mathcal{D}') = \mathcal{L}(\mathcal{D})$ by merging p into q and keeping the rest of \mathcal{D} and \mathcal{D}' the same.

□

Definition 5.5.3 (State equivalence in a FDFA). Definition 5.5.1 can be extended to the context of FDFAs.

$E(p, q)$ holds for two states p, q in $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ if and only if $E(p, q)$ holds for the embedded DFA of \mathcal{F} (i.e. $(Q, \Sigma, \delta, F, s)$) and $\mathfrak{f}(p) = \mathfrak{f}(q)$ (i.e. if there is a failure arc from p to some state s , then there is also a failure arc from q to s)

□

Definition 5.5.4 (Addition of a new state to an FDFA using function exF). Given an FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$, a state $p \in Q$, a new state $r \notin Q$ and $X \subseteq out(p)$ (See the definition of the function $out(p)$ in Definition 2.4.24), we define a function $exF(\mathcal{F}, p, X, r)$ that returns $\mathcal{F}' = (Q', \Sigma, \delta', \mathfrak{f}', F', s)$, that is an updated version of \mathcal{F} , such that the elements Q' , \mathfrak{f}' , δ' and F' of \mathcal{F}' are formed as follows:

$$\begin{aligned} Q' &= Q \cup \{r\} \\ \mathfrak{f}' &= \mathfrak{f} \cup \{\langle p, r \rangle\} \\ \delta' &= \delta \cup \left(\bigcup_{a, q : \langle p, a, q \rangle \in X} \{\langle r, a, q \rangle\} \right) \setminus X \\ F' &= \begin{cases} F \cup \{r\} & \text{if } p \in F \\ F & \text{otherwise} \end{cases} \end{aligned}$$

Informally, exF modifies FDFA \mathcal{F} to form FDFA \mathcal{F}' by

- forming Q' from Q by adding a disconnected state r to Q .
- forming f' from f by adding a failure arc from p to r .
- forming δ' from δ by adding a transition $\langle r, a, q \rangle$ for every $\langle p, a, q \rangle \in X$, where X is a subset of the out-transitions of p and then removing the same X from δ .

□

Lemma 5.5.5 (Language preservation under exF). Given the FDFA $\mathcal{F} = (Q, \Sigma, \delta, f, F, s)$, a state $p \in Q$, a set of out transitions $X \subseteq out(p)$ and a new state $r \notin Q$ then

$$\mathcal{L}(exF(\mathcal{F}, p, X, r)) = \mathcal{L}(\mathcal{F})$$

Informally this lemma holds because the substitution of $\langle r, a, q \rangle$ for every $\langle p, a, q \rangle \in X$ using exF has the effect that every such q is still reachable via a failure from p to r and a normal transition from r to q and $\mathcal{L}(\mathcal{F}') = \mathcal{L}(\mathcal{F})$, i.e. the language of \mathcal{F} is the same as the language of \mathcal{F}' formed by applying exF to \mathcal{F} as described above.

□

5.5.2 Mapping DFA states to State/Out-transition Lattice objects

The algorithm derives the target FDFA (introduced in Definition 5.2.1 above) from the attributes, objects and concepts of the State/Out-transition Lattice of a given DFA . By definition, every object in the lattice corresponds to exactly one state of the original DFA and visa versa. This means that there is a 1-to-1 relationship between objects in the lattice and states in the input DFA. This relationship is illustrated by three figures below. Figure 5.5.1 shows an example DFA. Figure 5.5.2 shows the DFA State/Out-transition Lattice derived from the DFA in Figure 5.5.1 above. Figure 5.5.3 shows the DFA in Figure 5.5.1 superimposed on the lattice from Figure 5.5.2 .This diagram clearly shows the 1-on-1 relationship between the objects of the lattice and the states in the DFA .

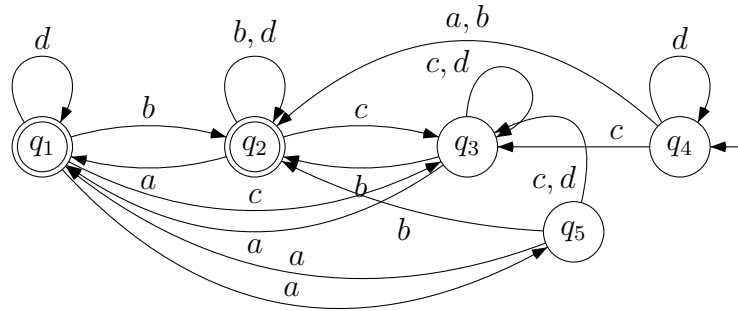


Figure 5.5.1: Input DFA

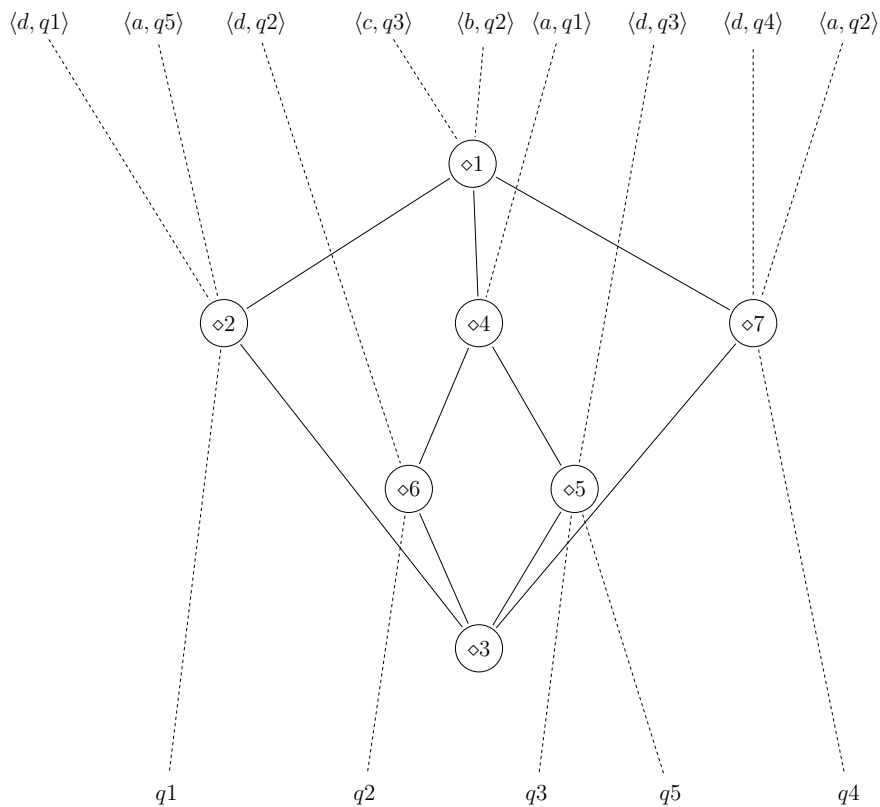


Figure 5.5.2: DFA State/Out-transition Lattice

5.5.3 Reducing the DFA based on own objects of the State/Out-transition Lattice of the input DFA

The algorithm also relies on a reduction step based on Lemma 5.5.2 given above. The reduction step involves a language preserving operation that

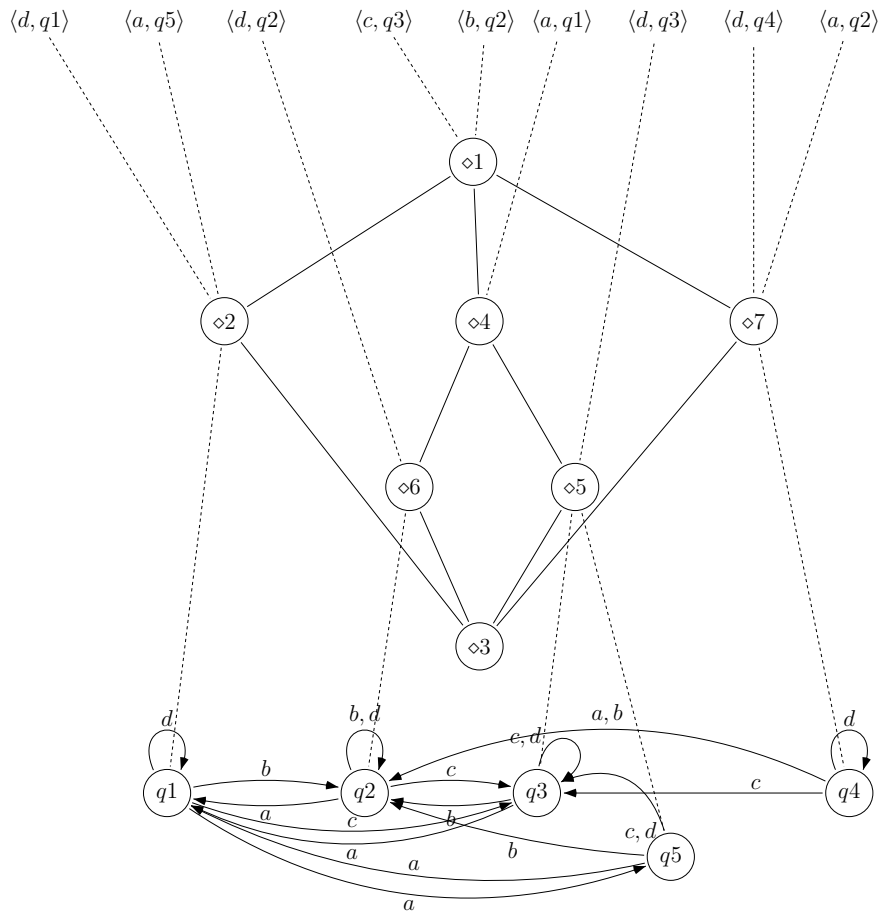


Figure 5.5.3: Input DFA states have a 1-to-1 relationship with corresponding State/Out-transition Lattice objects

merges those sets of states of the DFA or F DFA that share the same out-transitions. Consider a concept in the State/Out-transition Lattice of the input DFA that has more than one *own object*. The DFA's states corresponding to these own objects share the same out-transitions, these being specified by the concept's intent. Consequently, these states may be merged, the resulting DFA being language equivalent to the original one.

The set of states $\{q_3, q_5\}$ that form the own objects of concept $\diamond 5$ in Figure 5.5.3 is an example of such a set of *own objects*.

Algorithm 14 given below, reduces an input DFA based on the State/Out-transition Lattice of the input DFA. It uses the function *merge* that is assumed

to merge a set of states into one to produce a reduced version of a DFA. It takes a DFA and a set of states in the same DFA and returns a reduced version of the same DFA merging all given states into one and updating the transition function accordingly. This algorithm considers every concept in the State/Out-transition Lattice of the input DFA. If the concept has more than one own object, then the DFA states corresponding to these own objects are merged, and the revised DFA is returned.

Algorithm 14. *Merge states of a DFA based on its State/Out-transition Lattice*

```

{ pre ( $\mathcal{D}_r = \mathcal{D}$ ) }
{ invariant :  $\mathcal{L}(\mathcal{D}_r) = \mathcal{L}(\mathcal{D})$  }
for ( $c \in \mathcal{SO}(\mathcal{D})$ ) →
  if  $|\text{ownobj}(c)| > 1 \rightarrow \mathcal{D}_r = \text{merge}(\mathcal{D}_r, \text{ownobj}(c))$ 
   $|\text{ownobj}(c)| \leq 1 \rightarrow$  skip
  fi
  { invariant holds: see Lemma 5.5.2 }
rof;
{ post ( $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}_r)$ ) }

```

Figure 5.5.4 shows the reduced DFA produced by applying Algorithm 14 to the DFA shown in Figure 5.5.1 superimposed on the lattice from Figure 5.5.2

5.5.4 Embedding the reduced DFA into the State/ Out-transition Lattice of the initial input DFA

After creating the reduced DFA in the second step, we now have a 1-to-1 relationship between states of the reduced DFA and the concepts of the State/Out-transition Lattice of the initial input DFA that have *own objects*. We now create a new version of the input DFA called the *SO Lattice embedded reduced DFA* by simply replacing the id of each state of the reduced DFA with the id of its corresponding State/Out-transition Lattice concept.

Definition 5.5.6 (SO Lattice embedded reduced DFA). Given the reduced DFA $\mathcal{D}_r = (Q_r, \Sigma, \delta_r, F_r, s)$, derived from the DFA \mathcal{D} using Algorithm 14 and

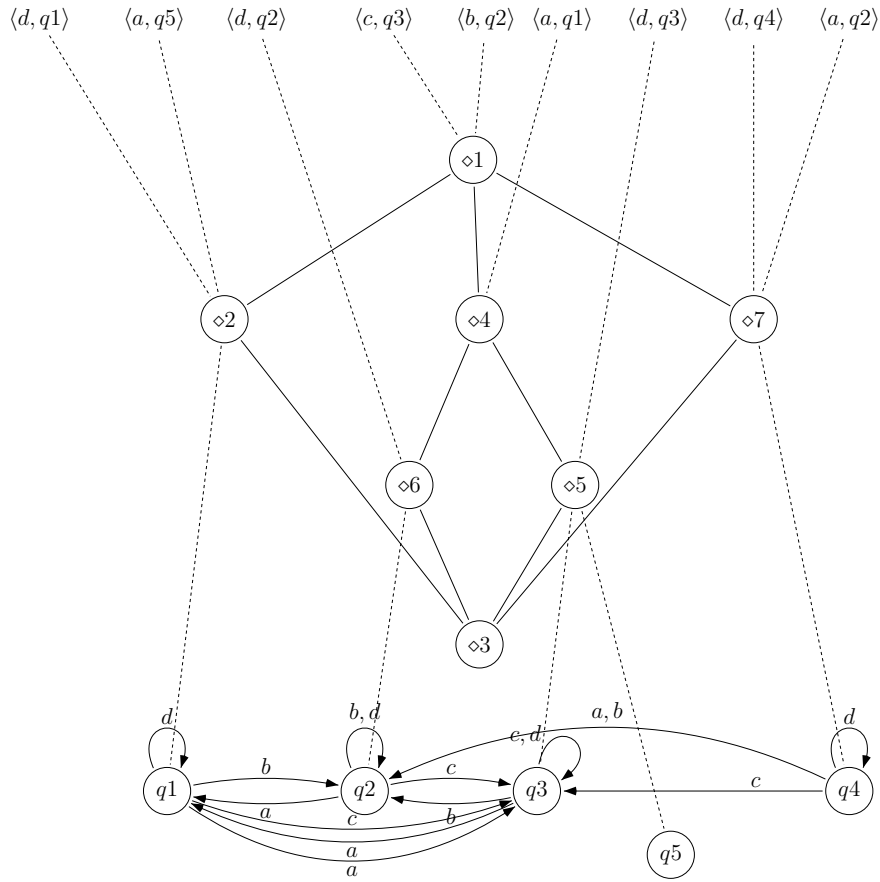


Figure 5.5.4: Reduced DFA formed by merging states in the same set of own objects of the input DFA’s State/Out-transition Lattice

the State/Out-transition Lattice $\mathcal{SO}(\mathcal{D})$, we define the function that gives *SO Lattice embedded reduced DFA*, denoted $soer(\mathcal{D}_r) = (Q_e, \Sigma, \delta_e, F_e, s_e)$ where

- $Q_e = (\bigcup q : q \in Q_r : \{\perp(q)\})$
- $\delta_e = (\bigcup q, a, p : \langle q, a, p \rangle \in \delta_r : \{\langle \perp(q), a, \perp(p) \rangle\})$
- $F_e = (\bigcup f : f \in F_r : \{\perp(f)\})$
- $s_e = \perp(s)$

□

Lemma 5.5.7 (Language preservation under *soer*). For the DFA $\mathcal{D}_e = soer(\mathcal{D}_r)$, where \mathcal{D}_r was derived from the DFA \mathcal{D} using Algorithm 14 and the State/Out-

transition Lattice $\mathcal{SO}(\mathcal{D})$, we can assert that $\mathcal{L}(\mathcal{D}_e) = \mathcal{L}(\mathcal{D}_r)$, since changing the labels of the states of a DFA does not affect the language of the DFA. Also, since $\mathcal{L}(\mathcal{D}_r) = \mathcal{L}(\mathcal{D})$, it follows that $\mathcal{L}(\mathcal{D}_e) = \mathcal{L}(\mathcal{D})$.

□

See Figure 5.5.5 for a depiction of the result of the third step of the algorithm.

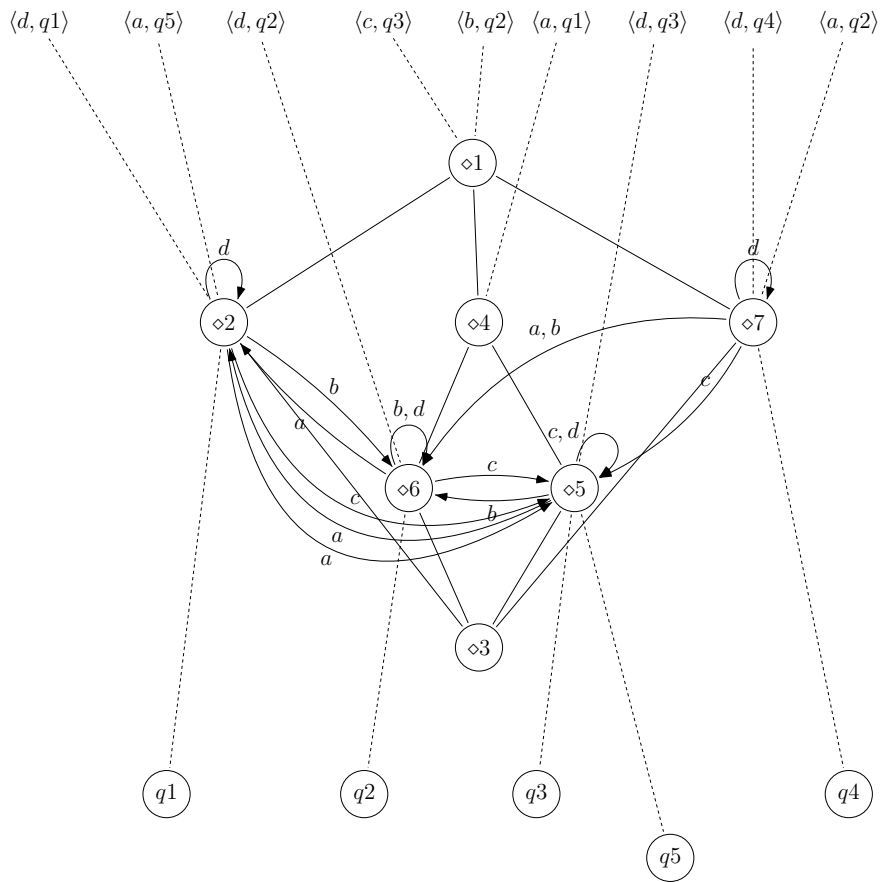


Figure 5.5.5: DFA derived using function *soer*

5.5.5 Initializing and Deriving the FDFA

Definition 5.5.8 (FDFA initialization). Given $\mathcal{D}_e = (Q_e, \Sigma, \delta_e, F_e, s_e) = soer(\mathcal{D}_r)$, such that \mathcal{D}_r was derived from the DFA \mathcal{D} using Algorithm 14 and the State/Out-transition Lattice $\mathcal{SO}(\mathcal{D})$, we define a function to create an initial FDFA, denoted $FDFA_{init}$ from \mathcal{D}_e as follows:

$$FDFA_{init}(\mathcal{D}_e) = (Q_e, \Sigma, \delta_e, \emptyset, F_e, s_e)$$

This means an FDFA derived using $FDFA_{init}$ has the embedded DFA \mathcal{D}_e derived from definitions above and the initial failure function is initialized to the empty set, i.e. $\mathbf{f} = \emptyset$.

□

Lemma 5.5.9 (Language preservation under $FDFA_{init}$). The $FDFA_{init}$ operation delivers an FDFA whose language is the same as that of the original DFA, since the empty failure function does not change the language from that of the DFA.

□

The algorithm also relies on the following DFA language preserving operation to further transform the FDFA:

Definition 5.5.10 (Atomic FDFA state addition operation). Given \mathcal{F} that had an initial value of $FDFA_{init}(\mathcal{D}_e)$, where $\mathcal{D}_e = (Q_e, \Sigma, \delta_e, F_e, s_e) = soer(\mathcal{D}_r)$, such that \mathcal{D}_r was derived from the DFA \mathcal{D} using Algorithm 14 and the State/Out-transition Lattice $\mathcal{SO}(\mathcal{D})$, we define a function to update \mathcal{F} , denoted $FDFA_{add}$. This function gives an updated FDFA for the following input parameters:

- FDFA \mathcal{F}
- a concept $c \in \mathcal{SO}(\mathcal{D})$
- a concept $d \in \mathcal{SO}(\mathcal{D})$

such that $\mathcal{F} = (Q_e, \Sigma, \delta_e, \mathbf{f}, F_e, s_e)$ and $c \prec d$.

It is defined as follows:

$$FDFA_{add}(\mathcal{F}, c, d) = exF(\mathcal{F}, id(c), int(c) \setminus int(d), \langle id(c), id(d) \rangle)$$

The function exF has been defined in Definition 5.5.4 above.

□

Lemma 5.5.11 (Language preservation under $FDFAadd$). The operation preserves the language of the given FDFA, i.e.

$\mathcal{L}(FDFAadd(\mathcal{F}, c, d)) = \mathcal{L}(exF(\mathcal{F}, id(c), int(c) \setminus int(d), \langle id(c), id(d) \rangle)) = \mathcal{L}(\mathcal{F})$ (see Lemma 5.5.5). The new state added to the resultant FDFA is labeled $\langle id(c), id(d) \rangle$ as a reference to the two concepts that lead to the creation of the new state in order to simplify the explanation of further steps below.

□

As an example, see Figure 5.5.6 that depicts the additional state and transitions for $FDFAadd(\mathcal{F}, \diamond 2, \diamond 1)$ applied to the result of the third step of the algorithm.

The algorithm initializes the FDFA using the function $FDFAinit$ and then uses the $FDFAadd$ operation to add an additional state and respective transitions and failure transitions for every pair $\langle c, d \rangle$, such that

- c is a concept in $\mathcal{SO}(\mathcal{D})$ that has more than one own objects
- and d is a concept in $\mathcal{SO}(\mathcal{D})$ that has one or more own attributes
- and $c \in descendants(\mathcal{SO}(\mathcal{D}), d)$

A function that formalizes this set $c \in \mathcal{SO}(\mathcal{D})$ for a given d is defined below.

Definition 5.5.12 (Set of concepts to process with a concept). For a given $d \in \mathcal{SO}(\mathcal{D})$ where $\mathcal{SO}(\mathcal{D})$ is the State/Out-transition Lattice derived from \mathcal{D} , we define the function $to-process$ that gives the set of concepts to be processed with d by the inner loop of the algorithm. It is defined as

$$to-process(\mathcal{SO}(\mathcal{D}), d) = \left(\bigcup c : c \in descendants(\mathcal{SO}(\mathcal{D}), d) \wedge |ownobj(c)| > 1 : \{c\} \right)$$

□

Figure 5.5.7 depicts the result of firstly invoking $to-process(\mathcal{SO}(\mathcal{D}), \diamond 1)$, where $\mathcal{SO}(\mathcal{D})$ is the FDFA depicted in Figure 5.5.2, and then secondly applying the function $FDFAadd$ to each element of the resulting set of pairs.

Every additional state $\langle c, d \rangle$ (and corresponding transitions and failure transitions) that the algorithm adds (using the operation $FDFAadd$) is also merged

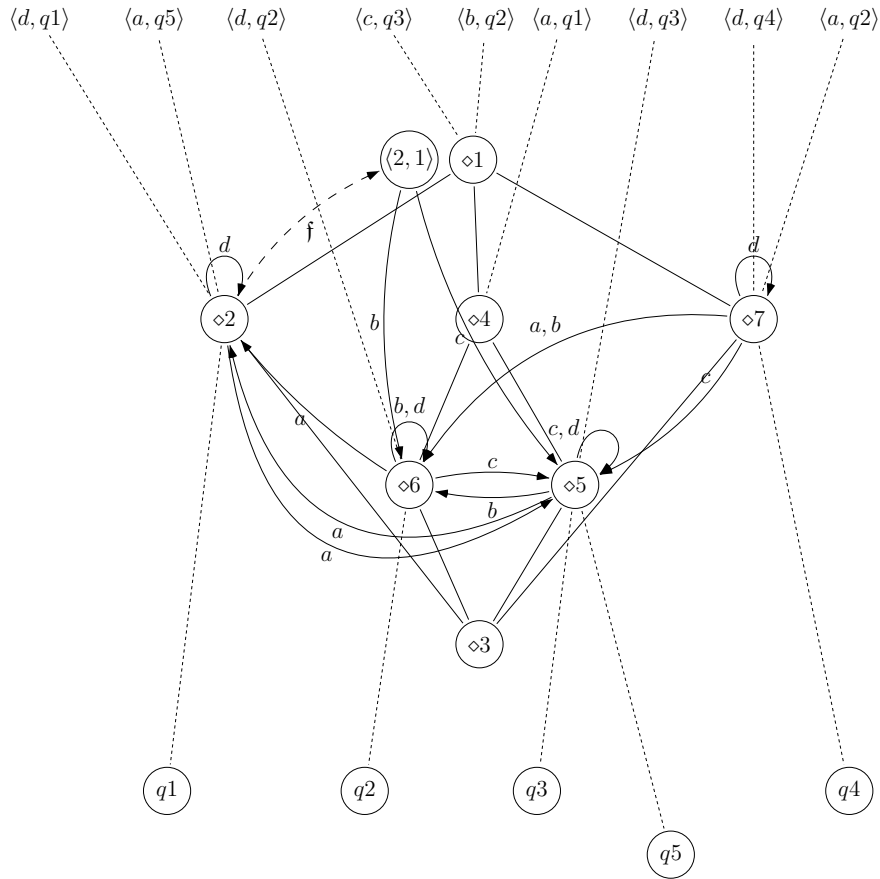


Figure 5.5.6: Transforming F DFA: $F DFA_{add}(\mathcal{F}, \diamond 2, \diamond 1)$

into a common state labelled d using a function f -merge that is assumed to merge a set of states of a F DFA into one to produce a reduced version of the F DFA. f -merge takes a F DFA and a set of states in the same F DFA and returns a reduced version of the same F DFA by merging all given states into one and updating the transition function and the failure function accordingly. This merging step preserves the language as every new $\langle c, d \rangle$ has exactly the same out transitions and can therefore be merged without affecting the language of the F DFA. See Lemma 5.5.2. See Figure 5.5.8 for a depiction of the result of the merging step applied after every state shown in the example in Figure 5.5.7 has been added.

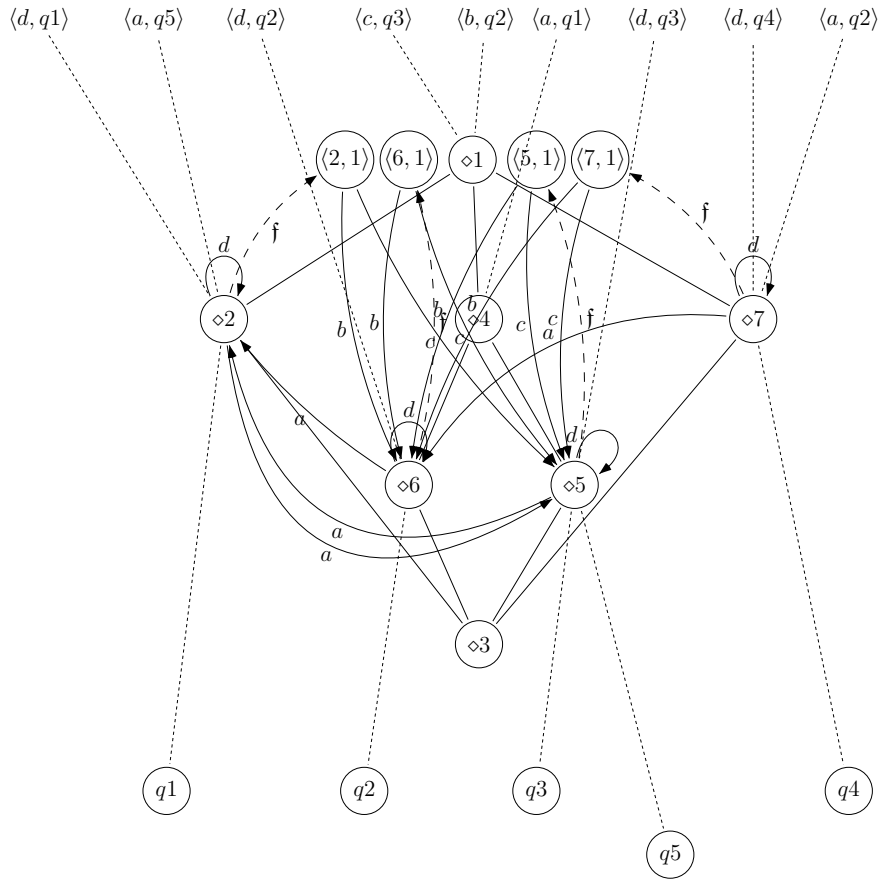


Figure 5.5.7: Transforming FDFA: $FDFA_{add}(\mathcal{F}, c, d)$ applied to all $\langle c, d \rangle \in \{\langle \diamond 2, \diamond 1 \rangle, \langle \diamond 6, \diamond 1 \rangle, \langle \diamond 5, \diamond 1 \rangle, \langle \diamond 7, \diamond 1 \rangle\}$

5.5.6 Lattice-Homomorphic Algorithm: Putting it all together

We now give the Lattice-Homomorphic Algorithm as a composition of the ideas introduced above.

Algorithm 15. *Lattice-Homomorphic Algorithm to derive a FDFA from a DFA*

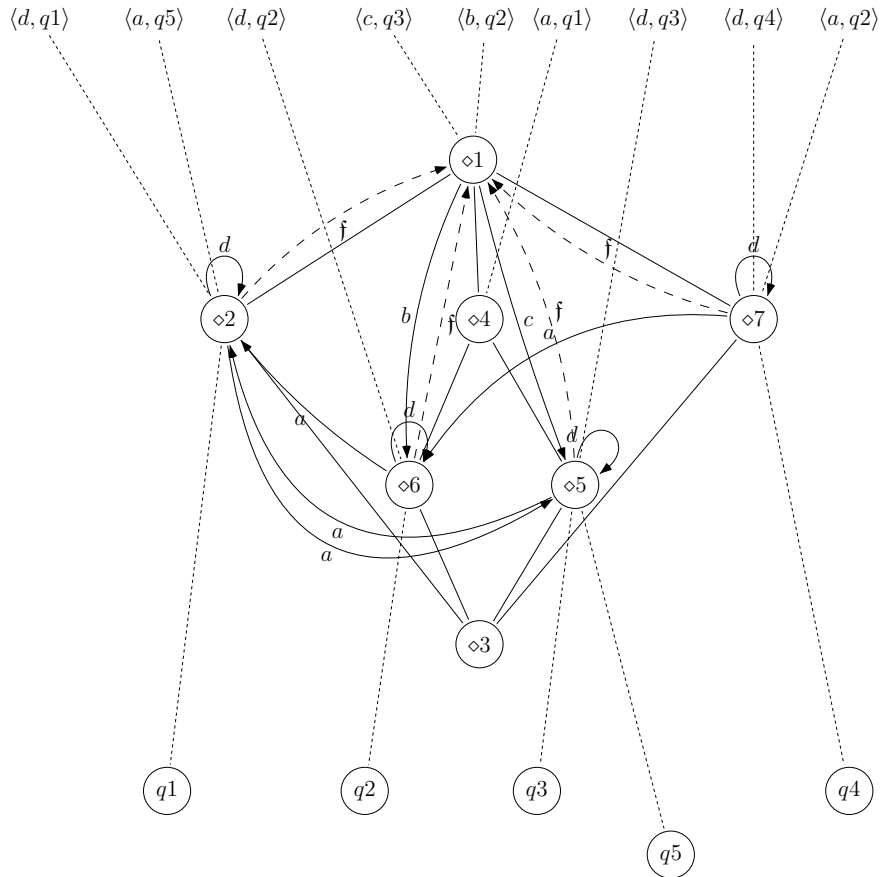


Figure 5.5.8: Transforming FDFA: merging states created by $FDFAadd(\mathcal{F}, c, d)$ as shown in Figure 5.5.7

```

{ ( The function pre-merge implements Algorithm 14 defined above) }
func pre-merge( $\mathcal{D}$ )
  result :=  $\mathcal{D}$ 
  { (language-preservation :  $\mathcal{L}(\text{result}) = \mathcal{L}(\mathcal{D})$ ) }
  for ( $c \in \mathcal{SO}(\mathcal{D})$ )  $\rightarrow$ 
    if  $\text{ownobj}(c) \neq \emptyset \rightarrow \text{result} := \text{merge}(\text{result}, \text{id}(c), \text{ownobj}(c))$ 
       $\text{ownobj}(c) = \emptyset \rightarrow$  skip
    fi

```



```

        { (language-preservation holds: see Lemma 5.5.2) }
    rof;
    { post ( $\mathcal{L}(\text{result}) = \mathcal{L}(\mathcal{D})$ ) }
    return result
cnuf

 $\mathcal{D}_r := \text{pre-merge}(\mathcal{D});$ 
{ ( $\mathcal{L}(\mathcal{D}_r) = \mathcal{L}(\mathcal{D})$ : See post condition of function pre-merge) }

 $\mathcal{D}_e := \text{soer}(\mathcal{D}_r);$ 
{ ( $\mathcal{L}(\mathcal{D}_e) = \mathcal{L}(\mathcal{D}_r)$ : See Lemma 5.5.7) }

 $\mathcal{F} := \text{FDFAinit}(\mathcal{D}_e);$ 

{ ( $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{D}_e)$  : See Lemma 5.5.9) }

for ( $d \in \mathcal{SO}(\mathcal{D})$ ) →
    if  $|\text{ownatt}(d)| > 0$  →
        for ( $c \in \text{to-process}(\mathcal{SO}(\mathcal{D}), d)$ ) →
             $\mathcal{F} := \text{FDFAadd}(\mathcal{F}, c, d)$ 
            { ( $\mathcal{L}(\mathcal{F})$  does not change: See Lemma 5.5.11) }
             $\mathcal{F} := \text{f-merge}(\mathcal{F}, \text{id}(d), \{\text{id}(c)\})$ 
            { ( $\mathcal{L}(\mathcal{F})$  does not change: See Lemma 5.5.2) }
        rof
    fi  $|\text{ownatt}(d)| = 0$  → skip
rof
```

See Figure 5.5.9 for the final FDFA derived using the Lattice Homomorphic Algorithm. Figure 5.5.10 shows the final derived FDFA without the underlying State/Out Transition Lattice of the input DFA. Note that the number of transitions in this case has been reduced from 20 down to 9 at the cost of 6 failure transitions.

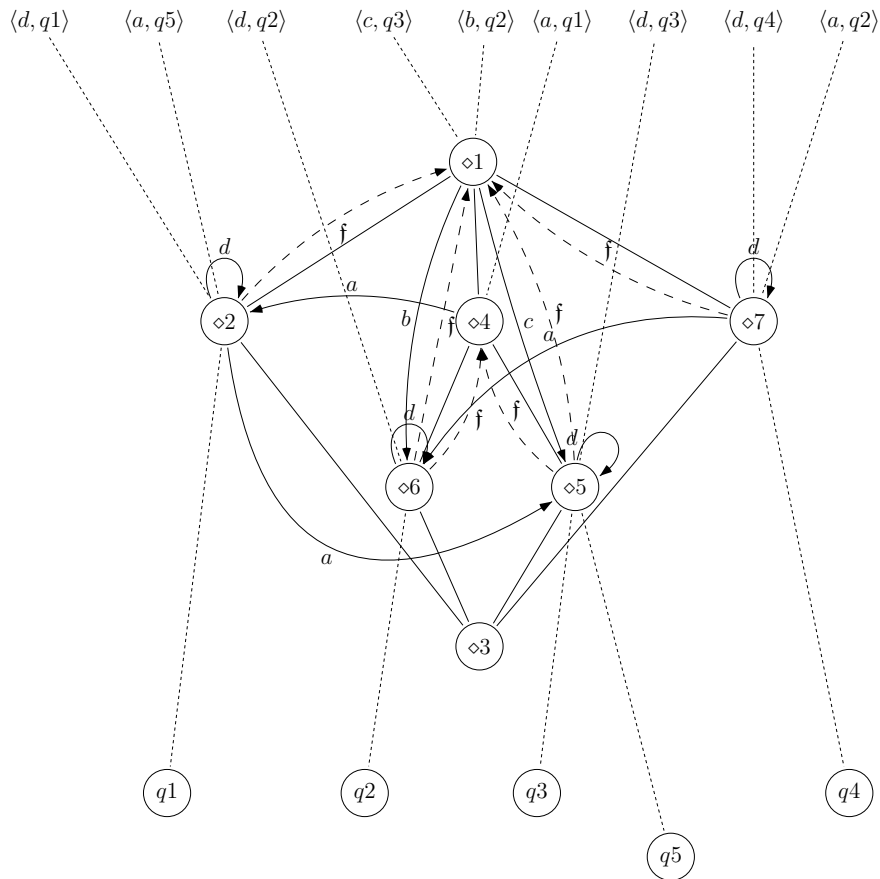


Figure 5.5.9: Final result of applying the Lattice Homomorphic Algorithm to the DFA in Figure 5.5.1

5.6 The Next Steps

It has been shown above how to derive an $\mathcal{F} \in \mathcal{E}_{\mathcal{F}}(\mathcal{L})$, given a $\mathcal{D} \in \mathcal{E}_{\mathcal{D}}(\mathcal{L})$. The algorithm has been designed to terminate in a state where \mathcal{F} has fewer transitions than \mathcal{D} . However, the number of states in \mathcal{F} will be the same as in \mathcal{D} .

The tradeoff between FDFA storage size reduction versus processing speed is has been investigated by Nxumalo in [Nxu16].

We know from classical FA theory that there will be at least one $\mathcal{D}_{\min} \in \mathcal{E}_{\mathcal{D}}(\mathcal{L})$ that is minimal—i.e. all other DFAs in $\mathcal{E}_{\mathcal{D}}(\mathcal{L})$ have the same number of states

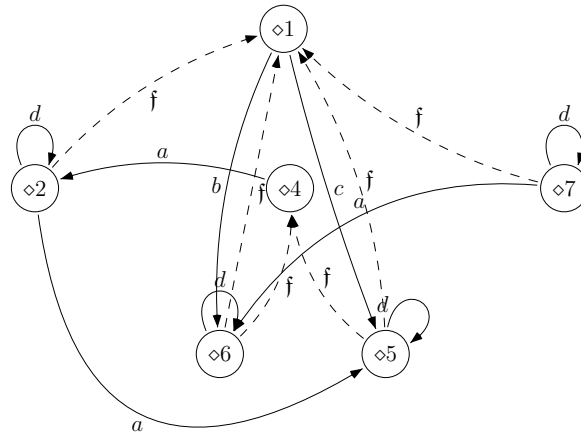


Figure 5.5.10: Final result of applying the Lattice Homomorphic Algorithm to the DFA in Figure 5.5.1 without the underlying State/Out Transition Lattice

or more, and the same number of transitions or more than \mathcal{D}_{\min} . This minimal DFA is unique up to isomorphism.

Clearly there will be one or more minimal $\mathcal{F} \in \mathcal{E}_{\mathcal{F}}(\mathcal{L})$ with respect to the number of states and one or more minimal $\mathcal{F} \in \mathcal{E}_{\mathcal{F}}(\mathcal{L})$ with respect to the number of transitions⁴.

All of the foregoing has stimulated a number of future research questions and ideas relating to FDFAs, their properties, their relation to DFAs, and their

⁴For the purposes of the present discussion, the number of transitions includes both δ transitions and f transitions. We note in passing that the latter are bound from above by the number of states, and that they require less storage than the former.

construction:

- If given a state-minimal DFA, the “greedy” algorithm will derive an F DFA with the same number of states but with fewer or equally many transitions. An F DFA in $\mathcal{E}_{\mathcal{F}}(\mathcal{L})$ will always have at least as many states as a state-minimal DFA in $\mathcal{E}_{\mathcal{D}}(\mathcal{L})$, since converting such an F DFA back to a DFA would add no new states (only add transitions), and that would then yield a ‘smaller’ minimal DFA. (An algorithm for such conversions is easily derived and omitted here.)
- Does a minimal F DFA necessarily have both a minimal number of states and a minimal number of transitions, as is the case for minimal DFAs?
- Are the members of the set of minimal F DFAs (whether with respect to states or transitions) homomorphic, as is the case for minimal DFAs?
- What algorithms can be derived in case changes to Q, s, F in the DFA to F DFA transformation *are* allowed?
- There are two variants of the Greedy algorithm, depending on whether or not the arc redundancy is recomputed at each step. It seems unlikely that the algorithm will guarantee arc-minimal optimality if arc redundancy is not recomputed. Even if arc redundancy is recomputed, the question of whether or not arc-minimality can be guaranteed is a matter of conjecture.
- Another currently unaddressed problem is to find a general algorithm for deriving an F DFA directly from a given regular expression.
- Yet another interesting question is how the addition or removal of failure transitions can be guided by information about (expected) hotspots among failure transitions, e.g. by using information about the automaton’s language and expected characteristics of texts to be processed with it.
- Finally, it would be of interest to characterise the embedded DFA of an F DFA.

Chapter 6

FCA based NFA to DFA reduction and A DFA pattern matching using FCA

6.1 Introduction

In this chapter I discuss ideas based on the intersection of FCA and Automata that could be explored in further research. In the first section I propose the outline of new algorithms based on FCA to reduce an NFA to an equivalent DFA. In the second section I propose new *Position Encoded Pattern Lattice* (PEPL) based algorithms for deriving an A DFA from a given set of keywords. In the same section I also introduce *Reverse Position Encoded Pattern Lattices* (RPEPLs) and an algorithm to derive an A DFA from a RPEPL of a set of keywords. The intuitions, algorithms and illustrative examples related to these ideas are presented. I also provide specific questions or hypotheses to explore in further research throughout the chapter.

The layout of the chapter is as follows:

- An FCA-based algorithm that transforms NFAs to language equivalent DFAs
- Algorithms for deriving A DFAs from PEPLs and RPEPLs

6.2 An FCA-based algorithm that transforms NFAs to language equivalent DFAs

6.2.1 Introduction

The outline of an algorithm to transform a non-deterministic finite automaton to a deterministic automaton using FCA is suggested.

However, I do not provide a proof that the language recognised by the resultant DFA created by this algorithm is exactly the same as the language recognised by the input NFA. A formal proof (or disproof) needs to be developed as future work. Further work also needs to be done to adapt the algorithm to handle NFAs that contain ϵ -transitions.

6.2.2 Algorithm Skeleton

6.2.2.1 Problem Definition

The algorithm needs to create a DFA D for the NFA N such that the following two predicates hold:

1. The language recognized by N is the same as the language recognized by D , i.e. $\mathcal{L}(D) = \mathcal{L}(N)$ and
2. D does not contain any useless states.

6.2.3 The algorithm overview

The proposed algorithm consists of two main steps that the following sections elaborate on. These steps are, in brief:

1. Derive a formal context K_N from a given NFA N .
2. The concept lattice $\mathfrak{N}(K_N)$ and objects in K_N are then used to generate the resultant DFA.

6.2.4 From NFA to NFA Concept Lattice

As was noted in Chapter 2, the signature of the transition function δ of an NFA N is defined as a mapping from state-symbol pair to a set of states. Formally:

$$\delta \in Q_N \times \Sigma \rightarrow \mathcal{P}(Q)$$

For the purposes of this chapter, any state symbol pair for which there is no out-transition, I assume that δ returns the empty set.

There are various algorithms that transform NFAs to equivalent DFAs. One such algorithm is called the *subset construction algorithm*, due to Rabin and Scott in [Sco59].

Example 6.2.1 (Example NFA). Figure 6.2.1 shows an example of an ϵ -free NFA. Its language, expressed as a regular expression, is $a^*b+a(a|b)^*$.

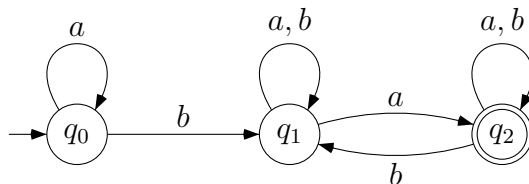


Figure 6.2.1: Example NFA

Figure 6.2.2 shows the DFA created by applying the *subset construction algorithm* on the NFA in Figure 6.2.1. Note that its language is identical to that of the NFA, namely $a^*b+a(a|b)^*$.

In order to derive a lattice from an NFA, the first step is to create a formal context from the NFA. There are various ways that the elements of an NFA can be used to form a formal context. For the purpose of the algorithm suggested here, the following approach is used:

Definition 6.2.2 (Set to set transition function). Given NFA $N = \langle Q, \Sigma, \delta, s, F \rangle$, the transition function $\Delta \in \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$ returns the set of states that

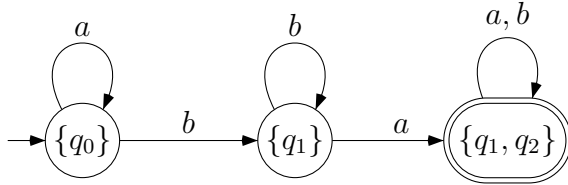


Figure 6.2.2: Example DFA from NFA using the *subset construction algorithm*

result from applying the NFA transition function to every state in a given set of states and a given symbol. Formally:

$$\Delta(R, \alpha) = \left(\bigcup r : r \in R : \delta(r, \alpha) \right)$$

□

In terms of the NFA's transition graph, $\Delta(R, \alpha)$ is the set of destination states having in-transitions on α from at least one state in R .

Definition 6.2.3 (NFA Formal Context). Given NFA $N = \langle Q, \Sigma, \delta, s, F \rangle$, the formal context of N is defined as $K_N = \langle G_N, Q, I_N \rangle$, where the set of objects G_N and incidence relation I_N are defined as follows:

- $G_N = \left(\bigcup R, \alpha : R \subseteq Q \wedge \alpha \in \Sigma : \{ \langle R, \alpha \rangle \} \right)$
- $I_N = \left(\bigcup \langle R, \alpha \rangle : \langle R, \alpha \rangle \in G_N : \{ \langle \langle R, \alpha \rangle, \Delta(R, \alpha) \rangle \} \right)$

□

Thus, each state of the input NFA is an attribute of the formal context. The objects of the formal context are all possible pairs $\langle R, \alpha \rangle$, such that R is a subset of the states of the input NFA.

The incidence relation of the formal context is the set of pairs $\{ \langle \langle R, \alpha \rangle, \Delta(R, \alpha) \rangle \}$

for all objects $\langle R, \alpha \rangle$ in the formal context.

Example 6.2.4 (NFA Formal Context). The NFA Formal Context for the NFA in Figure 6.2.1 is depicted as a cross table in Table 6.1.

| | q_0 | q_1 | q_2 |
|--|-------|-------|-------|
| $\langle \{\}, a \rangle$ | | | |
| $\langle \{\}, b \rangle$ | | | |
| $\langle \{q_0\}, a \rangle$ | 1 | | |
| $\langle \{q_0\}, b \rangle$ | | 1 | |
| $\langle \{q_0, q_1\}, a \rangle$ | 1 | 1 | 1 |
| $\langle \{q_0, q_1\}, b \rangle$ | | 1 | |
| $\langle \{q_0, q_2\}, a \rangle$ | 1 | | 1 |
| $\langle \{q_0, q_2\}, b \rangle$ | | 1 | 1 |
| $\langle \{q_1\}, a \rangle$ | | 1 | 1 |
| $\langle \{q_1\}, b \rangle$ | | 1 | |
| $\langle \{q_1, q_2\}, a \rangle$ | | 1 | 1 |
| $\langle \{q_1, q_2\}, b \rangle$ | | 1 | 1 |
| $\langle \{q_2\}, a \rangle$ | | | 1 |
| $\langle \{q_2\}, b \rangle$ | | | 1 |
| $\langle \{q_0, q_1, q_2\}, a \rangle$ | 1 | 1 | 1 |
| $\langle \{q_0, q_1, q_2\}, b \rangle$ | 1 | 1 | 1 |

Table 6.1: NFA Formal Context

Definition 6.2.5 (NFA Concept Lattice). By Definition 6.2.3 and Definition 2.3.11, the concept lattice of NFA $N = \langle Q, \Sigma, \delta, s, F \rangle$, is defined as $\mathfrak{N}(K_N)$.

□

Example 6.2.6 (NFA Concept Lattice). Figure 6.2.3 shows the line diagram of the NFA concept lattice for the NFA in Figure 6.2.1.

6.2.5 From NFA Concept Lattice to DFA

Given the above definitions, the algorithm to create a DFA $D = \langle Q', \Sigma, \delta', s', F' \rangle$ for the NFA $N = \langle Q, \Sigma, \delta, s, F \rangle$ is given in Algorithm 16 below, but needs a post processing step to remove useless paths. Algorithm 17 does the same, but does not require a post processing step.

These algorithms are based on the following conjectures:

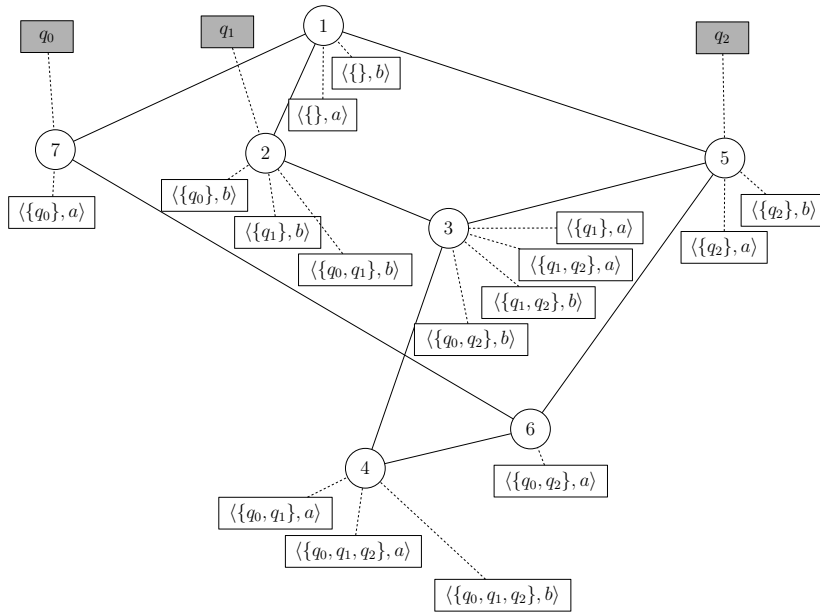


Figure 6.2.3: Line Diagram of the NFA Concept Lattice for the NFA in Figure 6.2.1

- A subset of the constructed concept lattice can be associated with the DFA's states.
- If p is such a concept and $\perp(\langle \text{int}(p), \alpha \rangle) = \top$ then the DFA state associate with p does not have an out-transition on α .
- If p is such a concept and $\perp(\langle \text{int}(p), \alpha \rangle) \neq \top$ then $\perp(\langle \text{int}(p), \alpha \rangle)$ is another such concept and that the DFA has a transition on α from the former to the latter associated DFA state.
- The resulting DFA is homomorphic with the DFA derived from the subset construction algorithm.

Algorithm 16.

$\{$ Create NFA formal context based on above definitions $\}$
 $G_N := (\bigcup R, \alpha : R \subseteq Q \wedge \alpha \in \Sigma : \{\langle R, \alpha \rangle\})$
 $I_N := (\bigcup R, \alpha : \langle R, \alpha \rangle \in G_N : \{\langle \langle R, \alpha \rangle, \Delta(R, \alpha) \rangle\})$
 $K_N := \langle G_N, Q, I_N \rangle$
 $\{$ Note that lattice operations used in statements below are assumed to $\}$

{ apply to the lattice $\mathfrak{N}(K_N)$ }

$Q', \delta', F', s' := \emptyset, \emptyset, \emptyset, \top(s)$

$C := (\bigcup c : c \in \mathfrak{N}(K_N) \wedge |\text{ownobj}(c)| > 0 : \{c\}) \cup \{s'\}$

for $p' \in C \rightarrow$

$R := \text{int}(p')$

for $\alpha \in \Sigma \rightarrow$

$q' := \perp(\langle R, \alpha \rangle)$

if $(q' \neq \top) \rightarrow$

$Q' := Q' \cup \{q'\}$

$\delta' := \delta' \cup \{\langle \langle p', \alpha \rangle, q' \rangle\}$

if $((R \cap F) \neq \emptyset) \rightarrow F' := F' \cup \{q'\}$

 || $((R \cap F) = \emptyset) \rightarrow$ **skip**

fi

 || $(q' = \top) \rightarrow$ **skip**

fi

rof

rof

As mentioned above, Algorithm 16 requires a post processing step to remove useless paths. This means that it will generate a graph, say X , that contains a subgraph, Y , that is a DFA whose language is equivalent to the original NFA. The question however, is: What is the nature of $Z = X - Y$? Is Z guaranteed to be just a set of unconnected states, not reachable from the start state? Could Z contain one or more *subgraphs*, not reachable from the start state, where the subgraph contains multiple connected states, perhaps even having final states? These are all questions to be answered in future work.

Algorithm 17.

```

 $s', \delta' := \top(s), \emptyset$ 
 $TD := \{s'\} \{ \text{Known DFA states still needing out-transitions} \}$ 
 $Q' := \emptyset \{ \text{All out-transitions of these DFA states have been set} \}$ 
if  $(s \in F) \rightarrow F' := \{s'\} \parallel (s \notin F) \rightarrow F' := \emptyset$  fi
do  $(TD \neq \emptyset) \rightarrow$ 
  select  $p' \in TD$ 
   $R' := \text{int}(p')$ 
   $\{ \text{For each } \alpha \text{ explore path from } p' \text{ that traces } \alpha^* \}$ 
  for  $\alpha \in \Sigma \rightarrow$ 
     $p, R := p', R'$ 
     $q := \perp(\langle R, \alpha \rangle)$ 
    do  $(q \neq \top) \wedge (\langle p, \alpha \rangle, q) \notin \delta' \rightarrow$ 
       $\delta' := \delta' \cup \{ \langle p, \alpha \rangle, q \}$ 
      if  $(q \notin Q') \rightarrow$ 
         $TD := TD \cup \{q\}$ 
        if  $((R \cap F) \neq \emptyset) \rightarrow F' := F' \cup \{q\}$ 
         $\parallel ((R \cap F) = \emptyset) \rightarrow$  skip
      fi
     $\parallel (q \in Q') \rightarrow$  skip
    fi
     $R := \text{int}(q)$ 
     $p := q$ 
     $q := \perp(\langle R, \alpha \rangle)$ 
  od
rof
 $TD, Q' := TD \setminus \{p'\}, Q' \cup \{p'\}$ 
od

```

Example 6.2.7 (DFA From NFA Lattice). To illustrate the application of Algorithm 16, see Figure 6.2.4. The diagram shows the DFA that the algorithm creates from the input NFA concept lattice for the NFA in Figure 6.2.1.

In this diagram, the resultant DFA is superimposed onto the input NFA lattice to clearly show how concepts from the NFA concept lattice become nodes in the DFA. Curved links between pairs of nodes depict the transition function of the DFA.

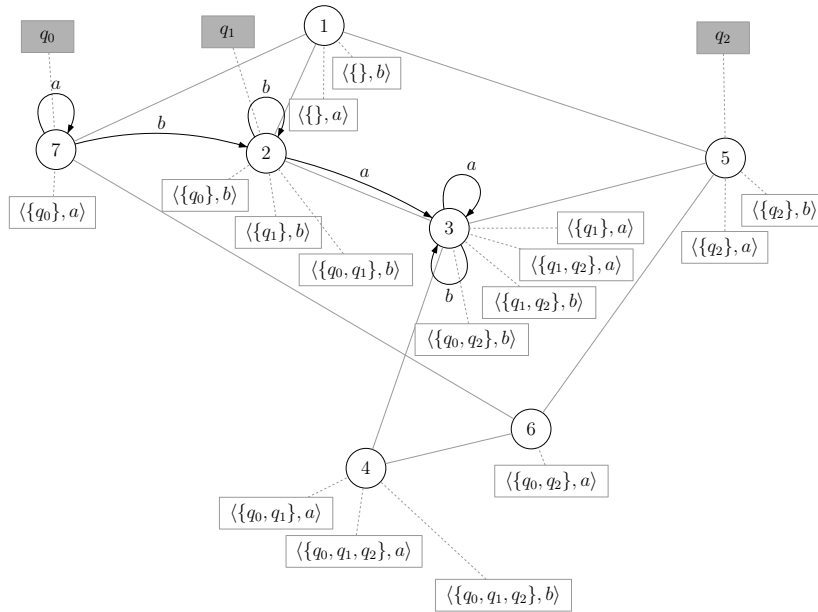


Figure 6.2.4: DFA from NFA Concept Lattice

It is instructive to compare the DFA shown in Figure 6.2.2 to verify that the result of applying Algorithm 17 on the NFA in Figure 6.2.1 is isomorphic to the result of applying the *subset construction algorithm* on the same NFA.

6.2.6 Conclusion

This section provided the outline of a new algorithm to transform a given NFA to its language equivalent DFA. The algorithm relies on creating a concept lattice from the input NFA and then forms the output DFA using various lattice operations on the intent of every concept and relevant objects. I believe this algorithm may be an interesting alternative to existing NFA to DFA reduction algorithms.

6.3 ADFAs for pattern matching using FCA

6.3.1 Overview

In this section I explore three approaches to encode a set of keywords P into a formal context and respective concept lattice that is then traversed to generate a DFA for P . The first such encoding is based on all prefixes of P . The second approach is based on the position encoding of P as introduced in Chapter 4. The third approach is based on the *reverse* position encoding of P . For every one of these encodings and corresponding lattices, ideas for further research are proposed. The layout of this into subsections is as follows:

- **Prefix lattices.** I also define the formal context and concept lattice formed using all prefixes of P and suggest that a *trie* derived from such a lattice is isomorphic to the classical *trie* of P . The definition of a classical trie due to Watson can be found in [Wat10].
- **PEPL based ADFAs** In this subsection I provide the outline of two algorithms that derive a *PEPL based ADFA* of a set of keywords and compare the size of PEPL based ADFAs to the *trie* derived from the prefix lattice of the same set of keywords.
- **RPEPL based ADFAs** In this subsection I propose a solution to the so-called *position misalignment problem* that uses *Reverse Position Encoded Pattern Lattices* (RPEPLs).

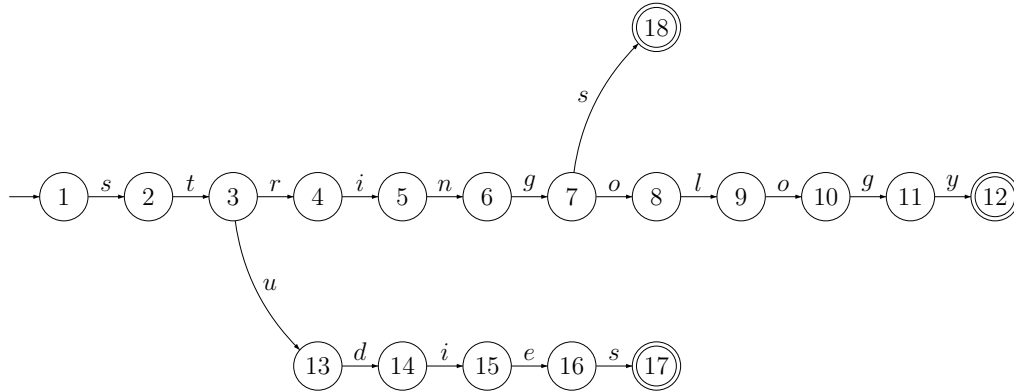
6.3.2 Prefix lattices

6.3.2.1 Introduction to tries

In classical multiple string pattern matching, the so-called *trie* of a set of keywords is often extended into a *DFA* that supports a specific matching algorithm.

Definition 6.3.1 (Trie). A DFA T is a *trie* if and only if the transition graph of T is a tree rooted at the start state of T . \square

Definition 6.3.2 (Trie of a set of keywords). Given a set of words P , $\mathcal{T}(P)$ is a function that derives a *trie* T such that the language of T is equal to P . T may also be referred to as a prefix tree.

Figure 6.3.1: The *trie* of a set of keywords

□

Example 6.3.3 (The *trie* of a set of keywords). Consider the set of keywords $P = \{\text{stringology}, \text{studies}, \text{strings}\}$. The *ADFA* representing the function $\mathcal{T}(P)$ of the *trie* of P is shown in Figure 6.3.1.

□

6.3.2.2 A lattice of prefixes

In this section I compare the structure of a formal concept lattice based on all the prefixes of a set of keywords with the *trie* of the same set of keywords. I suggest that the Hasse diagram (excluding the top and bottom nodes) of such a lattice is isomorphic to the *trie* created by the same set of keywords.

Definition 6.3.4 (The formal context and concept lattice of the (closed) set of keyword prefixes). I now define the formal concept lattice on the (closed) set of prefixes of a set of keywords P , as the concept lattice of a formal context whose set of objects *and* set of attributes are the same, namely the set of all prefixes of P - denoted by $\mathbf{pref}(P)$. Their incidence relation maps each object $g \in \mathbf{pref}(P)$ to the set of prefixes of g , i.e. to $\mathbf{pref}(g)$. Formally, the formal context of the closed set of prefixes of a set of keywords P is defined by:

$$\mathbb{K}_{\text{pref}} = \langle \text{pref}(P), \text{pref}(P), I_{\text{pref}}(P) \rangle$$

where

$$I_{\text{pref}}(P) = \left(\bigcup g : g \in \text{pref}(P) : \{ \langle g, \text{pref}(g) \rangle \} \right)$$

The concept lattice of the closed set of prefixes of a set of keywords P is then defined by:

$$\mathfrak{B}_{\text{pref}}(P) = \underline{\mathfrak{B}}(\mathbb{K}_{\text{pref}}) = \underline{\mathfrak{B}}(\langle \text{pref}(P), \text{pref}(P), I_{\text{pref}}(P) \rangle)$$

□

Example 6.3.5 (The *formal context* of prefixes of a set of keywords). The same set of keywords $P = \{\textit{stringology}, \textit{studies}, \textit{strings}\}$, used in the example above, yields the formal context $\langle \text{pref}(P), \text{pref}(P), I_{\text{pref}}(P) \rangle$, of the closed set of prefixes of P depicted in Table 6.2.

□

Example 6.3.6 (The *concept lattice* of the prefixes of the set of keywords $\{\textit{stringology}, \textit{studies}, \textit{strings}\}$). The line (Hasse) diagram of the concept lattice for the prefixes of the keywords $P = \{\textit{stringology}, \textit{studies}, \textit{strings}\}$ is shown in Figure 6.3.2¹.

□

Observation 6.3.7 (Trie-prefix concept lattice isomorphism). Closer inspection of Figure 6.3.2a and Figure 6.3.2b leads to an interesting observation: If \perp and its incident arcs are excluded from the graph representing the formal concept lattice of prefixes $\mathfrak{B}_{\text{pref}}(P)$, and if the initial state, 1 is excluded from the *trie* $\mathcal{T}(P)$ then $\mathfrak{B}_{\text{pref}}(P)$ and $\mathcal{T}(P)$ and all node and link labels are ignored, they are isomorphic. In further research, a proof can be developed that proves that this isomorphism holds generally.

□

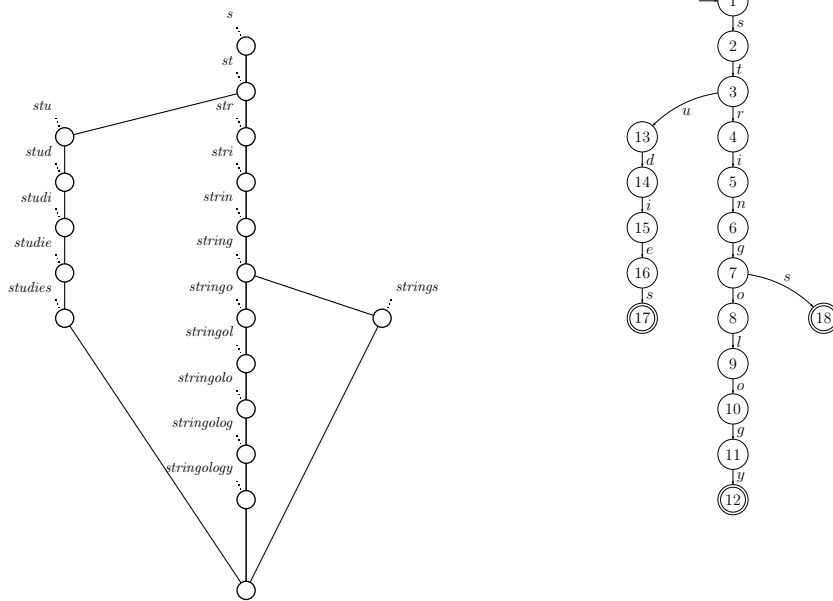
Definition 6.3.8 (Tries as lattices of prefixes). Assuming Observation 6.3.7 is also a property of the prefix concept lattice and trie of *any* set of keywords

¹The object labels have been omitted from this diagram for readability.

6.3 ADFAs for pattern matching using FCA

| K_{pref} | s | st | str | stri | strin | string | stringo | stringol | stringolo | stringolog | stringology | strings | stu | stud | studi | studie | studies |
|-------------------|---|----|-----|------|-------|--------|---------|----------|-----------|------------|-------------|---------|-----|------|-------|--------|---------|
| s | x | | | | | | | | | | | | | | | | |
| st | x | x | | | | | | | | | | | | | | | |
| str | x | x | x | | | | | | | | | | | | | | |
| stri | x | x | x | x | | | | | | | | | | | | | |
| strin | x | x | x | x | x | | | | | | | | | | | | |
| string | x | x | x | x | x | x | | | | | | | | | | | |
| stringo | x | x | x | x | x | x | x | | | | | | | | | | |
| stringol | x | x | x | x | x | x | x | x | | | | | | | | | |
| stringolo | x | x | x | x | x | x | x | x | x | | | | | | | | |
| stringolog | x | x | x | x | x | x | x | x | x | x | | | | | | | |
| stringology | x | x | x | x | x | x | x | x | x | x | x | | | | | | |
| strings | x | x | x | x | x | x | | | | | | x | | | | | |
| stu | x | x | | | | | | | | | | | x | | | | |
| stud | x | x | | | | | | | | | | | x | x | | | |
| studi | x | x | | | | | | | | | | | x | x | x | | |
| studie | x | x | | | | | | | | | | | x | x | x | x | |
| studies | x | x | | | | | | | | | | | x | x | x | x | x |

Table 6.2: Formal context for the prefixes of the keywords $P = \{stringology, studies, strings\}$



(a) The concept lattice derived from the prefixes of the keywords P .

(b) The trie of the keywords P

Figure 6.3.2: The line (Hasse) diagram of the concept lattice derived from the prefixes of the keywords $P = \{stringology, studies, strings\}$ compared to the trie of P

P , it follows that a *trie* T_P , of which the ADFA $\mathcal{M}_T(P)$ is extended to include the \top and \perp nodes of a concept lattice $\mathfrak{B}_{\text{pref}}(P)$, can also be represented by a *prefix lattice*. Using the more general (vs the Formal Concept Analysis) definition of a lattice, I define a *prefix lattice* of P :

$$L_{\text{pref}}(P) = (\mathbf{pref}(P), \leq_p)$$

where $\mathbf{pref}(P)$ is a *prefix closure system*, i.e. $\mathbf{pref}(P)$ gives the closed set of (all) prefixes of P and the elements of $\mathbf{pref}(P)$ are partially ordered by the \leq_p operator on prefixes given in Definition 2.4.15. \square

This definition of a *prefix lattice* corresponds to the classical definition of a lattice, since it is a partially ordered finite set, and thus has a unique supremum and a unique infimum. Because it does not require the formal context that contains the “artificial” incidence relation between prefixes and themselves it is also more concise than the corresponding *concept lattice of prefixes*. In further research, one could explore other closure systems that form the basis for classical lattices that may provide a benefit to the field of pattern matching.

6.3.3 PEPL based ADFAs

6.3.3.1 Introduction

In Chapter 4 I showed how a *Position Encoded Pattern Lattice* (PEPL) defined in Definition 4.2.5 provides a promising new data structure for the representation of and matching on multiple patterns. While Chapter 4 introduces pattern matching based on a PEPL and an APEPL Automaton (see Definition 4.4.12) derived from the APEPL (see Definition 4.4.9) of a set of keywords, in this subsection I provide the outline of two algorithms that derive an *ADFA* from the PEPL of the same set of keywords. See the definition of an ADFA in Definition 2.4.29. The first algorithm derives a lattice-homomorphic ADFA that has superfluous transitions for some common factors of the input keyword set. The second algorithm eliminates this duplication and thus generates smaller ADFAs from the same PEPL. Finally I also provide an outline of an algorithm that uses a PEPL based ADFA of a set of keywords to match an input string against the respective set of keywords.

6.3.3.2 PEPL based ADFA construction

Given $\mathbb{K}_{\overline{P}} = \langle P, \overline{P}, \overline{I} \rangle$, the formal context of the position encoding of a set of keywords P defined in Definition 4.2.3 and its corresponding PEPL $\overline{\mathfrak{P}}$, the PEPL based ADFA $D = \langle Q, \Sigma, \delta, s, F \rangle$ can be derived using Algorithm 18.

Algorithm 18.

{ Note that lattice operations used in statements below are assumed to }
 { apply to the lattice $\overline{\mathfrak{P}}$ }
 { It is assumed that numbering of concept ids for $\overline{\mathfrak{P}}$ starts at 1 }

$\Sigma, \delta, F := \overline{P}, \emptyset, \emptyset$
 { Q is initialised to the ids of all concepts in $\overline{\mathfrak{P}}$ }
 $Q := (\bigcup c : c \in \overline{\mathfrak{P}} : \{id(c)\})$
 $d := \text{ownatt}(\top)$
if $|d| > 0 \rightarrow$
 $s, q_0 := 0, 0$
 { keeps one element in d as final symbol to process }
 select $x_n \in d$
 $d' := d \setminus \{x_n\}$
 for $x \in d' \rightarrow$
 $q_1 := |Q|$
 $Q := Q \cup \{q_1\}$
 $\delta := \delta \cup \{\langle q_0, x, q_1 \rangle\}$
 $q_0 := q_1$
 rof
 { connects final symbol to top concept }
 $\delta := \delta \cup \{\langle q_0, x_n, id(\top) \rangle\}$
 || $|d| = 0 \rightarrow$
 $s := id(\top)$
fi
 { processes all parent - child pairs in lattice }
for $\langle p, c \rangle \in (\bigcup p, c : p \in \overline{\mathfrak{P}} \wedge c \in \text{children}(p) : \{\langle p, c \rangle\}) \rightarrow$
 $q_0 := id(p)$
 $d := \text{int}(c) \setminus \text{int}(p)$
 { keeps one element in d as final symbol to process }
 select $x_n \in d$
 $d' := d \setminus \{x_n\}$

| $K\#$ | $\langle 1, l \rangle$ | $\langle 2, a \rangle$ | $\langle 3, s \rangle$ | $\langle 4, t \rangle$ | $\langle 3, m \rangle$ | $\langle 4, b \rangle$ | $\langle 2, o \rangle$ |
|-------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| last | × | × | × | × | | | |
| lamb | × | × | | | × | × | |
| lost | × | | × | × | | | × |

Table 6.3: Position encoded formal context for the keywords $P = \{last, lamb, lost\}$.

```

for  $x \in d' \rightarrow$ 
     $q_1 := |Q|$ 
     $Q := Q \cup \{q_1\}$ 
     $\delta := \delta \cup \{\langle q_0, x, q_1 \rangle\}$ 
     $q_0 := q_1$ 
rof
    { connects final symbol to child concept }
     $\delta := \delta \cup \{\langle q_0, x_n, id(c) \rangle\}$ 
    { bottom nodes are final }
    if  $\perp \in children(c) \rightarrow$ 
         $F := F \cup \{id(c)\}$ 
    ||  $\perp \notin children(c) \rightarrow$  skip
    fi
rof

```

As an example of using this algorithm, consider the keywords $P = \{last, lamb, lost\}$. The position encoded formal context of P , defined as $\mathbb{K}_{\overline{P}} = \langle P, \overline{P}, \overline{I} \rangle$, is depicted in the cross table in Table 6.3 and its PEPL $\overline{\mathfrak{P}}$ is shown in Figure 6.3.3. The ADFA D derived by Algorithm 18 is shown as an overlay on the PEPL $\overline{\mathfrak{P}}$ in Figure 6.3.4 and in Figure 6.3.5 D is shown without the underlying PEPL for clarity.

Inspection of Figure 6.3.5 leads to the observation that the ADFA generated by Algorithm 18 from a set of keywords P may have superfluous paths for one or more keywords in P . For example, as can be seen in Figure 6.3.6, there are two paths for the keyword *last*. These two paths are shown in Figure 6.3.6.

An improvement on Algorithm 18 to solve this problem is given in Algorithm 19. Instead of processing all parent-child pairs from $\overline{\mathfrak{P}}$ as is done in

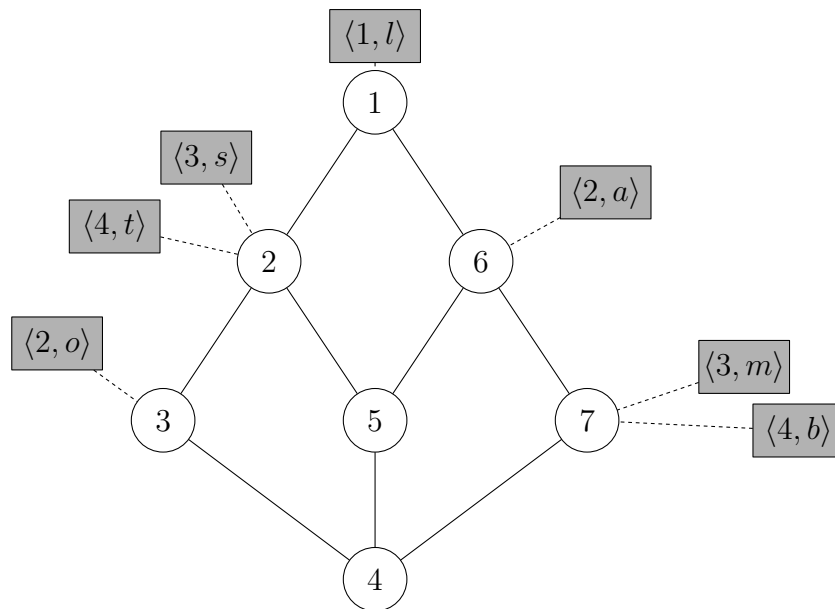


Figure 6.3.3: The line diagram of the concept lattice for the formal context shown in Figure 6.3

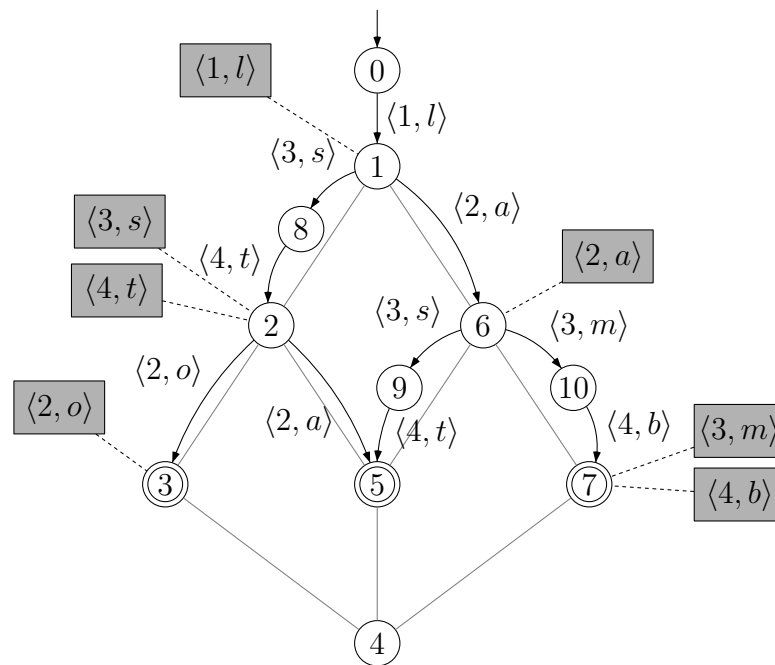
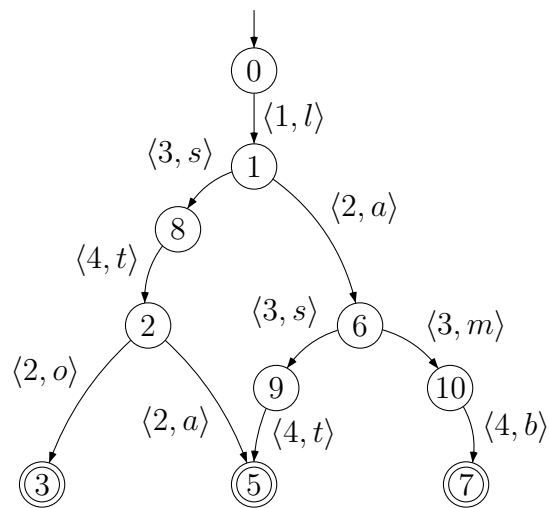


Figure 6.3.4: ADFA D overlaid onto the line diagram of the concept lattice shown in Figure 6.3.3

Figure 6.3.5: A DFA D with underlying lattice removed for clarity

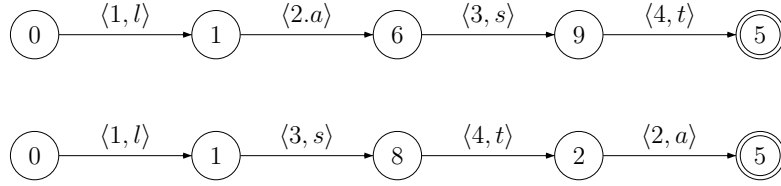


Figure 6.3.6: Two paths for the same keyword *last* in the ADFA shown in Figure 6.3.5

Algorithm 18, Algorithm 19 processes a subset of parent-child pairs in $\overline{\mathfrak{P}}$ using the function $MinP \in \overline{\mathfrak{P}} \rightarrow \mathcal{P}(\overline{\mathfrak{P}})$. This function returns the set of parents of a concept c whose *intents* differ from the intent of c as little as possible, where the difference between $int(c)$ and $int(r)$ (r being a parent of c) is defined as $|int(c) \setminus int(r)|$.

Formally, $MinP$ is defined as follows:

$$MinP(c) = \left(\bigcup p : p \in parents(c) \wedge MinDiff(c, p, parents(c)) : \{p\} \right)$$

where $MinDiff \in \overline{\mathfrak{P}} \times \overline{\mathfrak{P}} \times \mathcal{P}(\overline{\mathfrak{P}}) \rightarrow \mathbb{B}$ is defined as follows:

$$MinDiff(c, p, R) = (\forall r : r \in R \wedge r \neq p : IntDiffSize(c, r) \geq IntDiffSize(c, p))$$

where $IntDiffSize \in \overline{\mathfrak{P}} \times \overline{\mathfrak{P}} \rightarrow \mathbb{N}$

$$IntDiffSize(c, r) = |int(c) \setminus int(r)|$$

Algorithm 19 eliminates all superfluous paths in the output ADFA by processing a single (parent concept, concept) pair for every concept in the PEPL. The algorithm also minimises the number of nodes in the output ADFA due to the function $MinP$ that returns parents of every concept c that have minimal intent differences with respect to c .

A special case needs some discussion. It is possible that more than one parent of a concept is minimal in terms of intent difference. In this case *MinP* returns a set of more than one parent for a given concept. To handle this case, Algorithm 19 uses the GCL *select* keyword to select a parent in the respective set non-deterministically. In future work it should be determined in what sense — if any — might this non-deterministic selection be sub-optimal and if so, what deterministic scheme would work better.

Algorithm 19.

{ Note that lattice operations used in statements below are assumed to }
 { apply to the lattice $\overline{\mathfrak{P}}$ }
 { It is assumed that numbering of concept ids for $\overline{\mathfrak{P}}$ starts at 1 }

$\Sigma, \delta, F := \overline{P}, \emptyset, \emptyset$

{ *Q* is initialised to the ids of all concepts in $\overline{\mathfrak{P}}$ }

$Q := (\bigcup c : c \in \overline{\mathfrak{P}} : \{id(c)\})$

if $int(\top) = \emptyset \rightarrow$

$s, \overline{\mathfrak{P}}_1 := id(\top), \overline{\mathfrak{P}} \setminus \{\top\}$

|| $int(\top) \neq \emptyset \rightarrow$

$s, \overline{\mathfrak{P}}_1 := 0, \overline{\mathfrak{P}}$

fi

for $c \in \overline{\mathfrak{P}}_1 \rightarrow$

if $c = \top \rightarrow$

$q_0 := s$

$d := int(c)$

|| $c \neq \top \rightarrow$

{ See definition of *MinP* above }

select $p \in MinP(c)$

$d := int(c) \setminus int(p)$

$q_0 := id(p)$

fi

{ keeps one element in *d* as final symbol to process }

select $x_n \in d$

$d' := d \setminus \{x_n\}$

for $x \in d' \rightarrow$

$q_1 := |Q|$

```


$$Q := Q \cup \{q_1\}$$


$$\delta := \delta \cup \{\langle q_0, x, q_1 \rangle\}$$


$$q_0 := q_1$$

rof
{ connects final symbol to child concept }

$$\delta := \delta \cup \{\langle q_0, x_n, id(c) \rangle\}$$

{ bottom nodes are final }
if  $\perp \in children(c) \rightarrow$ 

$$F := F \cup \{id(c)\}$$


$$\parallel \perp \notin children(c) \rightarrow \text{skip}$$

fi
rof

```

Figure 6.3.7 shows the ADFA generated by Algorithm 19 on the PEPL for the keywords $P = \{last, lamb, lost\}$. It is clearly smaller than the ADFA generated by Algorithm 18 shown in Figure 6.3.5 as it has only one path for the keyword *last* as opposed to the two paths for the keyword *last* shown in Figure 6.3.6.

6.3.3.3 Comparing the size of PEPL based ADFAs to prefix lattice based ADFAs (tries)

Another hypothesis to be investigated in further research is that PEPL based ADFAs generated by Algorithm 19 are smaller or equal in size in terms of number of states to ADFAs derived from prefix lattices. This hypothesis is based on the following reasoning: A symbol a may occur at the same position k in more than one keyword in P , thereby constituting the single attribute $b = \langle k, a \rangle$ in the position encoded formal context for P . However, in the prefix context for the same keywords P , the symbol a might be part of different prefixes derived from P , resulting in more attributes and therefore a larger context, and a larger ADFA derived from the concept lattice of \mathbb{K}_{pref} than the corresponding PEPL used to derive an ADFA from P using Algorithm 19.

As an example, consider the attribute $\langle 3, s \rangle$ in Table 6.3, representing the formal context of the position encoding of the keywords $P = last, lamb, lost$. The symbol s in this attribute gives rise to *two* attributes *las, los* in the prefix formal context for the same P shown in Table 6.4.

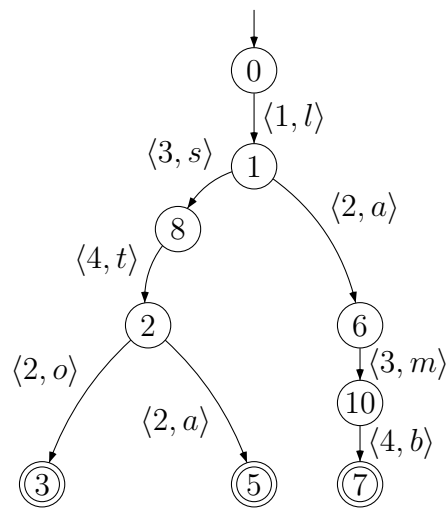
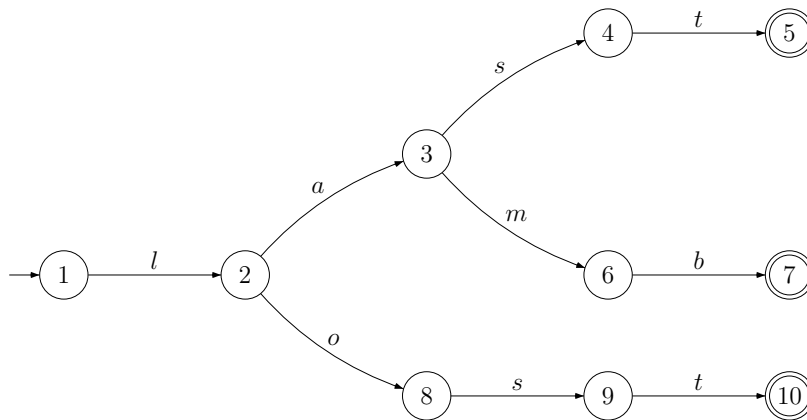


Figure 6.3.7: A DFA generated by Algorithm 19 on the PEPL for the keywords $P = \{last, lamb, lost\}$.

| \mathbb{K}_{pref} | l | la | las | last | lam | lamb | lo | los | lost |
|----------------------------|---|----|-----|------|-----|------|----|-----|------|
| l | × | | | | | | | | |
| la | × | × | | | | | | | |
| las | × | × | × | | | | | | |
| last | × | × | × | × | | | | | |
| lam | × | × | | | × | | | | |
| lamb | × | × | | | × | × | | | |
| lo | × | | | | | | × | | |
| los | × | | | | | | × | × | |
| lost | × | | | | | | × | × | × |

Table 6.4: Formal context for prefixes of the keywords $P = \{last, lamb, lost\}$.Figure 6.3.8: The ADFA generated from the prefix lattice of keywords $P = \{last, lamb, lost\}$.

Anecdotal evidence that this hypothesis may be true is illustrated by the difference in the number of states of the ADFA derived from the keywords $P = \{last, lamb, lost\}$ by algorithm shown in in Figure 6.3.7 and the number of states of the ADFA derived from the same P shown in Figure 6.3.8.

Note that if this hypothesis is proven to be true and isomorphism of prefix lattice ADFAs and corresponding tries proposed above holds, then it would follow that PEPL based ADFAs are smaller than their corresponding tries

also.

6.3.3.4 Pattern matching using PEPL based ADFA's

Clearly, traversal of a PEPL-based ADFA does not process symbols of an input string in a simple left-to-right fashion. Consequently, pattern matching using a PEPL-based ADFA requires that a part of the input string has to be loaded into a suitable data structure in memory, such as an array.

Note that this buffering approach is not a requirement for Algorithm 5 in Chapter 4. Recall that Algorithm 5 finds in an input string, s , the set of match occurrences, MO , of all patterns p in set P , i.e. $i \in MO$ if and only if there is a pattern $p \in P$ that matches a substring of s that starts in position i and is of length $|p|$. An outline of an equivalent algorithm is given in Algorithm 20. Algorithm 20 relies on the PEPL based ADFA derived from P using Algorithm 19.

Algorithm 20.

```
{ Assume the following: }
{  $s$  is the input string }
{  $D$  is the ADFA derived from  $\overline{\mathfrak{P}}$  using Algorithm 19 }
{  $F$  is the set of final states of the ADFA,  $D$  }
{  $s_0$  is the start state of  $D$  }
{  $\delta^{out}(q)$  is a function that gives all out-transitions from  $q$  }
{  $kw(f)$  is a function that gives the keyword associated with  $f \in F$  }
```

```
func MatchOut( $w, Q_n$ )
  result :=  $\emptyset$ 
  for  $q \in Q_n$ 
    for  $\langle \langle \alpha, i \rangle, p \rangle \in \delta^{out}(q) \rightarrow$ 
      if  $w[i] = \alpha \rightarrow$ 
        result := result  $\cup \{p\}$ 
      ||  $w[i] \neq \alpha \rightarrow$  skip
    fi
  rof
rof
return result
```

cnuf

$l, MO := (\mathbf{MAX} p : p \in P : |p|), \emptyset$

for $t \in [0..|s|) \rightarrow$

$b := s[t..t + l - 1]$

$Q_b := MatchOut(b, \{s_0\})$

do $Q_b \neq \emptyset \rightarrow$

for $q \in Q_b \rightarrow$

if $q \in F \rightarrow$

$MO \cup \{kw(q)\}$

$\parallel q \notin F \rightarrow$ **skip**

fi

rof

$Q_b := MatchOut(b, Q_b)$

od

rof

6.3.4 RPEPL based ADFAs

6.3.4.1 Introducing the position misalignment problem

The PEPL based ADFA approach has a limitation: The PEPL of a set of keywords will only “consolidate” common factors of a set of keywords that start at the same offset. This consolidation effect is illustrated by the single sub-path of the generated ADFA that has states $\{1, 8, 2\}$ shown in Figure 6.3.7. This sub-path recognises the factor st that is a suffix of two keywords $\{last, lost\} \subset P$ and as can be seen in this figure, there is no other sub-path in the ADFA that will recognise this same suffix. However, consider the ADFA generated for the keywords $P' = \{blast, lamb, lost\}$ by Algorithm 19. The position encoded formal context for P' is shown in Table 6.5. The corresponding PEPL $\overline{\mathfrak{P}}$ is shown in Figure 6.3.9 and the corresponding ADFA generated by Algorithm 19 is shown in Figure 6.3.10. It is clear that for the keywords $P' = \{blast, lamb, lost\}$, the factor st that is a suffix of two keywords $\{blast, lost\} \subset P'$ gives rise to two sub-paths in the output ADFA,

| $K\#$ | $\langle 1, b \rangle$ | $\langle 2, l \rangle$ | $\langle 3, a \rangle$ | $\langle 4, s \rangle$ | $\langle 5, t \rangle$ | $\langle 1, l \rangle$ | $\langle 2, a \rangle$ | $\langle 3, s \rangle$ | $\langle 4, t \rangle$ | $\langle 3, m \rangle$ | $\langle 4, b \rangle$ | $\langle 2, o \rangle$ |
|-------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| blast | × | × | × | × | × | | | | | | | |
| lamb | | | | | | × | × | | | × | × | |
| lost | | | | | | × | | × | × | | | × |

Table 6.5: Position encoded formal context for the keywords $P' = \{blast, lamb, lost\}$.

so the consolidation of the common suffix st does not happen as in the case of $P = \{last, lamb, lost\}$. I call this limitation of the PEPL based ADFA approach the *position misalignment problem*.

6.3.4.2 A proposed solution to the position misalignment problem

A possible solution to the position misalignment problem involves the use *reverse position encoding* of the input set of keywords P . This would ensure that a common suffix of keywords in P give rise to one set of attributes for the symbols in such a suffix in a corresponding formal context and a smaller concept lattice. Reverse position encoding of keywords and the resulting formal context and concept lattice for reverse position encoded keywords follows below.

Definition 6.3.9 (Reverse position encoding of a pattern and a set of patterns). The *reverse position encoding* of string w is the set of position-symbol pairs denoted by \underline{w} and is given by

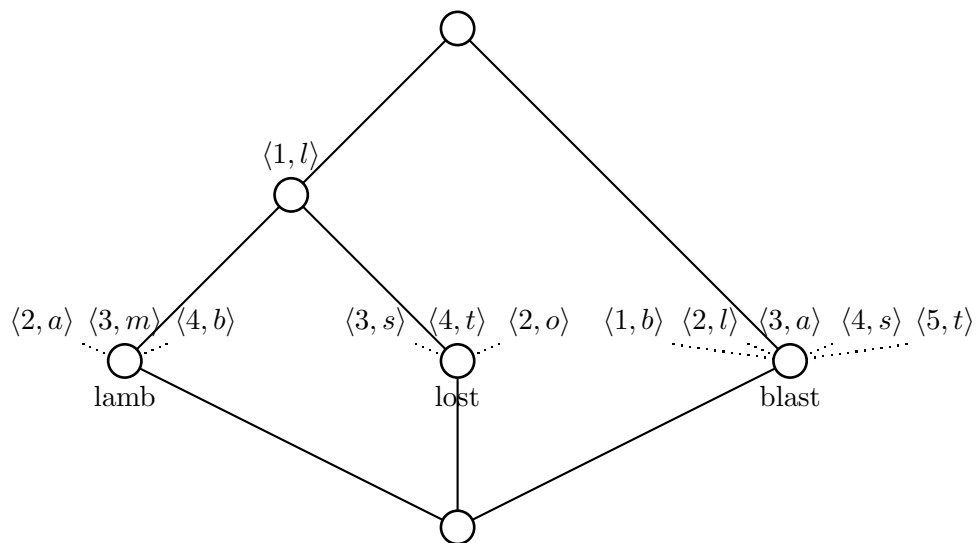
$$\underline{w} = \left(\bigcup k : k \in [1, |w|] : \{ \langle k, w_{|w|-k} \rangle \} \right)$$

The *reverse position encoding* of a set of strings P is denoted \underline{P} and is given by

$$\underline{P} = \left(\bigcup w : w \in P : \underline{w} \right)$$

□

Example 6.3.10. For example, the reverse position encoding of “pack” is $\underline{pack} = \{ \langle 1, k \rangle, \langle 2, c \rangle, \langle 3, a \rangle, \langle 4, p \rangle \}$, and of “packet” it is $\underline{packet} = \{ \langle 1, t \rangle, \langle 2, e \rangle, \langle 3, k \rangle, \langle 4, c \rangle, \langle 5, a \rangle, \langle 6, p \rangle \}$.

Figure 6.3.9: The line diagram of P'

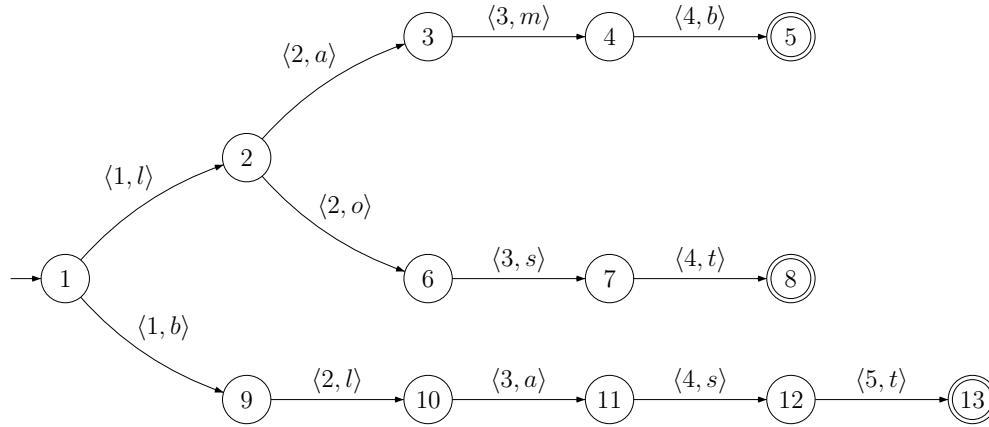


Figure 6.3.10: The A DFA derived from the PEPL for P' using Algorithm 19

Definition 6.3.11 (Formal context and concept lattice based on the reverse position encoding of a set of patterns). Given any set of patterns P , we can now constitute a formal context \mathbb{K}_P along the following lines. Regard the words in the set of patterns as a set of objects. Let the position-symbol pairs of the reverse position encoding of the set of patterns serve as attributes of these objects: a given word has as its attributes all the position-symbol pairs that make up its reverse position encoding.

This context is defined as $\mathbb{K}_P = \langle P, \underline{P}, \underline{I} \rangle$, where \underline{I} is the incidence relation between objects and attributes depicted in the cross table. The formal concept lattice to be derived from such a context will be called a *Reverse Position Encoded Pattern Lattice* (RPEPL), denoted by $\mathfrak{P}(\langle P, \underline{P}, \underline{I} \rangle)$ or, more concisely, by \mathfrak{P} . □

Consider the set of keywords $P = \{blast, lamb, lost\}$. The cross table that represents the *reverse position encoded* formal context (\mathbb{K}_P) for P is shown in Table 6.6. The line diagram of the corresponding concept lattice \mathfrak{P} is shown in Figure 6.3.11.

Algorithm 19 applied to the RPEPL for the keywords $P = \{blast, lamb, lost\}$ shown in Figure 6.3.11 generates the A DFA shown in Figure 6.3.12. As can be seen by comparing the number of states in the A DFAs depicted in Figure 6.3.12

| \mathbb{K}_P | $\langle 1, t \rangle$ | $\langle 2, s \rangle$ | $\langle 3, a \rangle$ | $\langle 4, l \rangle$ | $\langle 5, b \rangle$ | $\langle 1, b \rangle$ | $\langle 2, m \rangle$ | $\langle 3, o \rangle$ |
|----------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| blast | × | × | × | × | × | | | |
| lamb | | | × | × | | × | × | |
| lost | × | × | | × | | | | × |

Table 6.6: Reverse position encoded formal context for the keywords $P = \{blast, lamb, lost\}$.

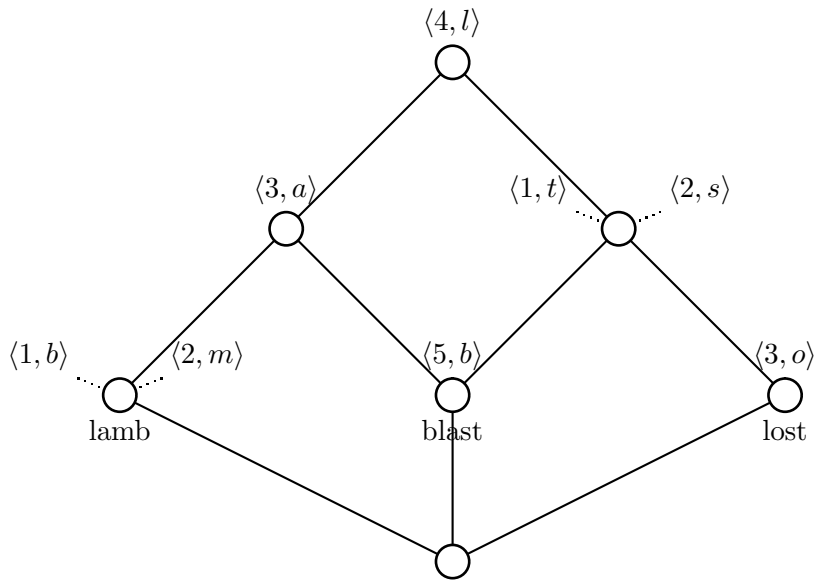


Figure 6.3.11: The line (Hasse) diagram of the RPEPL for the keywords $P = \{blast, lamb, lost\}$.

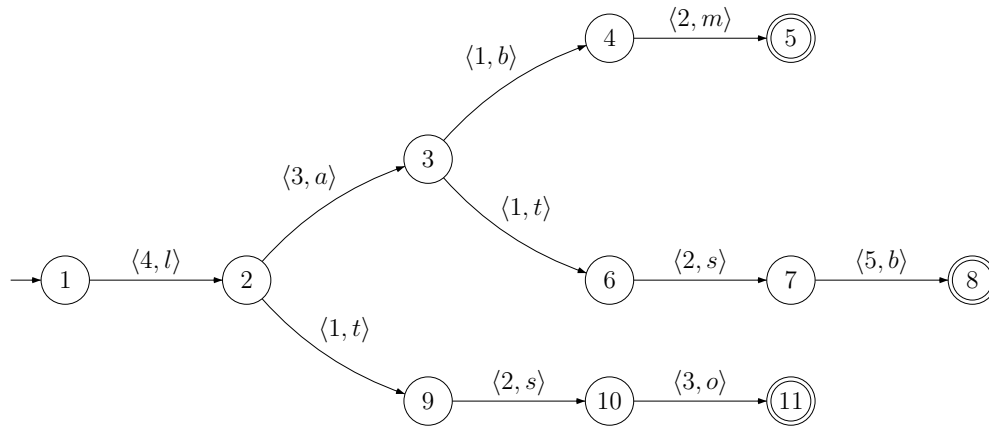


Figure 6.3.12: The line diagram of the ADFAs generated by Algorithm 19 on the RPEPL for the keywords $P = \{\textit{blast}, \textit{lamb}, \textit{lost}\}$.

and Figure 6.3.10 respectively, the ADFAs generated by the Algorithm 19 on the RPEPL of the same set of keywords, has solved the *position misalignment problem* for this specific case. Note that Algorithm 20 can be adapted to create an algorithm to match strings using RPEPL based ADFAs by simply replacing $w[i]$ in function *MatchOut* with $w[|w| - i]$.

6.3.4.3 Limitation of the RPEPL approach and Future Work

I have shown how the RPEPL solves the position misalignment problem for common suffixes and indeed would generate smaller ADFAs than a PEPL would for the set of keywords given in the example above. However, I can show cases for which PEPLs would lead to smaller ADFAs than RPEPLs using Algorithm 19. My intuition is that when a set of keywords has a larger set of common *prefixes* than *suffixes*, then Algorithm 19 applied to a PEPL would generate a smaller ADFAs than applying Algorithm 19 to a RPEPL would; and when a set of keywords has a larger set of common *suffixes* than *prefixes*, then Algorithm 19 applied to a RPEPL would generate a smaller ADFAs than applying Algorithm 19 applied to a PEPL would. In future work the following

questions could be investigated:

- Would augmented PEPLs (APEPLs) introduced in Chapter 4 passed into Algorithm 19 create smaller position encoded ADFAs than applying the algorithm to PEPLs or RPEPLs? If so, is this generally true?
- Is there a way to pre-process a set of keywords to determine which of a PEPL, RPEPL or APEPL should be passed into Algorithm 19 to create the smallest position encoded ADFA?
- Is there a different encoding scheme that would ensure that all common factors of a set of keywords lead to the same set of attributes in a new context, lattice and corresponding ADFA that is minimal?

Chapter 7

Conclusion

This work has demonstrated the theoretical viability of leveraging FCA technology in a variety of ways to look anew at many stringology problems. Indeed, wherever information needs to be clustered together in some ordered fashion, FCA could have a potential role to play. Concerns about the practical viability of these strategies could well be raised, particularly in the light of the state space explosion problem associated with FCA lattices. Two observations deserve mention in this regard.

- **Constrained lattice:** In many of the applications explored to date, the problem space explicitly prevents worst case lattice sizes from being generated. For example, the number of attributes in the context used for the F DFA application is maximally $|\Sigma| \times |Q|$ where Q is the set of DFA states and Σ is the alphabet. Each attribute represents a symbol / state destination pair. The theoretical worst case lattice size would be when every object (state) has exactly $(|\Sigma| \times |Q|) - 1$ attributes. However, this is not feasible in the problem domain being considered—there can only be $|Q|$ out-transitions for a DFA. This ameliorates the state space explosion problem.
- **Incremental Application:** It should also be noted that in some cases it is not necessary to build the entire lattice, because one is not seeking absolute optimality, but merely an improvement. Again, considering the F DFA construction example, it is not necessary that the entire DFA state space be considered for transformation to an F DFA—one may judiciously

select areas of the DFA state space that are most likely to be profitably changed. Precisely how this should be done is a matter of future research.

References

- [AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [AF92] Amihood Amir and Martin Farach. Two-dimensional dictionary matching. *Information Processing Letters*, 44(5):233–239, 1992.
- [Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
- [Bak78] T Baker. A technique for extending rapid exact string matching to arrays of more than one dimension. *Inf. Proc Letters*, 6:168–170, 1978.
- [BBE⁺83] Anselm Blumer, Janet Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross M. McConnell. Linear size finite automata for the set of all subwords of a word - an outline of results. *Bulletin of the EATCS*, 21:12–20, 1983.
- [BBH⁺85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40(Supplement C):31 – 55, 1985. Eleventh International Colloquium on Automata, Languages and Programming.
- [Bir77] R Bird. Two dimensional pattern matching. *Inf. Proc Letters*, 6:168–170, 1977.
- [BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772,

- October 1977.
- [BYR93] R. Baeza-Yates and M. Regnier. Fast two dimensional pattern matching. 1993.
- [CH97] Maxime Crochemore and Christophe Hancart. Automata for matching patterns. In *Handbook of formal languages, vol. 2*, pages 399–462. Springer-Verlag New York, Inc., New York, NY, USA, 1997.
- [CKW09] Wikus Coetser, Derrick G. Kourie, and Bruce W. Watson. On regular expression hashing to reduce FA size. *International Journal of Foundations of Computer Science*, 20(6):1069–1086, 2009.
- [Cle08] Loek Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. PhD thesis, Eindhoven University of Technology, the Netherlands, April 2008.
- [CR94] Maxime A. Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.
- [CR03] Maxime A. Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific Publishing Company, 2003.
- [CR04a] C. Carpineto and G. Romano. *Concept Data Analysis: Theory and Applications*. John Wiley & Sons, Ltd, 2004.
- [CR04b] Claudio Carpineto and Giovanni Romano. *Concept Data Analysis: Theory and Applications*. John Wiley & Sons, England, 2004.
- [Cro87] Maxime Crochemore. Longest common factor of two words. In *Colloquium on Trees in Algebra and Programming*, pages 26–36. Springer, 1987.
- [CWZ10] Loek Cleophas, Bruce W. Watson, and Gerard Zwaan. A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms. *Science of Computer Programming*, 75:1095–1112, 2010.
- [dBCKW10] Noud de Beijer, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. Improving automata efficiency by stretching and jamming. In *Proceedings of the Prague Stringology Con-*

- ference (*PSC*), pages 9–24, 2010.
- [DDH84] Peter Dencker, Karl Dürre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems*, 6(4):546–572, October 1984.
- [DF88] Edsger W. Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison Wesley, 1988.
- [DH95] Karel Driesen and Urs Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 141–155, New York, NY, USA, 1995. ACM.
- [DW11] Jan Daciuk and Dawid Weiss. Smaller representation of finite state automata. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA)*, pages 118–129, 2011.
- [Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 137–143. IEEE, 1997.
- [FJM10] Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar. Subtree matching by pushdown automata. *Computer Science and Information Systems*, 7(2):331–357, 2010.
- [FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
- [Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
- [FV97] J.D. Roos F.J. Venter, G.D. Oosthuizen. Knowledge discovery in databases using lattices. *Expert Systems with Applications*, 13(4):259–264, November 1997.
- [GG97] Raffaele Giancarlo and Roberto Grossi. Suffix tree data structures for matrices. In *Pattern matching algorithms*, pages 293–340. Oxford University Press, 1997.

- [Gon88] Gaston H Gonnet. *Efficient searching of text and pictures*. UW Centre for the New Oxford English Dictionary, 1988.
- [GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin, 1999.
- [Jan10] Jan Janoušek. *Arbology: Algorithms on trees and pushdown automata*. PhD thesis, habilitation thesis, Brno University of Technology, 2010, submitted, 2010.
- [JM09] Jan Janoušek and Bořivoj Melichar. On regular tree languages and deterministic pushdown automata. *Acta Informatica*, 46(7):533–547, 2009.
- [KJHMP77] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [KKP03] Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theoretical Computer Science*, 302(1-3):223–238, 2003.
- [KN07] Ernest Ketcha Ngassam. *Towards cache optimization in finite automata implementations*. PhD thesis, University of Pretoria, 2007.
- [KO98] Derrick G. Kourie and G. Deon Oosthuizen. Lattices in machine learning: Complexity issues. *Acta Informatica*, 35:269–292, 1998.
- [KO02] Sergei O. Kuznetsov and Sergei A. Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):189–216, 2002.
- [KOWvdM09] Derrick G. Kourie, Sergei A. Obiedkov, Bruce W. Watson, and Dean van der Merwe. An incremental algorithm to construct a lattice of set intersections. *Science of Computer Programming*, 74(3):128–142, 2009.
- [KS87] K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, 43:169–183, 1987.

- [KWCV12] Derrick G. Kourie, Bruce W. Watson, Loek Cleophas, and Fritz Venter. Failure deterministic finite automata. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Sixteenth Prague Stringologic Conference*, Czech Technical University in Prague, Czech Republic, August 2012.
- [McC76] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.
- [MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.
- [Mor68] Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.
- [NGP07] Joong Chae Na, Raffaele Giancarlo, and Kunsoo Park. On-line construction of two-dimensional suffix trees in $O(n^2 \log n)$ time. *Algorithmica*, 48(2):173–186, 2007.
- [Nxu16] M Nxumalo. An assessment of selected algorithms for generating failure deterministic finite automata. Dissertation, University of Pretoria, <http://hdl.handle.net/2263/57216>, April 2016.
- [OV93] G.D. Oosthuizen and F.J. Venter. Using a lattice for visual analysis of categorical data. In H.Levkowitz G. Grinstein, editor, *Perceptual Issues in Visualization*, volume 1 of *IFIP Series on Computer Graphics*. Springer-Verlag, New York, 1993.
- [RT89] Zhu R.F. and Takaoka T. A technique for two-dimensional pattern matching. *CACM*, 9(32):1110–1120, 1989.
- [Sco59] Michael O. Rabin; Dana S. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.
- [Smy03] William F. Smyth. *Computing Patterns in Strings*. Addison-Wesley, 2003.
- [TY79] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM* *ACM*, 22(11):606–

- 611, November 1979.
- [Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [Var04] George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2004.
- [VBV⁺18] Carl C. Valente, Florian F. Bauer, Fritz Venter, Bruce Watson, and Hélène H. Nieuwoudt. Modelling the sensory space of varietal wines: Mining of large, unstructured text data and visualisation of style patterns. *Nature Scientific Reports*, 8, March 2018.
- [VKW09] Fritz Venter, Derrick G. Kourie, and Bruce W. Watson. FCA-based two dimensional pattern matching. In *Proceedings of the 7th International Conference on Formal Concept Analysis*, 2009.
- [Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, September 1995.
- [Wat10] Bruce W. Watson. *Constructing Minimal Acyclic Deterministic Finite Automata*. PhD thesis, University of Pretoria, November 2010.
- [Wei73] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT’08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.
- [Yev00] S. A. Yevtushenko. System of data analysis “Concept Explorer”. In *Proceedings of the 7th National Conference on Artificial Intelligence KII-2000, Russia*, 2000.
- [Yev06] Serhiy A. Yevtushenko. <http://conexp.sourceforge.net/>, 2006.