Taxonomies and Toolkits of Regular Language Algorithms

Bruce William Watson

Eindhoven University of Technology
Department of Mathematics and Computing Science

Cover design by Nanette Saes.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Watson, Bruce William

Taxonomies and toolkits of regular language algorithms /
Bruce William Watson. — Eindhoven: Eindhoven University
of Technology
Thesis Technische Universiteit Eindhoven. — With index,
ref. — With summary in Dutch. ISBN 90-386-0396-7
Subject headings: finite automata construction /
taxonomies / pattern matching.

Taxonomies and Toolkits of Regular Language Algorithms

Proefschrift

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. J.H. VAN LINT, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP VRIJDAG 15 SEPTEMBER 1995 OM 16.00 UUR

door

Bruce William Watson

geboren te Mutare, Zimbabwe

Dit proefschrift is goedgekeurd door de promotoren

prof.dr. F.E.J. Kruseman Aretz
prof.dr. P. Klint

en de co-promotor

dr.ir. C. Hemerik

Forty Years On

Forty years on, when far and asunder
Parted are those who are singing today,
When you look back, and forgetfully wonder
What you were like in your work and your play;
Then, it may be, there will often come o'er you
Glimpses of notes like the catch of a song —
Visions of boyhood shall float them before you,
Echoes of dreamland shall bear them along.
Follow up! Follow up! Follow up! Follow up!
Till the field ring again and again,
With the tramp of the twenty-two men,
Follow up! Follow up!

Routs and discomfitures, rushes and rallies,
Bases attempted, and rescued, and won,
Strife without anger, and art without malice, —
How will it seem to you forty years on?
Then you will say, not a feverish minute
Strained the weak heart, and the wavering knee,
Never the battle raged hottest, but in it
Neither the last nor the faintest were we!
Follow up! Follow up!

O the great days, in the distance enchanted,
Days of fresh air, in the rain and the sun,
How we rejoiced as we struggled and panted —
Hardly believable, forty years on!
How we discoursed of them, one with another,
Auguring triumph, or balancing fate,
Loved the ally with the heart of a brother,
Hated the foe with a playing at hate!
Follow up! Follow up!

Forty years on, growing older and older,
Shorter in wind, and in memory long,
Feeble of foot and rheumatic of shoulder,
What will it help you that once you were strong?
God gives us bases to guard or beleaguer,
Games to play out, whether earnest or fun,
Fights for the fearless, and goals for the eager,
Twenty, and thirty, and forty years on!
Follow up! Follow up!

Edward Ernest Bowen (1836-1901)

To my parents, Lyn and Mervin, and to my uncle Ken

Acknowledgements

There are a number of people[1] that I would like to thank for their assistance (in various ways) during the research that is reported in this dissertation; I apologize in advance for any omissions. First and foremost, I would like to thank my parents, Mervin and Lyn Watson, and the rest of my immediate family, Ken Carroll, Renée Watson, and Richard Watson, for their loving support throughout the last 27 years. I also thank Nanette for her unfailing support during this research.

As good friends, colleagues, and mentors Frans Kruseman Aretz, Kees Hemerik, and Gerard Zwaan devoted a great deal of their time to directing my research, reviewing my ideas, and introducing me to the finer points of Dutch culture. As members of my kernel thesis committee, Roland Backhouse, Paul Klint, and Martin Rem, and my greater thesis committee, Emile Aarts, Hans Jonkers, and Jan van Leeuwen all provided a great deal of feedback on my research. The proofreading skills of Mervin Watson (grammatical), Nanette Saes (grammatical and translation to Dutch) and Richard Watson (technical) proved to be crucial in polishing this dissertation. Additionally, the other members of the original 'PI' research group, Huub ten Eikelder, Rik van Geldrop, Erik Poll, and (temporarily) Fairouz Kamareddine served as sounding boards for some of my ideas.

Quite a few people perform research in areas related to my thesis. The following people were particularly helpful in providing feedback on some of my ideas: Valentin Antimirov, Carel Braam, Anne Brüggemann-Klein, Jean-Marc Champarnaud, Manfred Dalmeijer, Rik van Geldrop, Roelf van den Heever, Pieter 't Hoen, Anne Kaldewaij, Derrick Kourie, Harold de Laat, Bob Paige, Darrell Raymond, Tom Verhoeff, Bart Wakkee, and Derick Wood.

A number of people were also helpful in pointing me to related research. These people are: Al Aho, Gerard Berry, John Brzozowski, Maxime Crochemore, Jo Ebergen, Nigel Horspool, Tiko Kameda, Thierry Lecroq, Ming Li, Andreas Pothoff, Marc Saes, Ravi Sethi, and Jeffrey Ullman.

As interesting as the development of taxonomies and toolkits is, it would not have been possible without the support of my friends (especially the 'Waterloo gang': Frank, Karim, Anne, and Roger). In particular, Jan and Willy went to great lengths to make me feel at home in the Netherlands.

[1] All titles have been omitted and first names are used.

Contents

List of Figures

Part I

Prologue

Chapter 1

Introduction

In this chapter, we present an introduction to the contents and the structure of this dissertation.

1.1 Problem statement

A number of fundamental computing science problems have been extensively studied since the 1950s and the 1960s. As these problems were studied, numerous solutions (in the form of algorithms) were developed over the years. Although new algorithms still appear from time to time, each of these fields can be considered mature. In the solutions to many of the well-studied computing science problems, we can identify three deficiencies:

1. Algorithms solving the same problem are difficult to compare to one another. This is usually due to the use of different programming languages, styles of presentation, or simply the addition of unnecessary details.

2. Collections of implementations of algorithms solving a problem are difficult, if not impossible, to find. Some of the algorithms are presented in a relatively obsolete manner, either using old notations or programming languages for which no compilers exist, making it difficult to either implement the algorithm or find an existing implementation.

3. Little is known about the comparative practical running time performance of the algorithms. The lack of existing implementations in one and the same framework, especially of some of the older algorithms, makes it difficult to determine the running time characteristics of the algorithms. A software engineer selecting one of the algorithms will usually do so on the basis of the algorithm's theoretical running time, or simply by guessing.

In this dissertation, a solution to each of the three deficiencies is presented for each of the following three fundamental computing science problems:

1. Keyword pattern matching in strings. Given a finite non-empty set of keywords (the patterns) and an input string, find the set of all occurrences of a keyword as a substring of the input string.

2. Finite automata (*FA*) construction. Given a regular expression, construct a finite automaton which accepts the language denoted by the regular expression.

3. Deterministic finite automata (*DFA*) minimization. Given a *DFA*, construct the unique minimal *DFA* accepting the same language.

We do not necessarily consider *all* of the known algorithms solving the problems. For example, we restrict ourselves to batch-style algorithms[1], as opposed to incremental algorithms[2]. Some finite automata construction algorithms considered in [Wat93a] can be used in a (rudimentary) incremental fashion (this is an coincidental side-effect of the algorithm derivations presented there). A much more advanced treatment of incremental algorithms is given in [HKR94], where the construction of finite automata (for compiler lexical analysis) is considered as an example.

In the following section, we present a broad overview of the structure of this dissertation, describing the solutions to the three deficiencies. Following this, is a discussion of the intended audience.

1.2 Structure of this dissertation

The dissertation is divided into five parts. Parts I contains the prologue — the introduction (this chapter) and the mathematical preliminaries. Part V contains the epilogue — the conclusions, some challenges and directions for future work, the literature references, the index, the summary, the Dutch summary, and my curriculum vitae.

The difficulty of comparing algorithms solving the same problem is addressed by constructing a *taxonomy* of all of the algorithms solving the particular problem. Part II presents a collection of such taxonomies. Since Chapter 3 contains an introduction to the method of constructing taxonomies, we present only a brief outline here. Each of the algorithms is rewritten in a common notation and inspected to determine its essential ideas and ingredients (collectively known as *details*). The details take one of two forms: *problem details* are restrictions of the problem, whereas *algorithm details* are transformations to the algorithm itself. Each algorithm can then be characterized by its set of constituent details. In constructing the taxonomy, the common details of several algorithms can be factored out and presented together. From this factoring process, we construct a 'family tree' of the algorithms — indicating what any two of the algorithms have in common and where they

[1] A batch-style algorithm is one which performs some computation on its input, produces output, and terminates.

[2] An incremental algorithm is one which is able to deal with a change in the input without necessarily recomputing from scratch. For example, an incremental keyword pattern matching algorithm would be able to deal with the addition of a new pattern keyword, without redoing all of the precomputation.

differ. After this introduction to the method of constructing taxonomies, the taxonomies themselves are presented in the remaining chapters of Part II. Pattern matching algorithms are considered in Chapter 4, *FA* construction algorithms in Chapter 6, and *DFA* minimization algorithms in Chapter 7. An additional chapter (Chapter 5) presents a new pattern matching algorithm (which was derived from one of the taxonomies), answering an open question posed by A.V. Aho in [Aho80, p. 342].

Part III presents a pair of toolkits, thus solving the second of the three deficiencies previously outlined. The toolkits are implemented (as object-oriented C++ class libraries) directly from the taxonomies given in Part II. The inheritance hierarchy of each of the toolkits also follows directly from the family tree structure of the (respective) taxonomies. Chapter 8 provides an introduction to the design and implementation of class libraries, including the attendant problems and issues. The SPARE Parts[3], a toolkit of pattern matching algorithms, is detailed in Chapter 9. Chapter 10 describes the *FA* construction algorithms implemented in FIRE Lite[4], a toolkit of finite automata algorithms. The *DFA* minimization algorithms which are implemented in FIRE Lite are described in Chapter 11. This part assumes a good grasp of the C++ programming language and the related terminology; references for books covering C++ are given in Chapter 8.

In Part IV, we consider the practical performance of many of the algorithms derived in Part II and implemented in the toolkits, thereby addressing the third deficiency introduced above. A wide variety of input data was used to gather information on most of the algorithms presented in the taxonomies. Although there is little to say about such data-gathering methods, Chapter 12 lists the principles used. Chapters 13, 14, and 15 present data on the performance of the keyword pattern matching, *FA* construction, and *DFA* minimization algorithms, respectively.

1.3 Intended audience

The intended audience of this dissertation can be divided into a number of different groups. Each of these groups is mentioned in the following paragraphs, along with an outline of chapters and topics of particular interest to each group.

- *Taxonomists.* A number of areas of computing science are now mature enough that the corresponding algorithms can be taxonomized. Creating a taxonomy serves to bring order to the field, in a sense 'cleaning it up'. Since the taxonomies also serve as useful teaching aids and surveys of the field, it can be expected that more algorithm families will be taxonomized. The taxonomies given in Part II can be used as examples when creating new taxonomies.

- *Algorithm designers.* Embedded in each of the taxonomies are a number of new algorithms. The method of taxonomy development is well suited to the discovery

[3]String PAttern REcognition.
[4]FInite automata and Regular Expressions.

of new algorithms. Algorithm designers can use the method of taxonomization to develop new algorithms, and should read Part II. (Readers who think that the most efficient algorithms for these problems have already been discovered, should read Part IV, which shows that some of the new algorithms have practical importance.) The new regular expression pattern matching algorithm (given in Chapter 5) is a particularly good example of an algorithm which was designed using a mathematical approach to program construction. It is unlikely that the algorithm could have been developed using more traditional 'hack and debug' techniques.

- *Class library writers/generic program writers.* Techniques for structuring and writing class libraries (especially generic class libraries — those that are type parameterized as in [MeyB94]) are still in their infancy. Parts II and III together present a method of structuring class libraries and their inheritance hierarchies from the corresponding taxonomies. The fact that the taxonomies implicitly contain algorithm proofs, yields a high degree of confidence in the quality of the class libraries.

- *Programmers.* Programmers needing implementations of algorithms for keyword pattern matching, finite automata construction, or deterministic finite automata minimization should read the chapters of Part III which are relevant to their needs. A solid background in C++ and object-oriented programming is required. To best understand the algorithms, programmers should read the chapters of Part II which correspond to the toolkits they are using. When selecting an algorithm, a programmer can make use of the data and recommendations presented in Part IV.

- *Software engineers.* The taxonomies (Part II) provide a successful example of manipulating abstract algorithms, obtained directly from a specification, into more easily implemented and efficient ones. Furthermore, the abstract algorithms and the implementation techniques described in Chapter 8 combine well to produce the toolkits described in Part III.

Chapter 2

Mathematical preliminaries

In this chapter, we present a number of definitions and properties required for reading this dissertation. This chapter can be skipped and referred to while reading individual chapters. Definitions that are used only in one chapter will be presented when needed, and can be considered 'local' to that chapter.

2.1 Notations and conventions

In the taxonomies, we aim to derive algorithms that correspond to the well-known ones found in the literature. For this reason, we will frequently name variables, functions, predicates, and relations such that they correspond to their names as given in the literature. Furthermore, we will adopt the commonly used names for standard concepts, such as $\mathcal{O}$ for 'big-oh' (running time) notation. While this makes it particularly difficult to adopt completely uniform naming conventions, the following conventions will be used for names that do not have historical significance.

Convention 2.1 (Naming functions, sets, etc.): We will adopt the following general naming conventions:

- A, B, C for arbitrary sets.
- D, E, F, G, H for relations.
- V, W for alphabets.
- a, b, c, d, e for alphabet symbols.
- $r, s, t, u, v, w, x, y, z$ for words (over an alphabet).
- L, P for languages.
- h, i, j, k for integer variables.
- M, N for finite automata (including Moore machines).

- p, q, r for states, and Q for state sets.
- Lower case Greek letters, such as γ, δ, τ for automata transition functions.
- I, J for predicates used to express program invariants.

Functions, some relations, and predicates (other than those used to express program invariants) will frequently be given names longer than one letter, chosen to be suggestive of their use. Sometimes subscripts, superscripts, hats, overbars or prime symbols will be used in addition to one of the aforementioned names. □

New terms will be typeset in an italic shape when they are first mentioned or defined.

Notation 2.2 (Symbol $\perp$): We will frequently use the symbol $\perp$ (pronounced 'bottom') to denote an undefined value (usually in the codomain of a function). □

2.2 Basic definitions

In this section, we present some basic definitions which are not specific to any one topic.

Definition 2.3 (Powerset): For any set A we use $\mathcal{P}(A)$ to denote the set of all subsets of A. $\mathcal{P}(A)$ is called the *powerset* of A; it is sometimes written as 2^A in the literature. □

Definition 2.4 (Alphabet): An *alphabet* is a finite non-empty set of *symbols*. We will sometimes use the term *character* instead of symbol. □

Definition 2.5 (Nondeterministic algorithm): An algorithm is called *nondeterministic* if the order in which its statements can be executed is not fixed. □

Notation 2.6 (Quantifications): We assume that the reader has a basic knowledge of the meaning of *quantification*. We use the following notation in this dissertation:

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

where $\oplus$ is the associative and commutative quantification operator (with unit $e_\oplus$), a is the dummy variable introduced (we allow the introduction of more than one dummy), R is the range predicate on the dummy, and f is the quantification expression (usually a function involving the dummy). By definition, we have:

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

The following table lists some of the most commonly quantified operators, their quantified symbols, and their units:

Operator	$\vee$ ($\exists$)	$\wedge$ ($\forall$)	$\cup$ ($\bigcup$)	**min** (**MIN**)	**max** (**MAX**)	$+$ (Σ)
Unit	*false*	*true*	$\emptyset$	$+\infty$	$-\infty$	0

□

Notation 2.7 (Sets): For any given predicate P, we use $\{ a \mid P(a) \}$ to denote the set of all a such that $P(a)$ holds. □

Definition 2.8 (for-rof statement): The **for-rof** statement is taken from [vdEi92]. Given predicate P and statement S, the statement **for** $x : P \to S$ **rof** amounts to executing statement list S once for each value of x that satisfies P initially (assuming that there are a finite number of such values for x). The order in which the values of x are chosen is arbitrary. □

Notation 2.9 (Conditional conjunction): We use **cand** and **cor** to refer (respectively) to conditional conjunction and conditional disjunction. □

Definition 2.10 (Sets of functions): For any two sets A and B, we use $A \longrightarrow B$ to denote the set of all total functions from A to B. We use $A \not\longrightarrow B$ to denote the set of all partial functions from A to B. □

Notation 2.11 (Function signatures): For any two sets A and B, we use the notation $f \in A \longrightarrow B$ to indicate that f is a total function from A to B. Set A is said to be the *domain* of f while B is the *codomain* of f. We can also write $dom(f) = A$ and $codom(f) = B$. □

Convention 2.12 (Relations as functions): For sets A and B and relation $E \subseteq A \times B$, we can interpret E as a function $E \in A \longrightarrow \mathcal{P}(B)$ defined as $E(a) = \{ b \mid (a, b) \in E \}$, or as a function $E \in \mathcal{P}(A) \longrightarrow \mathcal{P}(B)$ defined as $E(A') = \{ b \mid (\exists\ a : a \in A' : (a, b) \in E) \}$. □

Notation 2.13 (Naturals and reals): We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of all natural numbers, and the set of all real numbers respectively. Define $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. We will also define $[i, j) = \{ k \mid i \leq k < j \wedge k \in \mathbb{N} \}$, $(i, j] = \{ k \mid i < k \leq j \wedge k \in \mathbb{N} \}$, $[i, j] = [i, j) \cup (i, j]$, and $(i, j) = [i, j) \cap (i, j]$. □

Definition 2.14 (Relation composition): Given sets A, B, C (not necessarily different) and two relations, $E \subseteq A \times B$ and $F \subseteq B \times C$, we define relation composition (infix operator $\circ$) as

$$E \circ F = \{ (a, c) \mid (\exists\ b : b \in B : (a, b) \in E \wedge (b, c) \in F) \}$$

□

We will also use the symbol $\circ$ for the composition of functions.

Remark 2.15: Note that function composition is different from relation composition. This can cause confusion when we are viewing relations as functions. For this reason, it will be clear from the context which type of composition is intended. □

Definition 2.16 (Relation exponentiation): Given set A and relation $E \subseteq A \times A$, we define relation exponentiation recursively as: $E^0 = I_A$ (where I_A is the identity relation on A) and $E^k = E \circ E^{k-1}$ ($k \geq 1$). Note that I_A is the unit of $\circ$. □

Definition 2.17 (Closure operator on relations): Given set A and relation $E \subseteq A \times A$, we define the $*$-closure of E as:

$$E^* = (\cup\ i : 0 \leq i : E^i)$$

Note, if E is symmetrical then E^* is an equivalence relation. □

Definition 2.18 (Idempotence): For any set A and function $f \in A \longrightarrow A$, we say that f is *idempotent* if $f \circ f = f$. □

Notation 2.19 (Alternation expressions): For any given predicate P, we use the following shorthand

$$\textbf{if } P \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi} = \begin{cases} e_1 & \text{if } P \\ e_2 & \text{otherwise} \end{cases}$$

□

Definition 2.20 (Minimum and maximum): Define **max** and **min** to be infix binary functions on integers such that

$$i \,\mathbf{max}\, j = \textbf{if } i \geq j \textbf{ then } i \textbf{ else } j \textbf{ fi}$$

$$i \,\mathbf{min}\, j = \textbf{if } i \leq j \textbf{ then } i \textbf{ else } j \textbf{ fi}$$

□

Recall from Notation 2.6 that **max** and **min** have as units $-\infty$ and $+\infty$, respectively.

We will sometimes define functions with codomain $\mathbb{N}$; these functions will frequently have definitions involving **MIN** or **MAX** quantifications, which can have value $+\infty$ or $-\infty$ (respectively). For notational convenience, we assume $+\infty, -\infty \in \mathbb{N}$.

Property 2.21 (Conjunction and disjunction in MIN quantifications): For predicates P, P' and integer function f we have the following two properties:

$$\begin{array}{rcl} (\mathbf{MIN}\ i : P(i) \wedge P'(i) : f(i)) & \geq & (\mathbf{MIN}\ i : P(i) : f(i))\,\mathbf{max}(\mathbf{MIN}\ i : P'(i) : f(i)) \\ (\mathbf{MIN}\ i : P(i) \vee P'(i) : f(i)) & = & (\mathbf{MIN}\ i : P(i) : f(i))\,\mathbf{min}(\mathbf{MIN}\ i : P'(i) : f(i)) \end{array}$$

□

Property 2.22 (MIN and MAX quantifications with quantified ranges): Given that universal quantification over a finite domain is shorthand for conjunction, and existential quantification over a finite domain is shorthand for disjunction, we have the following general properties (where P is some range predicate, f is some integer function, and the $\forall$ and $\exists$ quantified j is over a finite domain):

$$\begin{array}{lcl}(\mathbf{MIN}\ i : (\forall\ j :: P(i,j)) : f(i)) & \geq & (\mathbf{MAX}\ j :: (\mathbf{MIN}\ i : P(i,j) : f(i))) \\ (\mathbf{MIN}\ i : (\exists\ j :: P(i,j)) : f(i)) & = & (\mathbf{MIN}\ j :: (\mathbf{MIN}\ i : P(i,j) : f(i)))\end{array}$$

□

Definition 2.23 (Precedence of operators): We specify that set operators have the following descending precedence: $\times$, $\cap$, and $\cup$. □

Definition 2.24 (Tuple projection operators): For an n-tuple $t = (x_1, \ldots, x_n)$ we use the notation $\pi_i(t)$ $(1 \leq i \leq n)$ to denote tuple element x_i; we use the notation $\bar{\pi}_i(t)$ $(1 \leq i \leq n)$ to denote the $(n-1)$-tuple $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. Both π_i and $\bar{\pi}_i$ extend naturally to sets of tuples. □

Convention 2.25 (Tuple arguments to functions): For functions (or predicates) taking a single tuple as an argument, we usually drop one set of parentheses in a function application. □

Definition 2.26 (Tuple and relation reversal): For an n-tuple $(x_1, x_2, \ldots, x_n)$ define *reversal* as function R given by:

$$(x_1, x_2, \ldots, x_n)^R = (x_n, \ldots, x_2, x_1)$$

□

Forward reference 2.27: We will also be defining reversal of strings (in Definition 2.40). These operators extend naturally to sets of tuples (relations) and to sets of strings (languages). A reversal operator is usually written as a postfix and superscript operator; however, we will sometimes write it as a normal function. Reversal operators are their own inverses. In subsequent sections of this dissertation, we will also be defining reversal operators for more complex structures, such as finite automata and regular expressions. □

Definition 2.28 (Dual of a function): We assume two sets A and B whose reversal operators are R_A and R_B respectively. Two functions $f \in A \longrightarrow B$ and $f_d \in A \longrightarrow B$ are one another's *dual* if and only if

$$f(a)^{R_B} = f_d(a^{R_A})$$

□

Definition 2.29 (Symmetrical function): A *symmetrical* function is one that is its own dual. □

Proposition 2.30 (Symmetrical functions): The composition of two symmetrical functions is again symmetrical. □

Notation 2.31 (Equivalence classes of an equivalence relation): For any equivalence relation E on set A, we use $[a]_E$ to denote the set $\{ b \mid (a, b) \in E \}$. Note that $a \in [a]_E$. We also denote the set of equivalence classes of E by $[A]_E$; that is

$$[A]_E = \{ [a]_E \mid a \in A \}$$

The set $[A]_E$ is also called the *partition* of A *induced* by E. Since all partitions are induced by a unique equivalence relation, we will sometimes also refer to *the* equivalence relation inducing a particular partition. □

Definition 2.32 (Index of an equivalence relation): For equivalence relation E on set A, define $\sharp E = |[A]_E|$ (i.e. the number of equivalence classes under E). $\sharp E$ is called the *index* of E. □

Definition 2.33 (Refinement of an equivalence relation): For equivalence relations E and E' (on set A), E is a *refinement* of E' (written $E \sqsubseteq E'$) if and only if $E \subseteq E'$. An equivalent statement is that $E \sqsubseteq E'$ if and only if every equivalence class (of A) under E is entirely contained in some equivalence class (of A) under E'. □

Definition 2.34 (Refinement relation on partitions): We can also extend our refinement relation to partitions. For equivalence relations E and E' (on set A), we write $[A]_E \sqsubseteq [A]_{E'}$ if and only if $E \sqsubseteq E'$. □

Property 2.35 (Equivalence relations): Given two equivalence relations E, F of finite index, we have the following property:

$$(E \sqsubseteq F) \wedge (\sharp E = \sharp F) \Rightarrow (E = F)$$

□

Definition 2.36 (Complement of a relation): Given two sets (not necessarily distinct) A and B, and relation $E \subseteq A \times B$ we define the complement of relation E (written $\neg E$) as $\neg E = (A \times B) \setminus E$. □

Definition 2.37 (Preserving a predicate): A function $f \in B^n \longrightarrow B$ (for fixed $n \geq 1$) is said to *preserve* predicate (or property) P (on B) if and only if

$$(\forall\ b : b \in B^n \cap (dom(f)) \wedge (\forall\ k : 1 \leq k \leq n : P(\pi_k(b))) : P(f(b)))$$

□

Intuitively, a function f preserves a property P if, when every argument of f satisfies P, the result of f applied to the arguments also satisfies P.

2.3 Strings and languages

In this section, we present a number of definitions relating to strings and languages.

Definition 2.38 (Set of all strings): Given alphabet V, we define V^* to be the set of all strings over V. For more on this, see Definition 2.48. □

Notation 2.39 (Empty string): We will use ε to denote the string of length 0 (the empty string). Some authors use ϵ or λ to denote the empty string. □

Definition 2.40 (String reversal function R): Assuming alphabet V, we define string reversal function R recursively as $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$ (for $a \in V, w \in V^*$). □

Definition 2.41 (String operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright$): Assuming alphabet V, we define four (infix) operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright \in V^* \times \mathbb{N} \longrightarrow V^*$ as follows:

- $w \upharpoonleft k$ is the $k \,\mathbf{min}\, |w|$ leftmost symbols of w
- $w \downharpoonleft k$ is the $(|w| - k) \,\mathbf{max}\, 0$ rightmost symbols of w
- $w \upharpoonright k$ is the $k \,\mathbf{min}\, |w|$ rightmost symbols of w
- $w \downharpoonright k$ is the $(|w| - k) \,\mathbf{max}\, 0$ leftmost symbols of w

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

Property 2.42 (String operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright$): Note that

$$(w \upharpoonleft k)(w \downharpoonleft k) = w$$

and

$$(w \downharpoonright k)(w \upharpoonright k) = w$$

□

Example 2.43 (String operators $\upharpoonleft, \downharpoonleft, \upharpoonright, \downharpoonright$): $(baab)\upharpoonleft 3 = baa$, $(baab)\downharpoonleft 1 = aab$, $(baab)\upharpoonright 5 = baab$, and $(baab)\downharpoonright 10 = \varepsilon$. □

Definition 2.44 (Language): Given alphabet V, any subset of V^* is a *language* over V. □

Definition 2.45 (Concatenation of languages): Language concatenation is an infix operator $\cdot \in \mathcal{P}(V^*) \times \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (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 2.46 (Concatenation of languages): We will frequently use juxtaposition instead of writing operator $\cdot$ (i.e. we use LL' instead of $L \cdot L'$). For language L and string w, we take Lw to mean $L\{w\}$. $\square$

Definition 2.47 (Language exponentiation): We define language exponentiation recursively as follows (for language L): $L^0 = \{\varepsilon\}$ and $L^k = LL^{k-1}$ $(k \geq 1)$. $\square$

Note that V^k is the set of all strings of length k over alphabet V.

Definition 2.48 (Closure operators on languages): We define two postfix and superscript operators on languages over alphabet V. Operator $* \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ (known as Kleene closure) is

$$L^* = (\cup\ i : 0 \leq i : L^i)$$

and operator $+ \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ is

$$L^+ = (\cup\ i : 1 \leq i : L^i)$$

Note that $L^* = L^+ \cup \{\varepsilon\}$. $\square$

The language V^* is the set of all strings over alphabet V.

Definition 2.49 (Unary language operators $\neg$ and ?): Assuming an alphabet V, prefix operator $\neg \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ is defined as

$$\neg L = V^* \setminus L$$

while postfix and superscript operator $? \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ is defined as

$$L^? = L \cup \{\varepsilon\}$$

$\square$

Definition 2.50 (Functions pref and suff): For any given alphabet V, define $\mathbf{pref} \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ and $\mathbf{suff} \in \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ as

$$\mathbf{pref}(L) = (\cup\ x, y : xy \in L : \{x\})$$

and

$$\mathbf{suff}(L) = (\cup\ x, y : xy \in L : \{y\})$$

Intuitively, $\mathbf{pref}(L)$ (respectively $\mathbf{suff}(L)$) is the set of all strings which are (not necessarily proper) prefixes (respectively suffixes) of strings in L. $\square$

Property 2.51 (Idempotence of pref and suff): It follows from their definitions that **pref** and **suff** are both idempotent. $\square$

Property 2.52 (Duality of pref and suff): Functions **pref** and **suff** are duals of one another. This can be seen as follows:

$$
\begin{array}{ll}
& \mathbf{pref}(L^R) \\
= & \quad \{\text{ definition of } \mathbf{pref} \,\} \\
& (\cup\ x, y : xy \in L^R : \{x\}) \\
= & \quad \{\ xy \in L^R \equiv (xy)^R \in L\ \} \\
& (\cup\ x, y : (xy)^R \in L : \{x\}) \\
= & \quad \{\ (xy)^R = y^R x^R\ \} \\
& (\cup\ x, y : y^R x^R \in L : \{x\}) \\
= & \quad \{\text{ change of bound variable: } x' = x^R,\ y' = y^R\ \} \\
& (\cup\ x', y' : y'x' \in L : \{x'^R\}) \\
= & \quad \{\ R \text{ distributes over } \cup\ \} \\
& (\cup\ x', y' : y'x' \in L : \{x'\})^R \\
= & \quad \{\text{ definition of } \mathbf{suff} \,\} \\
& \mathbf{suff}(L)^R
\end{array}
$$

□

Notation 2.53 (String arguments to functions pref and suff): For string $w \in V^*$, we will write **pref**(w) instead of **pref**$(\{w\})$ (and likewise for **suff**). □

Property 2.54 (Function suff): For non-empty language L and alphabet symbol $a \in V$, function **suff** has the property:

$$\mathbf{suff}(La) = \mathbf{suff}(L)a \cup \{\varepsilon\}$$

□

Property 2.55 (Non-empty languages and pref, suff): For any $L \neq \emptyset$, $\varepsilon \in \mathbf{pref}(L)$ and $\varepsilon \in \mathbf{suff}(L)$. □

Property 2.56 (Intersection and pref): Given languages $A, B \subseteq V^*$ and string $y \in V^*$ we have the following property:

$$A\{y\} \cap B = (A \cap \mathbf{pref}(B))\{y\} \cap B$$

□

Definition 2.57 (Prefix and suffix partial orderings $\leq_p$ and $\leq_s$): For any given alphabet V, partial orders $\leq_p$ and $\leq_s$ over $V^* \times V^*$ are defined as $u \leq_p v \equiv u \in \mathbf{pref}(v)$ and $u \leq_s v \equiv u \in \mathbf{suff}(v)$. □

Definition 2.58 (Operator $\mathbf{max}_{\leq_s}$): In a manner analogous to integer operator **max**, we define binary infix operator $\mathbf{max}_{\leq_s}$ on strings as (provided $x \leq_s y \vee y \leq_s x$):

$$x \, \mathbf{max}_{\leq_s} y = \mathbf{if} \ x \leq_s y \ \mathbf{then} \ y \ \mathbf{else} \ x \ \mathbf{fi}$$

This operator is also associative and commutative. The unit of $\mathbf{max}_{\leq_s}$ is ε, which will be used in Section 4.3 when we consider quantifications involving $\mathbf{max}_{\leq_s}$.

We could have given an analogous operator for the prefix ordering; since it would not be used in this dissertation, we do not define it here. □

Property 2.59 (Function suff): If A and B are languages, then

$$\mathbf{suff}(A) \cap B \neq \emptyset \equiv A \cap V^*B \neq \emptyset$$

□

Property 2.60 (Language intersection): If A and B are languages over alphabet V and $a \in V$, then

$$V^*A \cap V^*B \neq \emptyset \equiv V^*A \cap B \neq \emptyset \vee A \cap V^*B \neq \emptyset$$

and

$$V^*aA \cap V^*B \neq \emptyset \equiv V^*aA \cap B \neq \emptyset \vee A \cap V^*B \neq \emptyset$$

□

2.4 Regular expressions

In this section, we present some definitions and properties relating to regular expressions.

Definition 2.61 (Regular expressions and their languages): We simultaneously define regular expressions over alphabet V (the set RE) and the languages they denote (given by function $\mathcal{L}_{RE} \in RE \longrightarrow \mathcal{P}(V^*)$) as follows:

- $\varepsilon \in RE$ and $\mathcal{L}_{RE}(\varepsilon) = \{\varepsilon\}$
- $\emptyset \in RE$ and $\mathcal{L}_{RE}(\emptyset) = \emptyset$
- For all $a \in V$, $a \in RE$ and $\mathcal{L}_{RE}(a) = \{a\}$
- For $E, F \in RE$
  - $E \cup F \in RE$ and $\mathcal{L}_{RE}(E \cup F) = \mathcal{L}_{RE}(E) \cup \mathcal{L}_{RE}(F)$
  - $E \cdot F \in RE$ and $\mathcal{L}_{RE}(E \cdot F) = \mathcal{L}_{RE}(E) \cdot \mathcal{L}_{RE}(F)$
  - $E^* \in RE$ and $\mathcal{L}_{RE}(E^*) = \mathcal{L}_{RE}(E)^*$

– $E^+ \in RE$ and $\mathcal{L}_{RE}(E^+) = \mathcal{L}_{RE}(E)^+$
– $E^? \in RE$ and $\mathcal{L}_{RE}(E^?) = \mathcal{L}_{RE}(E)^?$

- Nothing else is in RE

Operators $*, +, ?$ have the highest precedence, followed by $\cdot$, and finally $\cup$. □

Some authors write $|$ or $+$ instead of $\cup$.

Note that regular expressions are syntactic objects which denote languages, even though they may look the same (as in the cases of $\emptyset$, ε, and $a \in V$). Whether we are dealing with regular expressions or languages will be clear from the context.

Definition 2.62 (RE reversal): Regular expression reversal is given by the postfix (superscript) function $R \in RE \longrightarrow RE$

$$\begin{array}{lcll}
\varepsilon^R & = & \varepsilon & \\
\emptyset^R & = & \emptyset & \\
a^R & = & a & \text{(for all } a \in V) \\
(E_0 \cup E_1)^R & = & (E_0^R) \cup (E_1^R) & \\
(E_0 \cdot E_1)^R & = & (E_1^R) \cdot (E_0^R) & \\
(E^*)^R & = & (E^R)^* & \\
(E^+)^R & = & (E^R)^+ & \\
(E^?)^R & = & (E^R)^? &
\end{array}$$

Function R satisfies the obvious property that

$$(\forall\ E : E \in RE : (E^R)^R = E \wedge (\mathcal{L}_{RE}(E^R))^R = \mathcal{L}_{RE}(E))$$

□

Remark 2.63: The property satisfied by regular expression reversal implies that function $\mathcal{L}_{RE}$ is symmetrical. □

Definition 2.64 (Regular languages): The set of all *regular languages* over alphabet V are defined as:

$$\{\, \mathcal{L}_{RE}(E) \mid E \in RE \,\}$$

□

Remark 2.65: The set of regular languages could have been defined in a number of other (equivalent) ways, for example, as the set of all languages accepted by some finite automaton. □

2.5 Trees

In this section, we present a number of tree-related definitions. We give a definition of trees which is slightly different than the traditional recursive one: we use the tree domain approach to trees, which will allow us to easily access individual nodes of a tree.

Notation 2.66 (Strings over $\mathbb{N}_+$): In order to designate particular nodes of a tree, we will be using strings over $\mathbb{N}_+$ (even though it is not finite, and therefore not an alphabet). To avoid confusion when writing such a string (element of ${\mathbb{N}_+}^*$), we make the concatenation operator explicit. Instead of writing it as the dot, for clarity we write it as $\diamond$. Furthermore, we will write the empty string ε as 0. $\square$

Definition 2.67 (Tree domain): A *tree domain* D is a non-empty subset of ${\mathbb{N}_+}^*$ such that the following two conditions hold:

1. D is prefix-closed: $\mathbf{pref}(D) \subseteq D$.
2. For all $x \in {\mathbb{N}_+}^*$ and $i, j \in \mathbb{N}_+$: $x \diamond j \in D \wedge i < j \Rightarrow x \diamond i \in D$.

$\square$

Example 2.68 (Tree domain): The set $\{0, 1, 2, 2 \diamond 1, 2 \diamond 2\}$ is a tree domain. The set $\{0, 1, 2 \diamond 1, 2 \diamond 2\}$ is not a tree domain since it is not prefix-closed (it does not contain 2). The set $\{0, 1, 2, 2 \diamond 2\}$ is not a tree domain since it does not satisfy the second requirement (it should also contain $2 \diamond 1$ since it contains $2 \diamond 2$ and $1 < 2$). $\square$

Definition 2.69 (Ranked alphabet): A *ranked alphabet* is a pair (V, r) such that V is an alphabet and $r \in V \longrightarrow \mathbb{N}$. $r(a)$ is called the rank of symbol a. Define $V_n = r^{-1}(n)$. Symbols of rank 0 are called *nullary* symbols, while those of rank 1 are called *unary* symbols. $\square$

Example 2.70 (Ranked alphabet): The pair $(\{a, b\}, \{(a, 2), (b, 0)\})$ is a ranked alphabet (with a as binary symbol and b as nullary symbol). We also have $V_0 = \{b\}$ and $V_2 = \{a\}$. There are no unary symbols. $\square$

Definition 2.71 (Tree): Let (V, r) be a ranked alphabet. A *tree* A over (V, r) is a function $A \in D \longrightarrow V$ (where D is a tree domain) such that

$$(\forall\ a : a \in D : r(A(a)) = (\mathbf{MAX}\ i : i \in \mathbb{N} \wedge a \diamond i \in D : i))$$

Set D (equivalently $dom(A)$) are called the *nodes* of tree A. $A(a)$ is the *label* of a. $\square$

Definition 2.72 (Set *Trees*): Define $Trees(V, r)$ to be the set of all trees over ranked alphabet (V, r). $\square$

Definition 2.73 (Nodes of a tree): Assuming a tree A, we can make the following definitions:

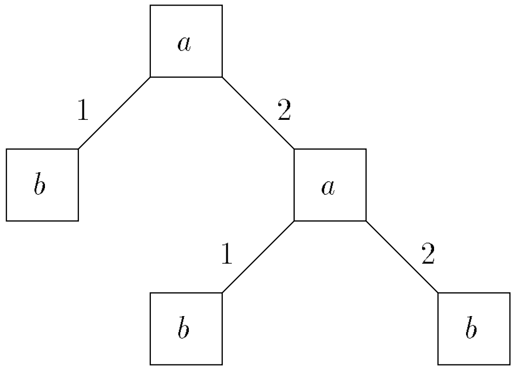


Figure 2.1: An example of a tree

- Node 0 is called the *root* of A.
- A node with a nullary label is a *leaf*.
- Nodes that are not leaves are *internal* nodes.

$\square$

Example 2.74 (Tree): Using the tree domain from Example 2.68 and the ranked alphabet from Example 2.70, we can give the following tree (in tabular form):

Node	0	1	2	$2 \diamond 1$	$2 \diamond 2$
Label	a	b	a	b	b

The tree can also be presented in the more usual graphical form, as in Figure 2.1. $\square$

2.6 Finite automata and Moore machines

In this section, we define finite automata, Moore machines, some of their properties, and some transformations on them.

Definition 2.75 (Finite automaton): A finite automaton (also known as an *FA*) is a 6-tuple (Q, V, T, E, S, F) where

- Q is a finite set of states.
- V is an alphabet.
- $T \in \mathcal{P}(Q \times V \times Q)$ is a transition relation.
- $E \in \mathcal{P}(Q \times Q)$ is an ε-transition relation.

- $S \subseteq Q$ is a set of start states.
- $F \subseteq Q$ is a set of final states.

□

Remark 2.76: We will take some liberty in our interpretation of the signatures of the transition relations. For example, we also use the signatures $T \in V \longrightarrow \mathcal{P}(Q \times Q)$, $T \in Q \times Q \longrightarrow \mathcal{P}(V)$, $T \in Q \times V \longrightarrow \mathcal{P}(Q)$, $T \in Q \longrightarrow \mathcal{P}(V \times Q)$, $T \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(Q)$, and $E \in Q \longrightarrow \mathcal{P}(Q)$. In each case, the order of the Qs from left to right will be preserved; for example, the function $T \in Q \longrightarrow \mathcal{P}(V \times Q)$ is defined as $T(p) = \{\, (a, q) \mid (p, a, q) \in T \,\}$. The signature that is used will be clear from the context. □

Remark 2.77: Our definition of finite automata differs from the traditional approach in two ways:

- Multiple start states are permitted.
- The ε-transitions (relation E) are separate from transitions on alphabet symbols (relation T).

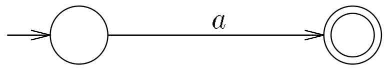
□

Since we only consider *finite* automata in this dissertation, we will frequently simply use the term *automata*.

Convention 2.78 (Finite automaton state graphs): When drawing the state graph corresponding to a finite automaton, we adopt the following conventions:

- All states are drawn as circles or ovals (vertices).
- Transitions are drawn as labeled (with ε or alphabet symbol $a \in V$) directed edges between states.
- Start states have an in-transition with no source (the transition does not come from another state).
- Final states are drawn as two concentric circles or ovals.

For example, the *FA* below has two states, one is the start state, and other is the final state, with a transition on a:



□

Definition 2.79 (Moore machine): A Moore machine (also known as an *MM*) is a 6-tuple (Q, V, W, T, λ, S) where

- Q is a finite set of states.
- V is an input alphabet.
- W is an output alphabet.
- $T \in \mathcal{P}(Q \times V \times Q)$ is a transition relation.
- $\lambda \in Q \longrightarrow W$ is an output function.
- $S \subseteq Q$ is a set of start states.

□

Note that an *MM* does not have any ε-transitions, as this would complicate the definition of the *transduction* of the *MM* (transduction will be defined later).

2.6.1 Definitions and properties involving *FA*s and *MM*s

In this subsection we define some properties of finite automata. Many of these are easily extended to Moore machines, and we do not present the Moore machine versions. To make these definitions more concise, we introduce particular finite automata $M = (Q, V, T, E, S, F)$, $M_0 = (Q_0, V_0, T_0, E_0, S_0, F_0)$, and $M_1 = (Q_1, V_1, T_1, E_1, S_1, F_1)$.

Definition 2.80 (Size of an *FA*): Define the size of an *FA* as $|M| = |Q|$. □

Definition 2.81 (Isomorphism ($\cong$) of *FA*s): We define isomorphism ($\cong$) as an equivalence relation on *FA*s. M_0 and M_1 are isomorphic (written $M_0 \cong M_1$) if and only if $V_0 = V_1$ and there exists a bijection $g \in Q_0 \longrightarrow Q_1$ such that

- $T_1 = \{ (g(p), a, g(q)) \mid (p, a, q) \in T_0 \}$,
- $E_1 = \{ (g(p), g(q)) \mid (p, q) \in E_0 \}$,
- $S_1 = \{ g(s) \mid s \in S_0 \}$, and
- $F_1 = \{ g(f) \mid f \in F_0 \}$.

□

Definition 2.82 (Extending the transition relation T): We extend transition function $T \in V \longrightarrow \mathcal{P}(Q \times Q)$ to $T^* \in V^* \longrightarrow \mathcal{P}(Q \times Q)$ as follows:

$$T^*(\varepsilon) = E^*$$

and (for $a \in V, w \in V^*$)

$$T^*(aw) = E^* \circ T(a) \circ T^*(w)$$

This definition could also have been presented symmetrically. □

Remark 2.83: We sometimes also use the signature $T^* \in Q \times Q \longrightarrow \mathcal{P}(V^*)$. □

Remark 2.84: If $E = \emptyset$ then $E^* = \emptyset^* = I_Q$ where I_Q is the identity relation on the states of M. □

Definition 2.85 (The language between states): The language between any two states $q_0, q_1 \in Q$ is $T^*(q_0, q_1)$. Intuitively, $T^*(q_0, q_1)$ is the set of all words on paths from q_0 to q_1. □

Definition 2.86 (Left and right languages): The left language of a state (in M) is given by function $\overleftarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overleftarrow{\mathcal{L}}_M(q) = (\cup\ s : s \in S : T^*(s, q))$$

The right language of a state (in M) is given by function $\overrightarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overrightarrow{\mathcal{L}}_M(q) = (\cup\ f : f \in F : T^*(q, f))$$

The subscript M is usually dropped when no ambiguity can arise. □

Definition 2.87 (Language of an *FA*): The language of a finite automaton (with alphabet V) is given by the function $\mathcal{L}_{FA} \in FA \longrightarrow \mathcal{P}(V^*)$ defined as:

$$\mathcal{L}_{FA}(M) = (\cup\ s, f : s \in S \wedge f \in F : T^*(s, f))$$

□

Property 2.88 (Language of an *FA*): From the definitions of left and right languages (of a state), we can also write:

$$\mathcal{L}_{FA}(M) = (\cup\ f : f \in F : \overleftarrow{\mathcal{L}}(f))$$

and

$$\mathcal{L}_{FA}(M) = (\cup\ s : s \in S : \overrightarrow{\mathcal{L}}(s))$$

□

Definition 2.89 (Extension of $\mathcal{L}_{FA}$): Function $\mathcal{L}_{FA}$ is extended to $[FA]_{\cong}$ as $\mathcal{L}_{FA}([M]_{\cong}) = \mathcal{L}_{FA}(M)$. The choice of representative is irrelevant, as isomorphic *FA*s accept the same language. □

Definition 2.90 (*Complete*): A *Complete* finite automaton is one satisfying the following:

$$Complete(M) \equiv (\forall\ q, a : q \in Q \wedge a \in V : T(q, a) \neq \emptyset)$$

Intuitively, an *FA* is *Complete* when there is at least one out transition from every state on every symbol in the alphabet. □

Property 2.91 (*Complete*)**:** For all *Complete FA*s (Q, V, T, E, S, F):

$$(\cup\ q : q \in Q : \overleftarrow{\mathcal{L}}(q)) = V^*$$

□

Instead of accepting a string (as *FA*s do), a Moore machine *transduces* the string. The transduction of a string is defined as follows.

Definition 2.92 (Transduction by a *MM*): Given Moore machine $N = (Q, V, W, T, \lambda, S)$ we define transduction helper function $\mathcal{H}_N \in \mathcal{P}(Q) \times V^* \longrightarrow (\mathcal{P}(W))^*$ as

$$\mathcal{H}_N(Q', \varepsilon) = \varepsilon$$

and (for $a \in V$, $w \in V^*$)

$$\mathcal{H}_N(Q', aw) = \{\ \lambda(q) \mid q \in T(Q', a)\ \} \cdot \mathcal{H}_N(T(Q', a), w)$$

The transduction of a string is given by function $\Theta \in V^* \longrightarrow (\mathcal{P}(W))^*$ defined as

$$\Theta_N(w) = \{\ \lambda(s) \mid s \in S\ \} \cdot \mathcal{H}_N(S, w)$$

The codomains of $\mathcal{H}_N$ and Θ_N are strings over alphabet $\mathcal{P}(W)$. Note that both functions depend upon N. □

Definition 2.93 (*ε-free*)**:** Automaton M is *ε-free* if and only if $E = \emptyset$. □

Remark 2.94: Even if M is *ε-free* it is still possible that $\varepsilon \in \mathcal{L}_{FA}(M)$: in this case $S \cap F \neq \emptyset$. □

Definition 2.95 (Reachable states): For $M \in FA$ we can define a reachability relation

$$Reach(M) \subseteq Q \times Q$$

as

$$Reach(M) = (\bar{\pi}_2(T) \cup E)^*$$

(In this definition, we have simply projected away the symbol component of the transition relation. State p reaches state q if and only if there is an ε-transition or a symbol transition from p to q.) Similarly, the set of start-reachable states is defined to be (here, we interpret the relation as a function $Reach(M) \in \mathcal{P}(Q) \longrightarrow \mathcal{P}(Q)$):

$$SReachable(M) = Reach(M)(S)$$

and the set of final-reachable states is defined to be:

$$FReachable(M) = (Reach(M))^R(F)$$

The set of useful states is:

$$Reachable(M) = SReachable(M) \cap FReachable(M)$$

□

Property 2.96 (Reachability): For automaton $M = (Q, V, T, E, S, F)$, *SReachable* satisfies the following interesting property:

$$q \in \mathit{SReachable}(M) \equiv \overleftarrow{\mathcal{L}}_M(q) \neq \emptyset$$

FReachable satisfies a similar property:

$$q \in \mathit{FReachable}(M) \equiv \overrightarrow{\mathcal{L}}_M(q) \neq \emptyset$$

□

Definition 2.97 (Useful automaton): A *Useful* finite automaton M is one with only reachable states:

$$\mathit{Useful}(M) \equiv (Q = \mathit{Reachable}(M))$$

□

Definition 2.98 (Start-useful automaton): A Useful_s finite automaton M is one with only start-reachable states:

$$\mathit{Useful}_s(M) \equiv (Q = \mathit{SReachable}(M))$$

□

Definition 2.99 (Final-useful automaton): A Useful_f finite automaton M is one with only final-reachable states

$$\mathit{Useful}_f(M) \equiv (Q = \mathit{FReachable}(M))$$

□

Remark 2.100: Useful_s and Useful_f are closely related by *FA* reversal (to be presented in Transformation 2.113). For all $M \in \mathit{FA}$ we have $\mathit{Useful}_f(M) \equiv \mathit{Useful}_s(M^R)$. □

Property 2.101 (Implication of Useful_f): Useful_f has the property:

$$\mathit{Useful}_f(M) \Rightarrow (\forall\ q : q \in Q : \overleftarrow{\mathcal{L}}(q) \subseteq \mathbf{pref}(\mathcal{L}_{FA}(M)))$$

□

Definition 2.102 (Deterministic finite automaton): A finite automaton M is deterministic if and only if

- It has one start state or no start states.
- It is ε-*free*.

- Transition function $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ does not map pairs in $Q \times V$ to multiple states.

Formally,

$$Det(M) \equiv |S| \leq 1 \wedge \varepsilon\text{-}free(M) \wedge (\forall\, q, a : q \in Q \wedge a \in V : |T(q, a)| \leq 1)$$

□

Definition 2.103 (Deterministic *FA*s): *DFA* denotes the set of all deterministic finite automata. We call $FA \setminus DFA$ the set of *nondeterministic finite automata.* □

Definition 2.104 (Deterministic Moore machine): The definition of a deterministic Moore machine is similar to that of a *DFA*. A Moore machine $N = (Q, V, W, T, \lambda, S)$ is deterministic if and only if

- It has one start state or no start states.
- Transition function $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ does not map pairs in $Q \times V$ to multiple states.

Formally,

$$Det(N) \equiv |S| \leq 1 \wedge (\forall\, q, a : q \in Q \wedge a \in V : |T(q, a)| \leq 1)$$

□

Definition 2.105 (Deterministic *MM*s): *DMM* denotes the set of all deterministic Moore machines. □

Convention 2.106 (Transition function of a *DFA*): For $(Q, V, T, \emptyset, S, F) \in DFA$ we can consider the transition function to have signature $T \in Q \times V \not\longrightarrow Q$. The transition function is total if and only if the *DFA* is *Complete*. □

Property 2.107 (Weakly deterministic automaton): Some authors use a property of a deterministic automaton that is weaker than *Det*; it uses left languages and is defined as follows:

$$Det'(M) \equiv (\forall\, q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overleftarrow{\mathcal{L}}(q_0) \cap \overleftarrow{\mathcal{L}}(q_1) = \emptyset)$$

□

Remark 2.108: $Det(M) \Rightarrow Det'(M)$ is easily proved. We can also demonstrate that there exists an $M \in FA$ such that $Det'(M) \wedge \neg Det(M)$; namely

$$(\{q_0, q_1\}, \{b\}, \{(q_0, b, q_0), (q_0, b, q_1)\}, \emptyset, \emptyset, \emptyset)$$

In this *FA*, $\overleftarrow{\mathcal{L}}(q_0) = \overleftarrow{\mathcal{L}}(q_1) = \emptyset$, but state q_0 has two out-transitions on symbol b. □

Definition 2.109 (Minimality of a *DFA*): An $M \in DFA$ is minimal as follows:

$$Min(M) \equiv (\forall\ M' : M' \in DFA \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate *Min* is defined only on *DFA*s. Some later definitions are simpler if we define a minimal, but *Complete*, *DFA* as follows $Min_C(M) \equiv$

$$(\forall\ M' : M' \in DFA \wedge Complete(M') \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate Min_C is defined only on *Complete DFA*s. □

Property 2.110 (Minimality of a *DFA*): An M, such that $Min(M)$, is the unique (modulo $\cong$) minimal *DFA*. □

Minimality of *DMM*s is discussed in Section 4.3.3 on page 68.

Property 2.111 (An alternate definition of minimality of a *DFA*): For minimizing a *DFA*, we use the predicate $Minimal(Q, V, T, \emptyset, S, F) \equiv$

$$\begin{aligned}&(\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1))\\ &\wedge\ Useful(Q, V, T, \emptyset, S, F)\end{aligned}$$

(This predicate is defined only on *DFA*s.) A similar predicate (relating to Min_C) is $Minimal_C(Q, V, T, \emptyset, S, F) \equiv$

$$\begin{aligned}&(\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1))\\ &\wedge\ Useful_s(Q, V, T, \emptyset, S, F) \wedge Complete(Q, V, T, \emptyset, S, F)\end{aligned}$$

(This predicate is only defined on *Complete DFA*s.)

We have the property that (for all $M, M_C \in DFA$ such that $Complete(M_C)$)

$$Minimal(M) \equiv Min(M) \wedge Minimal_C(M_C) \equiv Min_C(M_C)$$

Proof:
For brevity, we only prove $Minimal_C(M) \equiv Min_C(M)$. In order to prove $Min_C(M) \Rightarrow Minimal_C(M)$, consider its contrapositive $\neg Minimal_C(M) \Rightarrow \neg Min_C(M)$.

Since $\neg Minimal_C(M)$, at least one of the following holds:

- There is a start-unreachable state ($\neg Useful_s(M)$) which can be removed, meaning that $\neg Min_C(M)$.
- $\neg Complete(M)$. It follows that $\neg Min_C(M)$.
- There are two states p, q such that $\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$. In this case, the in-transitions to q can be redirected to p, and q can be entirely eliminated, meaning that $\neg Min_C(M)$.

We now continue with the proof that $Minimal_C(M) \Rightarrow Min_C(M)$. The following proof is by contradiction. Assume a *DFA* $M = (Q, V, T, \emptyset, S, F)$ such that $Minimal_C(M)$. For the contradiction, we assume the existence of a *Complete DFA* $M' = (Q', V, T', \emptyset, S', F')$ such that $\mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M')$ and $|M'| < |M|$. Since both M and M' are *Complete*, the sets of their left languages, $\{ \overleftarrow{\mathcal{L}}(q) \mid q \in Q \}$ and $\{ \overleftarrow{\mathcal{L}}(q) \mid q \in Q' \}$, are partitions of V^* with

$$|M'| = |\{ \overleftarrow{\mathcal{L}}(q) \mid q \in Q' \}| < |\{ \overleftarrow{\mathcal{L}}(q) \mid q \in Q \}| = |M|$$

By the pigeonhole principle, there exist words x, y which are in the same equivalence class of the M' partition, but in different equivalence classes of the M partition. More precisely, there exist $x, y \in V^*$ such that $T'^*(S', x) = T'^*(S', y) \wedge T^*(S, x) \neq T^*(S, y)$.

However, $\overrightarrow{\mathcal{L}}(T^*(S, x)) \neq \overrightarrow{\mathcal{L}}(T^*(S, y))$ since $Minimal_C(M)$. It follows that there exists $z \in V^*$ such that

$$z \in \overrightarrow{\mathcal{L}}(T^*(S, x)) \not\equiv z \in \overrightarrow{\mathcal{L}}(T^*(S, y))$$

or, equivalently

$$xz \in \mathcal{L}_{FA}(M) \not\equiv yz \in \mathcal{L}_{FA}(M)$$

Returning to M', we have

$$T'^*(S', xz) = T'^*(T'^*(S', x), z) = T'^*(T'^*(S', y), z) = T'^*(S', yz)$$

and so

$$xz \in \mathcal{L}_{FA}(M') \equiv yz \in \mathcal{L}_{FA}(M')$$

This gives $\mathcal{L}_{FA}(M) \neq \mathcal{L}_{FA}(M')$, which is a contradiction. □

Two automata M and M' such that $\mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M')$, $\neg Complete(M) \wedge Complete(M')$, and $Minimal(M) \wedge Minimal_C(M)$ would be isomorphic except for the fact that M' would have a sink state.

Remark 2.112: In the literature the second conjunct in the definition of predicate $Minimal_C$ is sometimes erroneously omitted. The necessity of the conjunct can be seen by considering the *DFA*

$$(\{p, q\}, \{a\}, \{(p, a, p), (q, a, q)\}, \emptyset, \emptyset, \{p\})$$

Here $\overleftarrow{\mathcal{L}}(p) = \overleftarrow{\mathcal{L}}(q) = \emptyset$ (which is also the language of the *DFA*), $\overrightarrow{\mathcal{L}}(p) = \{a\}^*$, and $\overrightarrow{\mathcal{L}}(q) = \emptyset$. Without the second conjunct, this *DFA* would be considered $Minimal_C$; clearly this is not the case, as the minimal *Complete DFA* accepting $\emptyset$ is $(\emptyset, \{a\}, \emptyset, \emptyset, \emptyset, \emptyset)$. □

2.6.2 Transformations on finite automata

In this section, we present a number of transformations on finite automata. Many of these could also be defined for Moore machines, although they will not be needed.

Transformation 2.113 (*FA* reversal): *FA* reversal is given by postfix (superscript) function $R \in FA \longrightarrow FA$, defined as:

$$(Q, V, T, E, S, F)^R = (Q, V, T^R, E^R, F, S)$$

Function R satisfies

$$(\forall\ M : M \in FA : (\mathcal{L}_{FA}(M))^R = \mathcal{L}_{FA}(M^R)).$$

and preserves ε-*free* and *Useful*. □

Remark 2.114: The property $(\mathcal{L}_{FA}(M^R))^R = \mathcal{L}_{FA}(M)$ means that function $\mathcal{L}_{FA}$ is its own dual, and is therefore symmetrical. □

Transformation 2.115 (Useless state removal): There exists a function $useful \in FA \longrightarrow FA$ that removes states that are not reachable. A definition of this function is not given here explicitly, as it is not needed. Function *useful* satisfies

$$(\forall\ M : M \in FA : Useful(useful(M)) \wedge \mathcal{L}_{FA}(useful(M)) = \mathcal{L}_{FA}(M))$$

and preserves ε-*free*, *Useful*, *Det*, and *Min*. □

Transformation 2.116 (Removing start state unreachable states): Transformation $useful_s \in FA \longrightarrow FA$ removes those states that are not start-reachable. A definition is not given here, as it is not needed. Function $useful_s$ satisfies

$$(\forall\ M : M \in FA : Useful_s(useful_s(M)) \wedge \mathcal{L}_{FA}(useful_s(M)) = \mathcal{L}_{FA}(M))$$

and preserves *Complete*, ε-*free*, *Useful*, *Det*, and (trivially) *Min* and Min_C. □

Remark 2.117: A function $useful_f \in FA \longrightarrow FA$ could also be defined, removing states that are not final-reachable. Such a function is not needed in this dissertation. □

Transformation 2.118 (Completing an *FA*): Function $complete \in FA \longrightarrow FA$ takes an *FA* and makes it *Complete*. It satisfies the requirement that:

$$(\forall\ M : M \in FA : Complete(complete(M)) \wedge \mathcal{L}_{FA}(complete(M)) = \mathcal{L}_{FA}(M))$$

In general, this transformation adds a sink state to the *FA*. This transformation preserves ε-*free*, (trivially) *Complete*, *Det*, and Min_C. □

Transformation 2.119 (ε-transition removal): An ε-transition removal transformation $remove_\varepsilon \in FA \longrightarrow FA$ is one that satisfies

$$(\forall\ M : M \in FA : \varepsilon\text{-}free(remove_\varepsilon(M)) \wedge \mathcal{L}_{FA}(remove_\varepsilon(M)) = \mathcal{L}_{FA}(M))$$

There are several possible implementations of $remove_\varepsilon$. The most useful one (for deriving finite automata construction algorithms in Chapter 6) is:

$$\begin{array}{rcll} rem_\varepsilon(Q,V,T,E,S,F) & = & \textbf{let} & Q' = \{E^*(S)\} \cup \{\, E^*(q) \mid Q \times V \times \{q\} \cap T \neq \emptyset \,\} \\ & & & T' = \{\, (q,a,E^*(r)) \mid (\exists\ p : p \in q : (p,a,r) \in T) \,\} \\ & & & F' = \{\, f \mid f \in Q' \wedge f \cap F \neq \emptyset \,\} \\ & & \textbf{in} & \\ & & & (Q',V,T',\emptyset,\{E^*(S)\},F') \\ & & \textbf{end} & \end{array}$$

In the above version of $remove_\varepsilon$, each of the new states is one of the following:

- A new start state, which is the set of all states ε-transition reachable from the old start states S.
- A set of states, all of which are ε-transition reachable from a state in Q which has a non-ε in-transition.

Note that this transformation yields an automaton which has a single start state. □

Given a finite automaton construction $f \in RE \longrightarrow FA$, in some cases the dual of the construction, $R \circ f \circ R$, can be even more efficient than f. For this reason, we will also be needing the dual of function rem_ε.

Transformation 2.120 (Dual of function rem_ε): The dual of function rem_ε is defined as $(R \circ rem_\varepsilon \circ R)(Q,V,T,E,S,F) =$

$$\begin{array}{ll} \textbf{let} & Q' = \{(E^R)^*(F)\} \cup \{\, (E^R)^*(q) \mid \{q\} \times V \times Q \cap T \neq \emptyset \,\} \\ & T' = \{\, ((E^R)^*(q),a,Q) \mid (\exists\ p : p \in Q : (q,a,p) \in T) \,\} \\ & S' = \{\, s \mid s \in Q' \wedge s \cap S \neq \emptyset \,\} \\ \textbf{in} & \\ & (Q',V,T',\emptyset,S',\{(E^R)^*(F)\}) \\ \textbf{end} & \end{array}$$

□

Transformation 2.121 (Subset construction): The function *subset* transforms an ε-*free* FA into a DFA (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$)

$$\begin{array}{rcll} subset(Q,V,T,\emptyset,S,F) & = & \textbf{let} & T'(U,a) = \{(\cup\ q : q \in U : T(q,a))\} \\ & & & F' = \{\, U \mid U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset \,\} \\ & & \textbf{in} & \\ & & & (\mathcal{P}(Q),V,T',\emptyset,\{S\},F') \\ & & \textbf{end} & \end{array}$$

In addition to the property that (for all $M \in FA$) $\mathcal{L}_{FA}(subset(M)) = \mathcal{L}_{FA}(M)$, function *subset* satisfies

$$(\forall\ M : M \in FA \land \varepsilon\text{-}free(M) : Det(subset(M)) \land Complete(subset(M)))$$

and preserves *Complete*, *ε-free*, *Det*. Some authors call this the 'powerset' construction. □

In order to present a useful property of the subset construction, we need the following lemma.

Lemma 2.122 (Subset construction): Let

$$M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$$

and

$$M_1 = (Q_1, V, T_1, \emptyset, S_1, F_1) = subset(M_0)$$

be finite automata. For all $w \in V^*$ and $q \in Q_1$

$$w \in \overleftarrow{\mathcal{L}}_{M_1}(q) \equiv q = (\cup\ p : p \in Q_0 \land w \in \overleftarrow{\mathcal{L}}_{M_0}(p) : \{p\})$$

Proof:
We rewrite the left side of the above equivalence as follows:

$$\begin{array}{ll} & w \in \overleftarrow{\mathcal{L}}_{M_1}(q) \\ \equiv & \quad \{\ M_1 \text{ is a } DFA\ \} \\ & q = T_1^*(S_1, w) \end{array}$$

We rewrite the right side of the equality in the right side of the above equivalence as follows:

$$\begin{array}{ll} & (\cup\ p : p \in Q_0 \land w \in \overleftarrow{\mathcal{L}}_{M_0}(p) : \{p\}) \\ = & \quad \{\ \text{definition of } \overleftarrow{\mathcal{L}}_{M_0}\ \} \\ & (\cup\ p : p \in Q_0 \land p \in T_0^*(S_0, w) : \{p\}) \\ = & \quad \{\ \text{set calculus}\ \} \\ & T_0^*(S_0, w) \end{array}$$

We now need to prove $q = T_1^*(S_1, w) \equiv q = T_0^*(S_0, w)$. We now proceed by proving that

$$(\forall\ w : w \in V^* : T_1^*(S_1, w) = T_0^*(S_0, w))$$

We prove this by induction on $|w|$.

Basis: For the case $w = \varepsilon$ we have:

$$
\begin{array}{ll}
& T_1^*(S_1, \varepsilon) \\
= & \quad \{ \text{definition of } S_1 \} \\
& T_1^*(\{S_0\}, \varepsilon) \\
= & \quad \{ \text{definition of } T_1^* \in Q_1 \times V^* \longrightarrow Q_1 \} \\
& S_0 \\
= & \quad \{ \text{definition of } T_0^*\text{; no } \varepsilon\text{-transitions} \} \\
& T_0^*(S_0, \varepsilon)
\end{array}
$$

Induction hypothesis: Assume that $T_1^*(S_1, w) = T_0^*(S_0, w)$ holds for $w : |w| = k$.

Induction step: Consider $wa : |w| = k \wedge a \in V$

$$
\begin{array}{ll}
& T_1^*(S_1, wa) \\
= & \quad \{ \text{definition of } T_1^* \in Q_1 \times V^* \longrightarrow Q_1 \} \\
& T_1(T_1^*(S_1, w), a) \\
= & \quad \{ \text{induction hypothesis} \} \\
& T_1(T_0^*(S_0, w), a) \\
= & \quad \{ \text{definition of } T_1 \text{ using signature } T_1 \in Q_1 \times V \longrightarrow Q_1 \} \\
& (\cup\ q : q \in T_0^*(S_0, w) : T_0(q, a)) \\
= & \quad \{ \text{definition of } T_0 \} \\
& T_0(T_0^*(S_0, w), a) \\
= & \quad \{ \text{definition of } T_0^*\text{; no } \varepsilon\text{-transitions} \} \\
& T_0^*(S_0, wa)
\end{array}
$$

□

Property 2.123 (Subset construction): Let $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ and $M_1 = \mathit{subset}(M_0)$ be finite automata. By the subset construction, the state set of M_1 is $\mathcal{P}(Q_0)$. We have the following properties:

$$(\forall\ p : p \in \mathcal{P}(Q_0) : \overrightarrow{\mathcal{L}}_{M_1}(p) = (\cup\ q : q \in p : \overrightarrow{\mathcal{L}}_{M_0}(q)))$$

and (from Lemma 2.122):

$$(\forall\ p : p \in \mathcal{P}(Q_0) : \overleftarrow{\mathcal{L}}_{M_1}(p) = (\cap\ q : q \in p : \overleftarrow{\mathcal{L}}_{M_0}(q)))$$

□

We can also define the subset construction for Moore machines.

Transformation 2.124 (Subset construction for *MM*s): Function *subsetmm* transforms a *MM* into a *DMM* (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$ and $\lambda' \in \mathcal{P}(Q) \longrightarrow \mathcal{P}(W)$)

$$
\begin{array}{lcll}
subsetmm(Q,V,W,T,\lambda,S) & = & \textbf{let} & T'(U,a) = \{(\cup\ q : q \in U : T(q,a))\} \\
 & & & \lambda'(U) = \{\ \lambda(q) \mid q \in U\ \} \\
 & & \textbf{in} & \\
 & & & (\mathcal{P}(Q), V, \mathcal{P}(W), T', \lambda', \{S\}) \\
 & & \textbf{end} &
\end{array}
$$

□

2.6.2.1 Imperative implementations of some transformations

In this section, we present some imperative implementations of a few of the finite automata transformations. These implementations will be used in Chapter 6 to present some finite automata construction algorithms from the literature.

All of the algorithms presented make use of a common 'reachability algorithm' skeleton. The difference lies in the way the transition relation is computed in each step. In each of these algorithms, variables D and U (both with domain $\mathcal{P}(\mathcal{P}(Q))$) accumulate the set of 'done' and the set of 'undone' (yet to be considered) states in the automaton under construction, respectively.

Algorithm 2.125 (Composition $useful_s \circ rem_\varepsilon$):

$\{\ (Q,V,T,E,S,F) \in FA\ \}$
$S', T' := \{E^*(S)\}, \emptyset;$
$D, U := \emptyset, S';$
do $U \neq \emptyset \rightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $p, a : p \in u \wedge a \in V \wedge T(p,a) \neq \emptyset \rightarrow$
    $d := E^*(T(p,a));$
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]\ \ d \in D \rightarrow$ **skip**
    fi;
    $T' := T' \cup \{(u,a,d)\}$
  rof
od;
$F' := \{\ f \mid f \in D \wedge f \cap F \neq \emptyset\ \}$
$\{\ \mathcal{L}_{FA}(D,V,T',\emptyset,S',F') = \mathcal{L}_{FA}(Q,V,T,E,S,F)\ \}$

□

Algorithm 2.126 (Composition $useful_s \circ subset$):

$\{ (Q, V, T, \emptyset, S, F) \in FA \}$
$S', T' := \{S\}, \emptyset;$
$D, U := \emptyset, S';$
do $U \neq \emptyset \rightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \rightarrow$
    $d := (\cup\ q : q \in u : T(q, a));$
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]$ $d \in D \rightarrow$ **skip**
    fi;
    $T' := T' \cup \{(u, a, d)\}$
  rof
od;
$F' := \{ f \mid f \in D \wedge f \cap F \neq \emptyset \}$
$\{ \mathcal{L}_{FA}(D, V, T', \emptyset, S', F') = \mathcal{L}_{FA}(Q, V, T, E, S, F) \}$
$\{ (D, V, T', \emptyset, S', F') \in DFA \}$

□

Algorithm 2.127 (Composition $useful_s \circ subset \circ rem_\varepsilon$):

$\{ (Q, V, T, E, S, F) \in FA \}$
$S', T' := \{E^*(S)\}, \emptyset;$
$D, U := \emptyset, S';$
do $U \neq \emptyset \rightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \rightarrow$
    $d := (\cup\ q : q \in u : E^*(T(q, a)));$
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]$ $d \in D \rightarrow$ **skip**
    fi;
    $T' := T' \cup \{(u, a, d)\}$
  rof
od;
$F' := \{ f \mid f \in D \wedge f \cap F \neq \emptyset \}$
$\{ \mathcal{L}_{FA}(D, V, T', \emptyset, S', F') = \mathcal{L}_{FA}(Q, V, T, E, S, F) \}$
$\{ (D, V, T', \emptyset, S', F') \in DFA \}$

□

This algorithm is the same as the one given in [ASU86].

Part II

The taxonomies

Chapter 3

Constructing taxonomies

In this chapter, we provide a brief introduction to the construction of taxonomies. The McGraw-Hill Dictionary of scientific and technical terms provides the following (somewhat biology oriented) definition of a *taxonomy:*

> A study aimed at producing a hierarchical system of classification of organisms which best reflects the totality of similarities and differences. [Park89, p. 1892].

In a manner analogous to a biological taxonomy, we intend to classify algorithms according to their essential details. This classification, which is frequently presented in the form of a *(directed acyclic) taxonomy graph*, will allow us to compare algorithms and determine easily what they have in common and where they differ. In the following paragraphs, we detail the structure and construction of a taxonomy.

Given a particular problem area (for example, keyword pattern matching), the algorithms will be derived from a common starting point. The starting point is usually a naïve algorithm whose correctness is shown easily. Each of the algorithms appears as a vertex in the taxonomy graph, and the first algorithm is placed at the root of the taxonomy graph. The derivation proceeds by adding either *problem* or *algorithm* details. A problem detail is a correctness preserving restriction of the problem. Such a detail may enable us to make a change in the algorithm, usually to improve performance. The more specific problem may permit some transformation which is not possible in the algorithm solving the general problem. An algorithm detail, on the other hand, is a correctness-preserving transformation of the algorithm itself. These algorithm details may be added to restrict nondeterminacy, or to make a change of representation; either of these changes to an algorithm, gives a new algorithm meeting the same specification. In the taxonomies presented in Chapters 4 and 6, the particular details are explicitly defined and given mnemonic names. In the remaining taxonomy (Chapter 7), the details are only introduced implicitly.

Both types of details are chosen so as to improve the performance of an algorithm, or to arrive at one of the well-known algorithms appearing in the literature. The addition of a detail to algorithm A (arriving at algorithm B) is represented by adding an edge from A to B (the vertices representing algorithms A and B, respectively) to the taxonomy graph. The edge is labeled with the name of the detail. The use of correctness preserving transformations, and the correctness of the algorithm at the root of the graph, means that the correctness argument for any given algorithm is encoded in the root path leading to that particular algorithm.

It should be noted that, while the taxonomy is presented in a top-down manner, the taxonomy construction process proceeds initially in a bottom-up fashion. Each of the algorithms found in the literature is rewritten in a common notation and examined for any essential components or encoding tricks. The common encoding tricks, algorithm skeletons, or algorithm strategies can be made into details. The details making up the various algorithms can then be factored, so that some of them are presented together in the taxonomy graph — highlighting what some of the algorithms have in common.

A few notes on the taxonomy graph are in order. In some cases, it may have been possible to derive an algorithm through the application of some of the details in a different order[1]. The particular order chosen (for a given taxonomy) is very much a matter of taste. A number of different orders were tried, the resulting taxonomy graphs were compared, and the most elegant one was chosen. Frequently, the taxonomy graph is a tree. When the graph is not a tree, there may be two or more root paths leading to algorithm A. This means that there are at least two ways of deriving A from the naïve algorithm appearing at the root. It is also possible that not all of the algorithms solving a particular problem can be derived from a common starting point. In this case, we construct two or more separate taxonomy graphs, each with its own root.

This type of taxonomy development and program derivation has been used in the past. A notable one is Broy's sorting algorithm taxonomy [Broy83]. In Broy's taxonomy, algorithm and problem details are also added, starting with a naïve solution; the taxonomy arrives at all of the well-known sorting algorithms. A similar taxonomy (which predates Broy's) is by Darlington [Darl78]; this taxonomy also considers sorting algorithms. Our particular incarnation of the method of developing a taxonomy was developed in the dissertation of Jonkers [Jonk82], where it was used to give a taxonomy of garbage collection algorithms. Jonkers' method was then applied successfully to attribute evaluation algorithms by Marcelis in [Marc90]. A recent taxonomy (not using Jonkers' method) by Hume and Sunday [HS91] gives variations on the Boyer-Moore pattern matching algorithms; the taxonomy concentrates on many of the practical issues, and provides data on the running time of the variations and their respective precomputations.

Two primary aims of the taxonomies are clarity and correctness of presentation. We abandon low levels of abstraction, such as indexing within strings. Instead, we adopt a more abstract (but equivalent) presentation. Because of this, all of the abstract algorithms derived in this dissertation will be presented in a slightly extended version of Dijkstra's *guarded command language* [Dijk76]. The reasons for choosing the guarded command language are:

- Correctness arguments are more easily presented in the guarded commands than in programming languages such as Pascal or C.

[1] Only some of the details may be rearranged, since the correctness of some details may depend upon the earlier application of some detail.

- In order to present algorithms that closely represent their original imperative presentations (in journals or conference proceedings), we do not make use of other formalisms and paradigms, such as functional or relational programming.

We will frequently present algorithms without full annotations (invariants, preconditions, and postconditions) since most of the algorithm skeletons are relatively simple, and the annotations do not add much to the taxonomic classification. Annotations will be used when they help to introduce a problem or an algorithm detail.

This part is structured as follows:

- Chapter 4 presents a taxonomy of keyword pattern matching algorithms. The taxonomy concentrates on those algorithms that perform pattern matching of a finite set of keywords, and those algorithms that do not use precomputation of the input string.

- Chapter 5 gives a derivation of a new regular expression pattern matching algorithm. The existence (and derivation) of the algorithm answers an open question first posed by A.V. Aho in 1980. The algorithm, which is a generalization of the Boyer-Moore keyword pattern matching algorithm, displays good performance in practice.

- Chapter 6 presents a taxonomy of algorithms which construct a finite automaton from a regular expression. All of the well-known algorithms (including some very recently developed ones) are included.

- Chapter 7 presents a taxonomy of deterministic finite automata minimization algorithms. All of the well-known algorithms, and a pair of new ones, are included.

Chapter 4

Keyword pattern matching algorithms

This chapter presents a taxonomy of keyword pattern matching algorithms. We assume that the keyword set (the patterns) will remain relatively unchanged, whereas a number of different input strings (or perhaps a very long input string) may be used with the same pattern set. Because of this, we consider algorithms which may require some precomputation involving the keyword set[1]. The algorithms considered include the well-known Aho-Corasick, Knuth-Morris-Pratt, Commentz-Walter, and Boyer-Moore algorithms[2]. In addition, a number of variants (some of them not found in the literature) of these algorithms are presented.

The taxonomy is a much-revised version of one originally co-developed with Gerard Zwaan of the Eindhoven University of Technology. In the original version [WZ92], Zwaan was the primary author of Part II of that paper (which gave derivations of the precomputation algorithms), while I was the primary author of Part I (the taxonomy proper). A version of Section 4.4 appeared in [WZ95]. Gerard can be reached at `wsinswan@win.tue.nl`.

4.1 Introduction and related work

Keyword pattern matching is one of the most extensively explored problems in computing science. Loosely stated, the problem is to find the set of all occurrences from a set of patterns in an input string.

This chapter presents a taxonomy of keyword pattern matching algorithms. The main results are summarized in the taxonomy graph presented at the end of this section, and in the conclusions presented in Section 4.6. A version of the taxonomy graph is presented in each section, highlighting the part of the taxonomy considered in that section.

We systematically present a number of variants of four well-known algorithms in a common framework. Two of the algorithms to be presented require that the set of patterns is a single keyword, while the other two require that the set of patterns is a finite set of

[1] An alternative is to require that the subject string remain unchanged, with various different pattern sets being used. In this case, precomputation involving the subject string is preferred.

[2] We restrict ourselves to these 'classical' pattern matching algorithms and do not consider algorithms which are substantially different, such as those given in [WM94].

keywords. The algorithms are:

- The Knuth-Morris-Pratt (KMP) algorithm as presented in [KMP77]. This algorithm matches a single keyword against the input string. Originally, the algorithm was devised to find only the first match in the input string. We will consider a version that finds all occurrences within the input string.

- The Boyer-Moore (BM) algorithm as presented in [BM77]. This is also a single keyword matching algorithm. Several corrections and improvements to this algorithm have been published; good starting points for these are the bibliographic sections of [Aho90, CR94, Step94].

- The Aho-Corasick (AC) algorithm as presented in [AC75]. This algorithm can match a finite set of keywords in the input string.

- The Commentz-Walter (CW) algorithm as presented in [Com79a, Com79b]. This algorithm can also match a finite set of keywords in the input string. Few papers have been published on this algorithm, and its correctness, time complexity, and precomputation are ill-understood.

These four algorithms are also presented in the overview of [Aho90]. The first three algorithms are also covered quite extensively in a new book [CR94].

The recent taxonomy of pattern matching algorithms presented by Hume and Sunday (in [HS91]) gives variations on the Boyer-Moore algorithm; the taxonomy concentrates on many of the practical issues, and provides data on the running time of the variations, and their respective precomputation. In Chapter 9, we will consider a C++ class library (and many of the associated practical issues) implementing many of the algorithms presented in this taxonomy. In Chapter 13, we will consider the performance (in practice) of some of the algorithms implemented in the class library.

The taxonomy graph that we arrive at after deriving the algorithms is shown in Figure 4.1. Each vertex corresponds to an algorithm. If the vertex is labeled with a number, that number refers to an algorithm in this chapter. If it is labeled with a page number, that page number refers to the page where the algorithm is first mentioned. Each edge corresponds to the addition of either a problem or algorithm detail and is labeled with the name of that detail (a list of detail names follows). Each of the algorithms will either be called by their algorithm number, by their name as found in the literature (for the well-known algorithms), or by the parenthesized sequence of all labels along the path from the root to the algorithm's vertex. For example, the algorithm known as the optimized Aho-Corasick algorithm can also be called (P_+, E, AC, AC-OPT) (it is also Algorithm 4.53 in this dissertation). All of the well known algorithms appear near the bottom of the graph. Due to its labeling, the graph can be used as an alternative table of contents to this chapter.

Four algorithm details (P_+, S_+, P_-, and S_-) are actually composed of two separate algorithm details. For example, detail (P_+) is composed of details (P) and detail (+).

However the second detail must always follow either detail (P) or detail (S) and so we treat them as a single detail. The edges labeled MO and SL in Figure 4.1 represent generic algorithm details that still have to be instantiated. Possible instantiations are given by the two small trees at the bottom of Figure 4.1. The details and a short description of each of them are as follows:

P (Algorithm detail 4.4) Examine prefixes of a given string in any order.

P_+ Examine prefixes of a given string in order of increasing length.

P_- As in (P_+), but in order of decreasing length.

S (Algorithm detail 4.6) Examine suffixes of a given string in any order.

S_+ Examine suffixes of a given string in order of increasing length.

S_- As in (S_+), but in order of decreasing length.

RT (Algorithm detail 4.17) Usage of the reverse trie corresponding to the set of keywords to check whether a string which is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.

FT (Algorithm detail 4.28) Usage of the forward trie corresponding to the set of keywords to check whether a string which is a prefix of some keyword, followed by a symbol is again a prefix of some keyword.

E (Problem detail 4.33) Matches are registered by their endpoints.

AC (Algorithm detail 4.42) Maintain a variable, which is the longest suffix of the current prefix of the input string, which is still a prefix of a keyword.

AC-OPT (Algorithm detail 4.52) A single 'optimized' transition function is used to update the state variable in the Aho-Corasick algorithm.

LS (Algorithm detail 4.64) Use linear search to update the state variable in the Aho-Corasick algorithm.

AC-FAIL (Algorithm detail 4.71) Implement the linear search using the transition function of the extended forward trie and the failure function.

KMP-FAIL (Algorithm detail 4.75) Implement the linear search using the extended failure function.

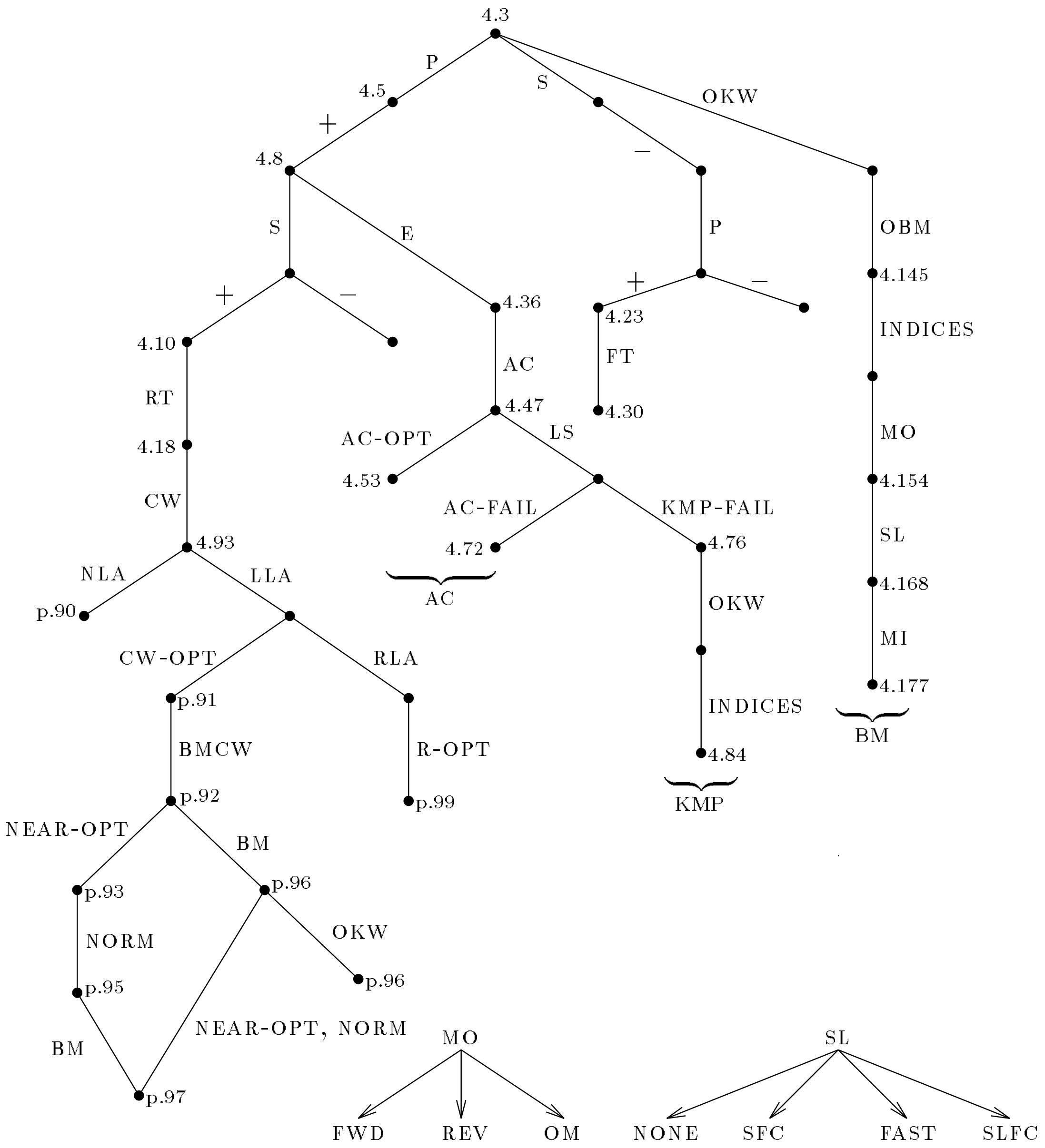


Figure 4.1: A taxonomy of pattern matching algorithms. An explanation of the graph and its labels is given in the text of this section. Algorithm 4.53 corresponds to the optimized Aho-Corasick algorithm [AC75, Section 6]. Algorithm 4.72 corresponds to the Aho-Corasick failure function algorithm [AC75, Section 2, Algorithm 1]. Algorithm 4.84 corresponds to the Knuth-Morris-Pratt algorithm [KMP77, Section 2, p. 326]. The algorithm of the vertex labeled p. 95 and with incoming edge labeled NORM corresponds to the Commentz-Walter algorithm [Com79a, Section II], [Com79b, Sections II.1 and II.2]. The algorithm of the vertex labeled p. 96 and with incoming edge labeled BM corresponds to the Boyer-Moore algorithm [BM77, Section 4]. Algorithm 4.177 corresponds to the Boyer-Moore algorithm as well [BM77, Sections 4 and 5].

OKW (Problem detail 4.77) The set of keywords contains one keyword.

INDICES (Algorithm detail 4.82) Represent substrings by indices into the complete strings, converting a string-based algorithm into an indexing-based algorithm.

CW (Algorithm detail 4.90) Consider any shift distance that does not lead to the missing of any matches. Such shift distances are called *safe*.

NLA (Algorithm detail 4.103) The left and right lookahead symbols are not taken into account when computing a safe shift distance. The computation of a shift distance is done by using two precomputed shift functions applied to the current longest partial match.

LLA (Algorithm detail 4.104) The left lookahead symbol is taken into account when computing a safe shift distance.

CW-OPT (Algorithm detail 4.108) Compute a shift distance using a single precomputed shift function applied to the current longest partial match and the left lookahead symbol.

BMCW (Algorithm detail 4.116) Compute a shift distance using a single precomputed shift function which is applied to the current longest partial match and the left lookahead symbol. The function yields shifts that are no greater than the function in detail (CW-OPT).

NEAR-OPT (Algorithm detail 4.121) Compute a shift distance using a single precomputed shift function applied to the current longest partial match and the left lookahead symbol. The function is derived from the one in detail (BMCW), and it yields shifts which are no greater.

NORM (Algorithm detail 4.127) Compute a shift distance as in (NLA) but additionally use a third shift function applied to the lookahead symbol. The shift distance obtained is that of the normal Commentz-Walter algorithm.

BM (Algorithm detail 4.135) Compute a shift distance using one shift function applied to the lookahead symbol, and another shift function applied to the current longest partial match. The shift distance obtained is that of the Boyer-Moore algorithm.

RLA (Algorithm detail 4.137) The right lookahead symbol is taken into account when computing a safe shift distance.

R-OPT (Algorithm detail 4.141) Compute a shift distance using precomputed shift functions (applied to the current longest partial match and the left lookahead symbol) and a shift function applied to the right lookahead symbol.

OBM (Algorithm detail 4.144) Introduce a particular program skeleton as a starting point for the derivation of the different Boyer-Moore variants.

MO (Algorithm detail 4.148) A match order is used to determine the order in which symbols of a potential match are compared against the keyword. This is only done for the one keyword case (OKW). Particular instances of match orders are:

- FWD (Algorithm detail 4.149) The forward match order is used to compare the (single) keyword against a potential match in a left to right direction.
- REV (Algorithm detail 4.150) The reverse match order is used to compare the (single) keyword against a potential match in a right to left direction. This is the original Boyer-Moore match order.
- OM (Algorithm detail 4.151) The symbols of the (single) keyword are compared in order of ascending probability of occurrence in the input string. In this way, mismatches will generally be discovered as early as possible.

SL (Algorithm detail 4.167) Before an attempt at matching a candidate string and the keyword, a 'skip loop' is used to skip portions of the input that cannot possibly lead to a match. Particular 'skips' are:

- NONE (Algorithm detail 4.169) No 'skip' loop is used.
- SFC (Algorithm detail 4.170 The 'skip loop' compares the first symbol of the match candidate and the keyword; as long as they do not match, the candidate string is shifted one symbol to the right.
- FAST (Algorithm detail 4.171) As with (SFC), but the last symbol of the candidate and the keyword are compared and possibly a larger shift distance (than with with SFC) is used.
- SLFC (Algorithm detail 4.172) As with (FAST), but a low frequency symbol of the keyword is first compared.

MI (Algorithm detail 4.176) The information gathered during an attempted match is used (along with the particular match order used during the attempted match) to determine a safe shift distance.

4.2 The problem and some naïve solutions

In this section, we start with a formal statement of the pattern matching problem that we consider. We present a number of simple algorithms of which the correctness is easily established. These simple algorithms are not particularly interesting or practical, but they provide convenient starting points for different branches of the taxonomy. The solid part of Figure 4.2 show the part of the taxonomy that we will consider in this section.

The problem is to find all occurrences of any of a set of keywords in an input string.

Definition 4.1 (Keyword pattern matching problem): Given an alphabet V, an input string $S \in V^*$, and a finite non-empty pattern set $P \subseteq V^*$, establish

$$PM: \ O = (\cup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\})$$

We will sometimes refer to S as the *subject* string. □

Example 4.2 (Pattern matching): Given input string $S = hishershey$ (a 'hershey' is a type of chocolate-bar available in North America) and keyword set $P = \{her, his, she\}$ over alphabet $V = \{e, h, i, r, s, y\}$, when PM holds we have

$$O = \{(\varepsilon, his, hershey), (hi, she, rshey), (his, her, shey), (hisher, she, y)\}$$

Notice that two matches are allowed to overlap, as in the case of leftmost *she* match and the *her* match. This example will be used throughout this chapter. □

Registering keyword matches (in the input string) as a set of strings is difficult and inefficient to implement in practice. A practical implementation would make use of indexing within the input string, and would encode matches using the indices, as is done in Chapter 9. In this chapter, we pursue this more abstract presentation (using strings and sets of strings) for clarity.

A trivial (but unrealistic) solution to the pattern matching problem is:

Algorithm 4.3 ():

$O := (\cup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\})$
$\{\ PM\ \}$

□

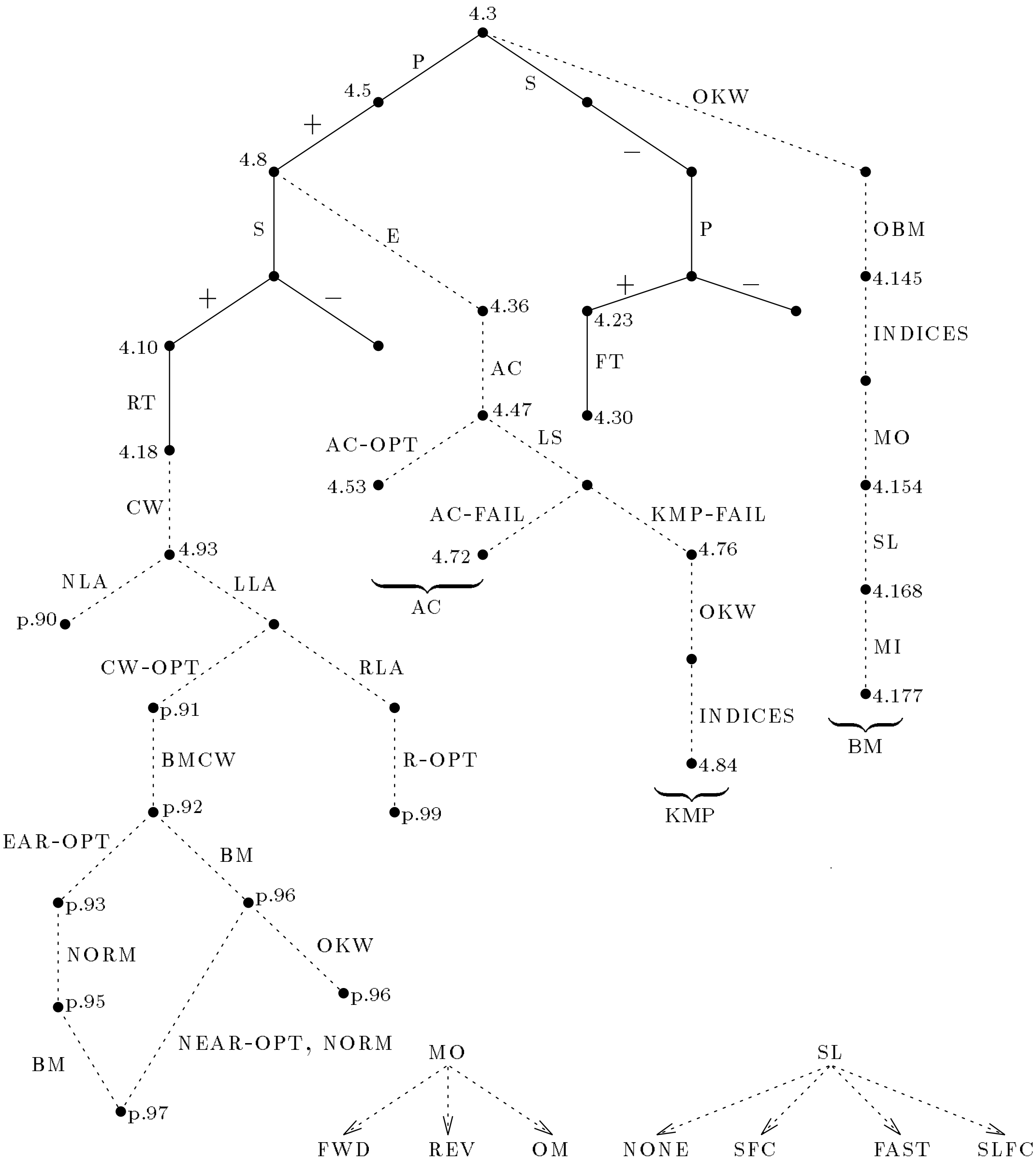


Figure 4.2: The algorithms considered in Section 4.2 are denoted by solid circles, connected by solid lines.

The sequence of details (between parentheses, after the algorithm number) describing this algorithm is the empty sequence. This algorithm appears at the root of the taxonomy graph.

There are two basic directions in which to proceed while developing naïve algorithms to solve this problem. Informally, a substring of S (such as v in the $\cup$ quantification of the above algorithm) can be considered a "suffix of a prefix of S" or a "prefix of a suffix of S". These two possibilities are considered separately below.

Formally, we can consider "suffixes of prefixes of S" as follows:

$$
\begin{array}{ll}
 & (\cup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\}) \\
= & \quad \{\text{introduce } u : u = lv\} \\
 & (\cup\ l, v, r, u : ur = S \wedge lv = u : \{l\} \times (\{v\} \cap P) \times \{r\}) \\
= & \quad \{l, v \text{ only occur in the latter range conjunct, so restrict their scope}\} \\
 & (\cup\ u, r : ur = S : (\cup\ l, v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\}))
\end{array}
$$

A simple nondeterministic algorithm is obtained by introducing the following algorithm detail:

Algorithm detail 4.4 (P): Examine prefixes of a given string in any order. □

The resulting algorithm is:

Algorithm 4.5 (P):

$O := \emptyset;$
for $u, r : ur = S \rightarrow$
  $O := O \cup (\cup\ l, v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})$
rof$\{ PM \}$

□

Again starting from Algorithm 4.3() we can also consider "prefixes of suffixes of S" as follows:

$$
\begin{array}{ll}
 & (\cup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\}) \\
= & \quad \{\text{introduce } w : w = vr\} \\
 & (\cup\ l, v, r, w : lw = S \wedge vr = w : \{l\} \times (\{v\} \cap P) \times \{r\}) \\
= & \quad \{v, r \text{ only occur in the latter range conjunct, so restrict their scope}\} \\
 & (\cup\ l, w : lw = S : (\cup\ v, r : vr = w : \{l\} \times (\{v\} \cap P) \times \{r\}))
\end{array}
$$

As with Algorithm detail (P), we introduce the following detail.

Algorithm detail 4.6 (S): Examine suffixes of a given string in any order. □

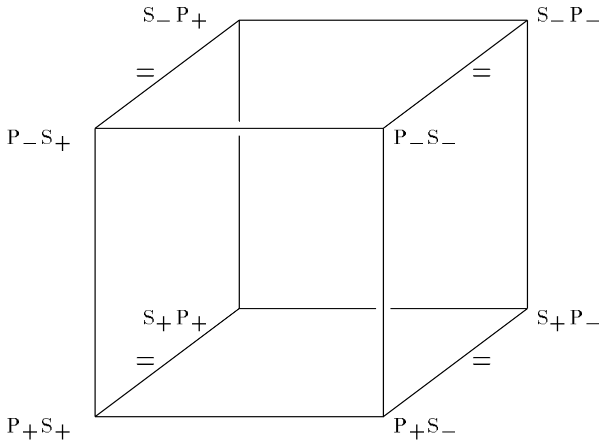


Figure 4.3: The 3-cube of naïve pattern matching algorithms.

Using this algorithm detail yields the simple nondeterministic algorithm (S) which is analogous to Algorithm 4.5(P). Hence, it is not presented here.

The update of O (with another quantification) in the inner repetitions of algorithms (P) and (S) can be computed with another nondeterministic repetition. In the case of (P), the inner repetition would consider suffixes of u to give algorithm (P, S); similarly, in (S) the inner repetition would consider prefixes of u to give algorithm (S, P).

Each of (P, S) and (S, P) consists of two nested nondeterministic repetitions. In each case, the repetition can be made deterministic by considering prefixes (or suffixes as the case is) in increasing (called detail (+)) or decreasing (detail (−)) order of length. For each of (P, S) and (S, P) this gives two binary choices. Along with the binary choice between (P, S) and (S, P) this gives eight possible naïve algorithms, arranged in a 3-cube representing the three binary choices; the cube is depicted in Figure 4.3 with vertices representing the eight possible algorithms for the two nested repetitions. The edges marked '=' join algorithms which are symmetrical; for example, the order in which (P_+, S_-) considers input string S and keyword set P is mirrored (with respect to string reversal of S and P) by the order in which (S_+, P_-) considers S and P. Because of this symmetry, we present only four algorithms in this section: (P_+, S_+), (P_+, S_-), (S_-, P_-), and (S_-, P_+). These algorithms were chosen because their outer repetitions examine S in left to right order.

Forward reference 4.7: In Section 4.2.1, Algorithm 4.5(P) will be refined further and in Section 4.2.2, Algorithm (S) will be refined. In Section 4.3, Algorithm (P_+) will be developed into the Aho-Corasick and Knuth-Morris-Pratt algorithms, while in Sections 4.4 and 4.5, Algorithm (P_+, S_+) will be developed into the Commentz-Walter and Boyer-Moore algorithms. □

4.2.1 The (P$_+$) algorithms

The (P) algorithm presented in the previous section can be made deterministic by considering prefixes of S in order of increasing length (Algorithm detail (P$_+$)). The outer union quantification in the required value of O can be computed with a deterministic repetition:

Algorithm 4.8 (P$_+$):

$u, r := \varepsilon, S;\ O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $O := O \cup (\cup\ l, v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})$
od$\{\ PM\ \}$

□

Forward reference 4.9: This algorithm will be used in Section 4.3 as a starting point for the Aho-Corasick and Knuth-Morris-Pratt algorithms. □

The inner union quantification in the required value of O can be computed with a nondeterministic repetition. This algorithm is called (P$_+$, S) but will not be given here.

4.2.1.1 The (P$_+$, S$_+$) algorithm and its improvement

Starting with algorithm (P$_+$, S) we make its inner repetition deterministic by considering suffixes of u in order of increasing length:

Algorithm 4.10 (P$_+$, S$_+$):

$u, r := \varepsilon, S;\ O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $l, v := u, \varepsilon;\ O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\};$
  do $l \neq \varepsilon \rightarrow$
    $l, v := l{\downharpoonright}1, (l{\upharpoonright}1)v;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od
od$\{\ PM\ \}$

□

Remark 4.11: This algorithm has $\mathcal{O}(|S|^2)$ running time, assuming that intersection with P is a $\mathcal{O}(1)$ operation. □

We can make an improvement to the above algorithm by noting that $v \in P \Rightarrow \mathbf{suff}(v) \subseteq \mathbf{suff}(P)$, and therefore $\mathbf{suff}(v) \not\subseteq \mathbf{suff}(P) \Rightarrow v \notin P$. Intuitively, when a string v is not in $\mathbf{suff}(P)$, there is no string u such that $uv \in P$. The property that we need can be stated more precisely.

Property 4.12 (Suffixes of P): Note that

$$(\forall\ w, a : w \notin \mathbf{suff}(P) : aw \notin \mathbf{suff}(P)).$$

□

In other words, in the inner repetition when $(l{\upharpoonright}1)v \notin \mathbf{suff}(P)$ we need not consider any longer suffixes of u. This means that the inner repetition guard $(l \neq \varepsilon)$ can be strengthened to

$$l \neq \varepsilon \ \mathbf{cand}\ (l{\upharpoonright}1)v \in \mathbf{suff}(P).$$

The direct evaluation of $(l{\upharpoonright}1)v \in \mathbf{suff}(P)$ is expensive. Therefore, it is done using a function (corresponding to P) called a *reverse trie* [Fred60], defined as follows:

Definition 4.13 (Reverse trie corresponding to P): The reverse trie corresponding to P is defined as function $\tau_{P,r} \in \mathbf{suff}(P) \times V \longrightarrow \mathbf{suff}(P) \cup \{\perp\}$ defined by

$$\tau_{P,r}(w,a) = \begin{cases} aw & \text{if } aw \in \mathbf{suff}(P) \\ \perp & \text{if } aw \notin \mathbf{suff}(P) \end{cases}$$

□

Convention 4.14 (Reverse trie): Since we usually refer to the trie corresponding to P we will write τ_r instead of $\tau_{P,r}$. □

Example 4.15 (Reverse trie): The reverse trie corresponding to our example keyword set $P = \{her, his, she\}$ is show in Figure 4.4. The vertices in the directed graph represent elements of $\mathbf{suff}(P)$, while the edges represent the mapping of an element of $\mathbf{suff}(P)$ and an element of V to $\mathbf{suff}(P)$. Note that cases where the reverse trie takes value $\perp$ are not shown. □

Remark 4.16: Since $|\mathbf{suff}(P)|$ is finite, function τ_r can be viewed as a kind of transition function, with $\perp$ meaning 'undefined'. □

Algorithm detail 4.17 (RT): Given the reverse trie, the guard conjunct $(l{\upharpoonright}1)v \in \mathbf{suff}(P)$ becomes $\tau_r(v, l{\upharpoonright}1) \neq \perp$. □

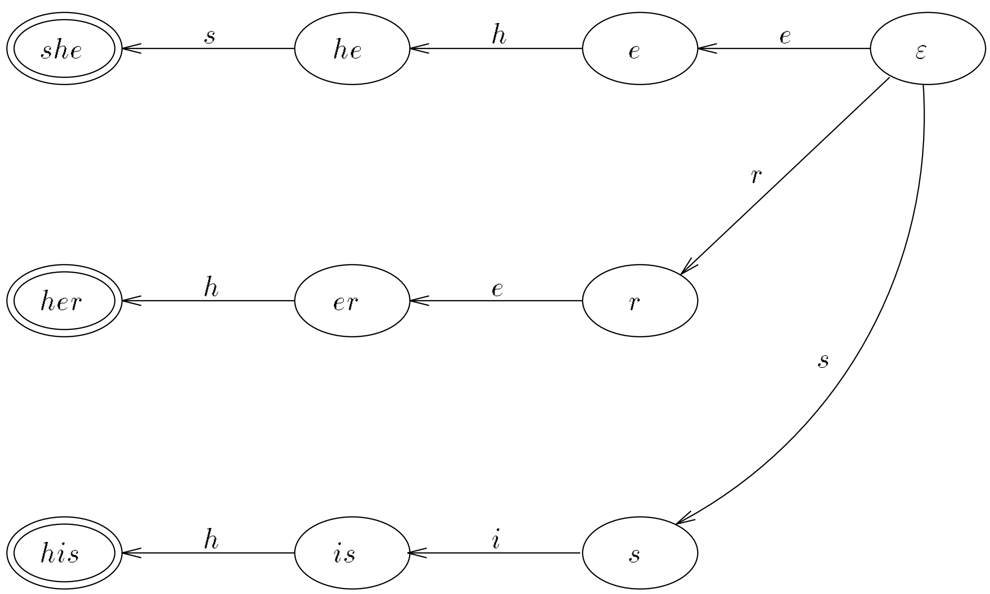


Figure 4.4: Example of a reverse trie.

Algorithm 4.18 (P_+, S_+, RT):

$u, r := \varepsilon, S$; $O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\}$;
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1$;
  $l, v := u, \varepsilon$; $O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\}$;
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l{\upharpoonright}1) \neq \perp \rightarrow$
    $l, v := l{\downharpoonright}1, (l{\upharpoonright}1)v$;
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od
od$\{ PM \}$

□

Forward reference 4.19: Observe that $u = lv \wedge v \in \textbf{suff}(P)$ is an invariant of the inner repetition, initially established by the assignment $l, v := u, \varepsilon$. This invariant will be used in Section 4.4 to arrive at the Commentz-Walter algorithms. □

Remark 4.20: This algorithm has $\mathcal{O}(|S| \cdot (\textbf{MAX}\ p : p \in P : |p|))$ running time. The precomputation of τ_r is similar to the precomputation of the forward trie τ_f (see Definition 4.26) which is discussed in [WZ92, Part II, Section 6]. □

In practice, a reverse trie can be implemented as a table with $|\textbf{suff}(P)| \cdot |V|$ entries, with elements of $\textbf{suff}(P)$ and V being encoded as integers and used to index the table. Such an implementation is used in Chapter 9.

4.2.1.2 The (P$_+$, S$_-$) algorithm

In the previous section, we modified the inner repetition of algorithm (P$_+$, S) to consider suffixes of u in order of increasing length. In this section, we will make use of an inner repetition which considers them in order of decreasing length. This gives us the following algorithm:

Algorithm 4.21 (P$_+$, S$_-$):

$u, r := \varepsilon, S;\ O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1;$
  $l, v := \varepsilon, u;$
  do $v \neq \varepsilon \rightarrow$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\};$
    $l, v := l(v\upharpoonright 1), v\downharpoonright 1$
  od;
  $O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\}$
od$\{\ PM\ \}$

□

Remark 4.22: This algorithm has $\mathcal{O}(|S|^2)$ running time and it appears difficult to improve its performance. □

4.2.2 The (S$_-$) algorithms

Algorithm (S) can be made deterministic by considering suffixes of S in order of decreasing length. This results in the deterministic algorithm (S$_-$) which will not be given here. Furthermore, the assignment to O in the repetition can be written as a nondeterministic repetition to give the algorithm (S$_-$, P) which will not be given here.

4.2.2.1 The (S$_-$, P$_+$) algorithms

Starting with algorithm (S$_-$, P) we make the inner repetition deterministic by considering prefixes of each suffix of the input string in order of increasing length. The algorithm is:

Algorithm 4.23 (S$_-$, P$_+$):

$l, w := \varepsilon, S; O := \emptyset;$
do $w \neq \varepsilon \rightarrow$
  $v, r := \varepsilon, w;\ O := O \cup \{l\} \times (\{\varepsilon\} \cap P) \times \{w\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r\upharpoonright 1), r\downharpoonright 1;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$

od;
$l, w := l(w\upharpoonright 1), w\downharpoonright 1$
od;
$O := O \cup \{S\} \times (\{\varepsilon\} \cap P) \times \{\varepsilon\}$
$\{ \ PM \ \}$

□

Remark 4.24: This algorithm has $\mathcal{O}(|S|^2)$ running time, like Algorithm 4.10(P+, S+). □

In a manner similar to the introduction of the reverse trie (Definition 4.13 and Algorithm 4.18(P+, S+, RT)), we can strengthen the inner repetition guard. The following definitions are reflections (under string reversal) of those presented starting on page 51.

Property 4.25 (Prefixes of P): Note that

$$(\forall\ u, a : u \notin \mathbf{pref}(P) : ua \notin \mathbf{pref}(P))$$

Given this property, we can strengthen the guard of the inner repetition to

$$r \neq \varepsilon \ \mathbf{cand} \ v(r\upharpoonright 1) \in \mathbf{pref}(P)$$

This property is the reflection of Property 4.12. □

Efficient computation of the strengthened guard ($r \neq \varepsilon$ **cand** $v(r\upharpoonright 1) \in \mathbf{pref}(P)$) can be done by using the forward trie corresponding to P.

Definition 4.26 (Forward trie corresponding to P): The forward trie function corresponding to P is $\tau_f \in \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P) \cup \{\perp\}$, defined by

$$\tau_f(u, a) = \begin{cases} ua & \text{if } ua \in \mathbf{pref}(P) \\ \perp & \text{if } ua \notin \mathbf{pref}(P) \end{cases}$$

□

Example 4.27 (Forward trie): The forward trie corresponding to our example keyword set $P = \{her, his, she\}$ is shown in Figure 4.5. In a manner analogous to the reverse trie example, the vertices in the directed graph represent elements of $\mathbf{pref}(P)$, while the edges represent the mapping of an element of $\mathbf{pref}(P)$ and an element of V to $\mathbf{pref}(P)$ (and cases where the reverse trie takes value $\perp$ are not shown). □

Algorithm detail 4.28 (FT): Given the forward trie, the guard conjunct $v(r\upharpoonright 1) \in \mathbf{pref}(P)$ now becomes $\tau_f(v, r\upharpoonright 1) \neq \perp$. □

Remark 4.29: The forward trie detail (FT) is defined and used symmetrically to the reverse trie detail (RT) (see Algorithm detail 4.17). □

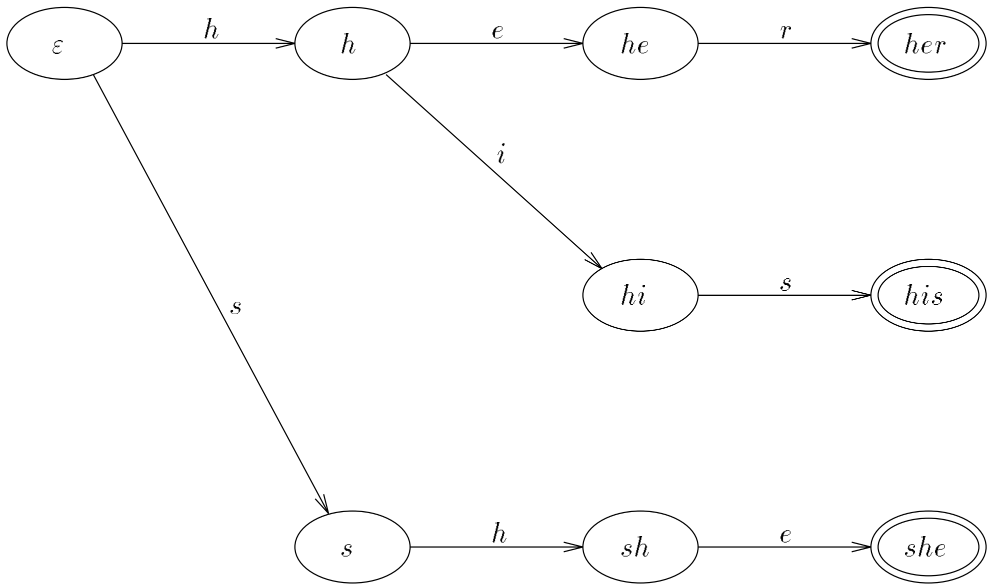


Figure 4.5: Example of a forward trie.

Introducing algorithm detail (FT) yields

Algorithm 4.30 (S$_-$, P$_+$, FT):

$l, w := \varepsilon, S;\ O := \emptyset;$
do $w \neq \varepsilon \rightarrow$
  $v, r := \varepsilon, w;\ O := O \cup \{l\} \times (\{\varepsilon\} \cap P) \times \{w\};$
  do $r \neq \varepsilon$ **cand** $\tau_f(v, r\upharpoonright 1) \neq \perp \rightarrow$
    $v, r := v(r\upharpoonright 1), r\downharpoonright 1;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od;
  $l, w := l(w\upharpoonright 1), w\downharpoonright 1$
od;
$O := O \cup \{S\} \times (\{\varepsilon\} \cap P) \times \{\varepsilon\}$
$\{\ PM\ \}$

□

Remark 4.31: As in Forward reference 4.19, observe that $w = vr \wedge v \in \mathbf{pref}(P)$ is an invariant of the inner repetition. It is initially established by the assignment $v, r := \varepsilon, w$. □

Remark 4.32: This algorithm has $\mathcal{O}(|S| \cdot (\mathbf{MAX}\ p : p \in P : |p|))$ running time, like Algorithm 4.18(P$_+$, S$_+$, RT). □

4.2.2.2 The (S_-, P_-) algorithm

The inner repetition of algorithm (S_-, P) can also be made deterministic by considering prefixes of w in order of decreasing length. This yields algorithm (S_-, P_-) which is not given here. Its running time is $\mathcal{O}(|S|^2)$.

4.3 The Aho-Corasick algorithms

In this section, starting with the naïve Algorithm 4.8(P_+) from Section 4.2, we derive the Aho-Corasick [AC75] and Knuth-Morris-Pratt [KMP77] algorithms and their variants. The Knuth-Morris-Pratt (KMP) algorithm is a well known single keyword matching algorithm operating in time linear in the length of the subject string. The article [KMP77] gives an interesting account of the history of development of the algorithm. Aho and Corasick (AC) combined its essential idea with concepts from automata theory to obtain two multiple keyword matching algorithms, also operating in linear time.

The common aspect of all these algorithms is the construction of a kind of Moore[3] machine (see Definition 2.79) during a preprocessing phase. Using this Moore machine, the subject string can be scanned in linear time. The variants of the AC and the KMP algorithms all make use of the same Moore machine, but each of them uses a different method to compute the next transition. In the failure function AC algorithm, computation of the transition function is implemented using the forward trie (corresponding to P), while in the KMP algorithm it is realized by indexing in the pattern. These differences lead to tradeoffs between the time to process the input string, the time to precompute the required functions, and the space to store the required functions. The optimized AC algorithm (Algorithm 4.53) can process each symbol of the input string in constant time, but requires $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ time and space for precomputed functions. On the other hand, the failure function AC algorithm (Algorithm 4.72) can require $\mathcal{O}(|\mathbf{pref}(P)|)$ time to process a given input symbol, but it only requires $\mathcal{O}(|\mathbf{pref}(P)|)$ time and space for precomputed functions.

Although these algorithms are frequently presented in an automata-theoretic way, in this dissertation the automata aspects of these algorithms will not be stressed. Instead, a more general (algorithmic) presentation will be used. Automata theoretic approaches to deriving these algorithms are presented in Sections 4.3.3 and 4.3.7. Both of these sections can be omitted without loss of continuity.

Figure 4.6 shows the part of the taxonomy that we will consider in this section. The algorithms to be presented are denoted by solid circles, connected by solid lines.

The triple format of set O used so far has been redundant. This redundancy can be removed by registering matches in S by their end-points only; that is, the first component of the triple will be dropped. This modification is known as problem detail (E).

[3]The inventor of Moore machines, E.F. Moore, is not the co-inventor (with Boyer), J Strother Moore, of the Boyer-Moore pattern matching algorithms. E.F. Moore performed much of the original research into the minimization of *DFA*s — see Chapter 7.

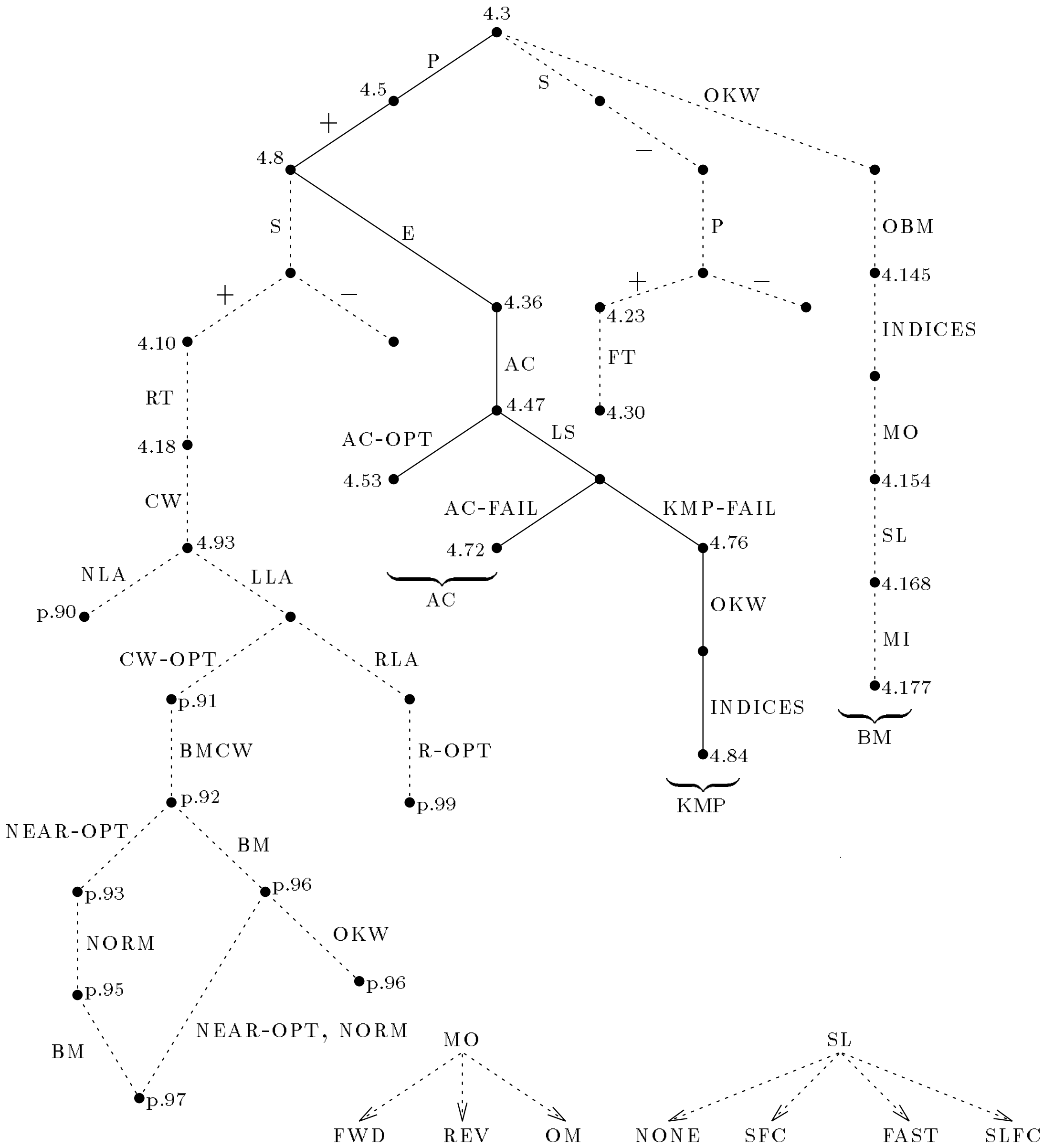


Figure 4.6: The algorithms considered in Section 4.3 are denoted by solid circles, connected by solid lines.

Problem detail 4.33 (E): Matches are registered by their end-points. □

Dropping the first component of the triples will allow us to make some efficiency improvements to the algorithms.

The desired value of O (with the first component dropped) in postcondition PM can be rewritten as follows:

$$\begin{array}{ll} & \bar{\pi}_1(\cup\, u,r : ur = S : (\cup\, l,v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})) \\ = & \quad \{\ \bar{\pi} \text{ distributes over } \cup\ \} \\ & (\cup\, u,r : ur = S : (\cup\, l,v : lv = u : \bar{\pi}_1(\{l\} \times (\{v\} \cap P) \times \{r\}))) \\ = & \quad \{\ \text{definition of } \bar{\pi}\ \} \\ & (\cup\, u,r : ur = S : (\cup\, l,v : lv = u : (\{v\} \cap P) \times \{r\})) \\ = & \quad \{\ \times \text{ distributes over } \cup\ \} \\ & (\cup\, u,r : ur = S : (\cup\, l,v : lv = u : \{v\} \cap P) \times \{r\}) \\ = & \quad \{\ \cap \text{ distributes over } \cup\ \} \\ & (\cup\, u,r : ur = S : ((\cup\, l,v : lv = u : \{v\}) \cap P) \times \{r\}) \\ = & \quad \{\ \text{definition of } \mathbf{suff}\ \} \\ & (\cup\, u,r : ur = S : (\mathbf{suff}(u) \cap P) \times \{r\}) \end{array}$$

Definition 4.34 (Refined postcondition): The derivation above yields a new postcondition

$$PM_e :\ O_e = (\cup\, u,r : ur = S : (\mathbf{suff}(u) \cap P) \times \{r\})$$

□

Example 4.35 (End-point pattern matching): Assuming the input string and keyword set from Example 4.2, when PM_e holds we have

$$O_e = \{(his, hershey), (she, rshey), (her, shey), (she, y)\}$$

□

This postcondition is established by a modified version of Algorithm 4.8(P$_+$) (we could have used Algorithm 4.5(P), however, we choose to use a deterministic algorithm):

Algorithm 4.36 (P$_+$, E):

$u, r := \varepsilon, S;\ O_e := (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $O_e := O_e \cup (\mathbf{suff}(u) \cap P) \times \{r\}$
od$\{\ PM_e\ \}$

□

In the following sections, algorithm details unique to the AC and KMP algorithms will be introduced.

4.3.1 Algorithm detail (AC)

In Algorithm 4.36, we see that new matches are registered whenever the condition $\mathbf{suff}(u) \cap P \neq \emptyset$ holds, i.e. when one or more patterns occur as suffixes of u, the part of the subject string read thus far. The essential idea of both the AC and the KMP algorithms is the use of an easily updateable *state* variable that gives information about (partial) matches in $\mathbf{suff}(u)$, and from which the set $\mathbf{suff}(u) \cap P$ can easily be computed.

In order to facilitate the update of O_e in Algorithm 4.36(P$_+$, E) we introduce a new variable U and attempt to maintain invariant $U = \mathbf{suff}(u) \cap P$. When u is updated to $u(r\lceil 1)$, we require an update of U to maintain the invariant. We begin deriving this update as follows:

$$\begin{array}{ll}
 & \mathbf{suff}(u(r\lceil 1)) \cap P \\
= & \quad \{\text{ Property 2.54, } \cap \text{ distributes over } \cup \,\} \\
 & (\mathbf{suff}(u)(r\lceil 1) \cap P) \cup (\{\varepsilon\} \cap P) \\
= & \quad \{\text{ Property 2.56 }\} \\
 & ((\mathbf{suff}(u) \cap \mathbf{pref}(P))(r\lceil 1) \cap P) \cup (\{\varepsilon\} \cap P)
\end{array}$$

From the above derivation, it seems difficult to derive an easily computed update for U. The update of U could more easily be accomplished, given the set $\mathbf{suff}(u) \cap \mathbf{pref}(P)$ rather than the old value of U (which is $\mathbf{suff}(u) \cap P$). The set $\mathbf{suff}(u) \cap \mathbf{pref}(P)$ can be viewed as a generalization of the set $\mathbf{suff}(u) \cap P$.

In order to obtain an algorithm that is more easily implemented in practice, we try to maintain invariant

$$U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$$

which is initially established by assignment $u, U := \varepsilon, \{\varepsilon\}$ since $P \neq \emptyset$. Assuming $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ we derive the following update of U:

$$\begin{array}{ll}
 & \mathbf{suff}(u(r\lceil 1)) \cap \mathbf{pref}(P) \\
= & \quad \{\text{ preceding derivation with } \mathbf{pref}(P) \text{ instead of } P \,\} \\
 & ((\mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{pref}(P)))(r\lceil 1) \cap \mathbf{pref}(P)) \cup (\{\varepsilon\} \cap \mathbf{pref}(P)) \\
= & \quad \{\text{ Property 2.51 — idempotence of } \mathbf{pref} \,\} \\
 & ((\mathbf{suff}(u) \cap \mathbf{pref}(P))(r\lceil 1) \cap \mathbf{pref}(P)) \cup (\{\varepsilon\} \cap \mathbf{pref}(P)) \\
= & \quad \{\, U = \mathbf{suff}(u) \cap \mathbf{pref}(P), \text{ Property 2.55 — } \varepsilon \in \mathbf{pref}(P) \,\} \\
 & (U(r\lceil 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}
\end{array}$$

The new invariant relating U and u yields another interesting property:

Property 4.37 (Set U): From $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ and $P \subseteq \mathbf{pref}(P)$ it follows that $U \cap P = \mathbf{suff}(u) \cap \mathbf{pref}(P) \cap P = \mathbf{suff}(u) \cap P$. We can use the expression $U \cap P$ in the update of variable O_e. □

This all leads to the following modification of Algorithm 4.36(P$_+$, E):

Algorithm 4.38:

$u, r := \varepsilon, S;\ U := \{\varepsilon\};\ O_e := (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $U := (U(r\upharpoonright 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\};$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1;$
  $O_e := O_e \cup (U \cap P) \times \{r\}$
od$\{\ PM_e\ \}$

□

It should be noted that variable u is now superfluous. It will, however, be kept in all subsequent algorithms to help formulate invariants.

Returning to our introduction of variable U with invariant $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ in Algorithm 4.38, we see no easy way to implement this algorithm in practice (given that U is a language) — it appears difficult to implement the update statement $U := (U(r\upharpoonright 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$. Therefore, we try to exploit the internal structure of U to obtain an easier update statement. We proceed by using the following property.

Property 4.39 (Set $\mathbf{suff}(u) \cap \mathbf{pref}(P)$): For each $u \in V^*$ the set $\mathbf{suff}(u) \cap \mathbf{pref}(P)$ is nonempty, finite, and linearly ordered with respect to the suffix ordering $\leq_s$. The set therefore has a maximal element $(\mathbf{MAX}_{\leq_s}\ w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)$. (The set is also (partially) ordered with respect to the prefix ordering, $\leq_p$, but that does not prove to be particularly useful.) This maximal element also characterizes the set, since

$$\mathbf{suff}(u) \cap \mathbf{pref}(P) = \mathbf{suff}((\mathbf{MAX}_{\leq_s}\ w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)) \cap \mathbf{pref}(P)$$

□

Example 4.40 (Characterizing $\mathbf{suff}(u) \cap \mathbf{pref}(P)$): If we take $u = hish$ (and therefore $r = ershey$), we have

$$\mathbf{suff}(hish) \cap \mathbf{pref}(P) = \{\varepsilon, h, sh\}$$

with maximal (under $\leq_s$) element sh. □

The implication of the above property is that we can encode any possible value of U by an element of $\mathbf{pref}(P)$. Such an encoding is of great practical value since a state which is a string is much more easily implemented than a state which is a language. Note that the set of possible values that U can take is $\{\, \mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^* \,\}$. We now define the encoding function.

Definition 4.41 (Encoding function enc): Bijective encoding function

$$enc \in \{\, \mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^* \,\} \longrightarrow \mathbf{pref}(P)$$

is

$$enc(U) = (\mathbf{MAX}_{\leq_s}\, w : w \in U : w)$$

with inverse $enc^{-1}(w) = \mathbf{suff}(w) \cap \mathbf{pref}(P)$. (Note that there are different uses of w in the lines above.) □

The fact that function enc is bijective means that $\{\, \mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^* \,\}$ and $\mathbf{pref}(P)$ are isomorphic (and therefore have the same cardinality).

We replace variable U in the algorithm by variable q and maintain invariant $q = enc(U)$, equivalently

$$q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)$$

which is initially established by $q := \varepsilon$ (since $enc(\{\varepsilon\}) = \varepsilon$). The introduction of variable q constitutes the essential idea of the Aho-Corasick family of algorithms. We call this algorithm detail (AC).

Algorithm detail 4.42 (AC): A variable q is introduced into Algorithm 4.36(P$_+$, E) such that

$$q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)$$

□

Property 4.43 (Variable q): It follows from the update of O_e using U, and function enc that we can rewrite the update of O_e in terms of q, since $U \cap P = enc^{-1}(q) \cap P = \mathbf{suff}(q) \cap \mathbf{pref}(P) \cap P = \mathbf{suff}(q) \cap P$. □

In order to make the update of program variable O_e more concise, we introduce the following auxiliary function:

Definition 4.44 (Function $Output$): Function $Output \in \mathbf{pref}(P) \longrightarrow \mathcal{P}(P)$ is defined by

$$Output(w) = \mathbf{suff}(w) \cap P$$

□

Example 4.45 (Function $Output$):

w	ε	h	s	he	hi	sh	her	his	she
$Output(w)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{her\}$	$\{his\}$	$\{she\}$

□

The update of O_e can be done by assignment $O_e := O_e \cup Output(q) \times \{r\}$.

Remark 4.46: The precomputation of function $Output$ can be done in $\mathcal{O}(|\mathbf{pref}(P)|)$ time; an algorithm doing this is presented in [WZ92, Part II, Section 6]. □

We now require an update of variable q, in order to maintain the invariant. The update of variable U was

$$U := (U(r{\upharpoonright}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$$

Given bijection enc and the invariant relating U and q, the update of q is:

$$q := enc((enc^{-1}(q)(r{\upharpoonright}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\})$$

We can manipulate the right side into a more readable form as follows:

$$\begin{array}{ll}
 & enc((enc^{-1}(q)(r{\upharpoonright}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}) \\
= & \quad \{ \text{definition of } enc^{-1} \} \\
 & enc(((\mathbf{suff}(q) \cap \mathbf{pref}(P))(r{\upharpoonright}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}) \\
= & \quad \{ \text{Property 2.56} \} \\
 & enc((\mathbf{suff}(q)(r{\upharpoonright}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}) \\
= & \quad \{ \cap \text{ distributes over } \cup\text{; Property 2.55 — } \varepsilon \in \mathbf{pref}(P) \} \\
 & enc((\mathbf{suff}(q)(r{\upharpoonright}1) \cup \{\varepsilon\}) \cap \mathbf{pref}(P)) \\
= & \quad \{ \text{Property 2.54} \} \\
 & enc(\mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P)) \\
= & \quad \{ \text{definition of } enc \} \\
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P) : w)
\end{array}$$

We now have obtained algorithm

Algorithm 4.47 (P$_+$, E, AC):

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $q := (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P) : w);$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $O_e := O_e \cup Output(q) \times \{r\}$
od$\{ PM_e \}$

□

Forward reference 4.48: Sections 4.3.2, 4.3.4, 4.3.5, and 4.3.6 are concerned with alternative ways of implementing assignment

$$q := (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P) : w)$$

□

4.3.2 Method (AC-OPT)

In this section, we aim for an implementation of the assignment given in Forward reference 4.48 by means of a simple statement of the form $q := \gamma_f(q, r\lceil 1)$, where γ_f is a suitable transition function, defined as follows:

Definition 4.49 (Function γ_f): Function $\gamma_f \in \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P)$ is defined as

$$\gamma_f(q, a) = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(qa) \cap \mathbf{pref}(P) : w)$$

□

Remark 4.50: Subscript f in γ_f is used to indicate that γ_f corresponds to the forward trie transition function τ_f. That is, $\tau_f \subseteq \gamma_f$ if we assume that $\perp$ in the codomain of τ_f corresponds to the function not being defined at that point. Compare Examples 4.27 and 4.51. □

Example 4.51 (Function γ_f): Function γ_f corresponding to our example keyword set $P = \{her, his, she\}$ is shown in Figure 4.7. In keeping with the above remark, we can see that the graphical representation of the example forward trie (Example 4.27, Figure 4.5) is contained (from a graph-theoretic point of view) in Figure 4.7. □

Given function γ_f, the assignment to q in Algorithm 4.47(P$_+$, E, AC) can be written as $q := \gamma_f(q, r\lceil 1)$.

Algorithm detail 4.52 (AC-OPT): Usage of function γ_f to update variable q. □

This leads to algorithm:

Algorithm 4.53 (P$_+$, E, AC, AC-OPT):

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $q := \gamma_f(q, r\lceil 1);$
  $u, r := u(r\lceil 1), r\lfloor 1;$
  $O_e := O_e \cup Output(q) \times \{r\}$
od$\{\ PM_e\ \}$

□

Remark 4.54: Provided evaluating $\gamma_f(q, a)$ and $Output(q)$ are $\mathcal{O}(1)$ operations (for instance, if γ_f and $Output$ are tabulated), Algorithm 4.53(P$_+$, E, AC, AC-OPT) has $\mathcal{O}(|S|)$ running time complexity. □

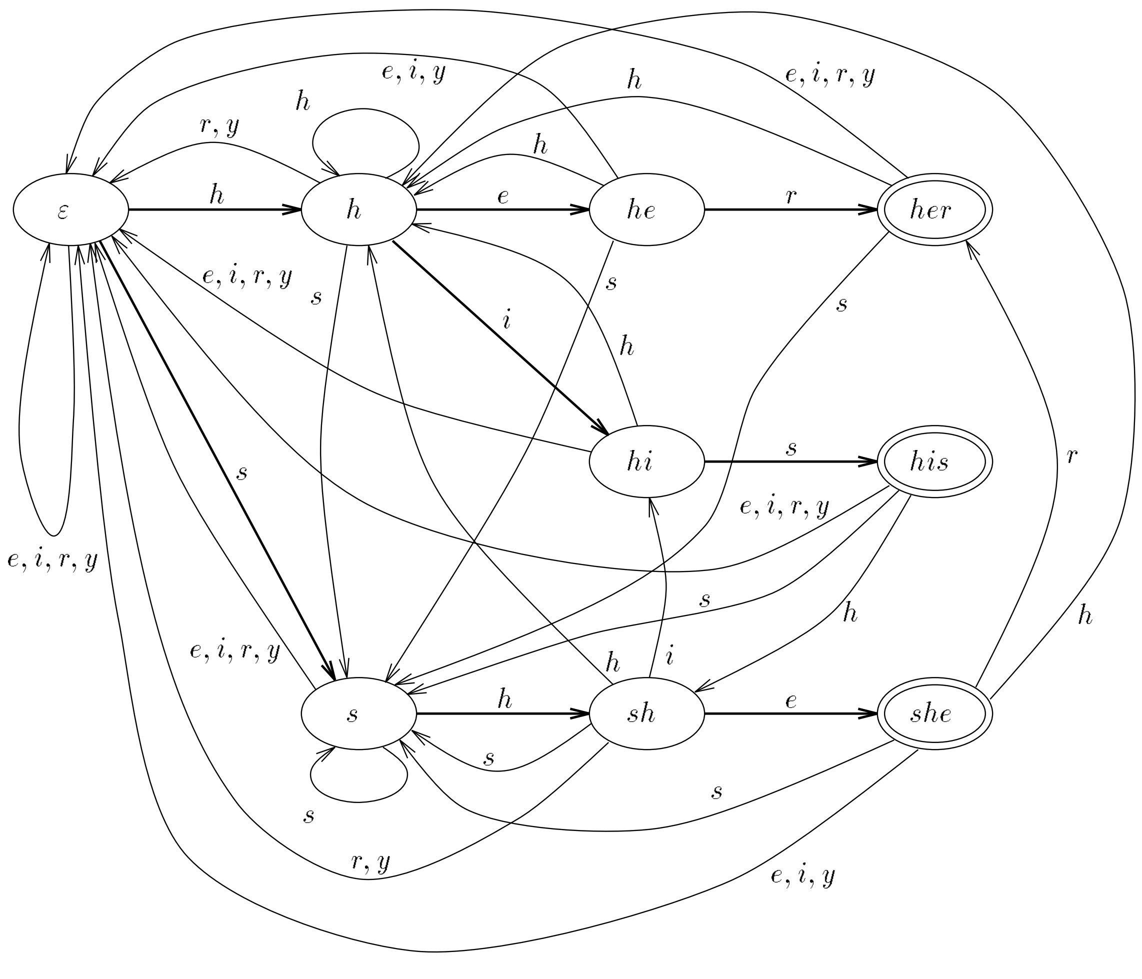


Figure 4.7: Example of function γ_f. In Remark 4.50, we mention that $\tau_f \subseteq \gamma_f$. In this figure, the edges corresponding to τ_f are thickened.

This is the Aho-Corasick optimized algorithm [AC75, Section 6]. Historically, this algorithm was usually derived from a less efficient algorithm (known as the Aho-Corasick failure function algorithm); in the next section, we will derive the failure function algorithm. The algorithm given here is known as the 'optimized' algorithm because it is able to make transitions in constant time, while using $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ time and space to precompute and store function γ_f. This algorithm would be used instead of the failure function algorithm in cases where the input string is to be processed quickly at the cost of precomputation time and space. The failure function algorithm requires $\mathcal{O}(|\mathbf{pref}(P)|)$ time to process a symbol of the input string, while only using $\mathcal{O}(|\mathbf{pref}(P)|)$ time and space for precomputed functions.

Precomputation of γ_f is discussed in [WZ92, Part II, Section 6]. It involves the so-called failure function which is introduced in Section 4.3.4.

4.3.3 A Moore machine approach to the AC-OPT algorithm

In this section, we use an automata-based approach to derive the Aho-Corasick optimized algorithm. This section may be omitted without loss of continuity.

We begin by examining the structure of Algorithm 4.38 (taken from page 61):

$u, r := \varepsilon, S;\ U := \{\varepsilon\};\ O_e := (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $U := (U(r\upharpoonright 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\};$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1;$
  $O_e := O_e \cup (U \cap P) \times \{r\}$
od$\{\ PM_e\ \}$

This algorithm bears a resemblance to algorithms used to simulate deterministic Moore machines. Thanks to encoding function enc, we know that the set of values that U can take is isomorphic to the set $\mathbf{pref}(P)$; the set of possible values for U is therefore finite (see Property 4.39). In the simulated Moore machine, variable U corresponds to the current state (while processing input string S), the expression $U \cap P$ corresponds to the output function, and variable O_e can be viewed as an encoding of the output string of the Moore machine.

Remark 4.55: There are some algorithms in the literature (see, for example, [GB-Y91]) that implement the state of the Moore machine by means of a bit vector in the case of single keyword pattern matching. For a practical use of bit vectors to represent states, see Chapter 10. □

These Moore machine observations can be made more precise in the following definition.

Definition 4.56 (Deterministic Moore machine M_0): We define deterministic Moore machine (corresponding to keyword set P) $M_0 = (Q_0, V, \Delta_0, \delta_0, \lambda_0, \{s_0\})$ as

- State set $Q_0 = \{\, \mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^* \,\}$
- Input alphabet V
- Output alphabet $\Delta_0 = \mathcal{P}(P)$
- Transition function $\delta_0 \in Q_0 \times V \longrightarrow Q_0$ defined by

$$\delta_0(q, a) = (qa \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$$

- Output function $\lambda_0 \in Q_0 \longrightarrow \Delta_0$ defined by

$$\lambda_0(q) = q \cap P$$

- Singleton start state set $\{s_0\}$ where $s_0 = \varepsilon$

Since M_0 corresponds to P, we could have named it $M_{P,0}$; since no confusion arises, we simply drop the subscript P. □

Forward reference 4.57: In Section 4.3.7, we show that Moore machine M_0 can be obtained in a different way (primarily using finite automata), while in Property 4.62 we show that Moore machine M_0 is minimal. □

The states of Moore machine M_0 are languages (sets of strings). Given the bijection *enc*, we can encode each M_0 state. This results in a Moore machine which is isomorphic to M_0 — an *MM* more easily implemented in practice. The encoding of M_0 parallels the introduction of variable q (along with functions *Output* and γ_f) to replace variable U.

We can now give the isomorphic image of M_0 under *enc*.

Definition 4.58 (Deterministic Moore machine M_1): Moore machine M_1 is the isomorphic image of M_0 under *enc*. It is

$$M_1 = (\mathbf{pref}(P), V, \mathcal{P}(P), \gamma_f, \mathit{Output}, \{\varepsilon\})$$

where

- State set $\mathbf{pref}(P)$ is the codomain of *enc*
- The input and output alphabets are unchanged
- γ_f (see Definition 4.49) is obtained as

$$\gamma_f(q, a) = \mathit{enc}(\delta_0(\mathit{enc}^{-1}(q), a)) = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(qa) \cap \mathbf{pref}(P) : w)$$

- *Output* (see Definition 4.44) is obtained as

$$\mathit{Output}(q) = \lambda_0(\mathit{enc}^{-1}(q)) = \mathbf{suff}(q) \cap P$$

- Start state ε is obtained as $\varepsilon = enc(\{\varepsilon\})$

Since M_0 depends upon P, so does M_1. □

The simulation of M_1 yields Algorithm 4.53.

Interestingly, the Moore machine M_1 is in fact the minimal deterministic Moore machine for its language (it follows from the isomorphism of M_0 and M_1 that M_0 is also minimal). This will be shown in the following definitions and properties.

Property 4.59 (Minimality of a Moore machine): A $DMM(Q, V, \Delta, \delta, \lambda, \{s\})$ is minimal if and only if all of its states are useful, and

$$(\forall\ q_0, q_1 : q_0 \neq q_1 \wedge q_0 \in Q \wedge q_1 \in Q : (\exists\ w : w \in V^* : \lambda(\delta^*(q_0, w)) \neq \lambda(\delta^*(q_1, w))))$$

□

Remark 4.60: This definition can be viewed as a generalization of a property of minimality for deterministic finite automata (see Property 2.111) — replace $\lambda(\delta^*(q, w))$ by $\delta^*(q, w) \in F$ in the definition where F is the set of final states of the finite automaton. □

Before presenting the derivation, we will require a property of function ${\gamma_f}^*$ (the extension of γ_f from domain $\mathbf{pref}(P) \times V$ to $\mathbf{pref}(P) \times V^*$).

Property 4.61 (Function ${\gamma_f}^*$): For $q \in \mathbf{pref}(P)$ and $z \in V^*$, we have

$${\gamma_f}^*(q, z) = (\mathbf{MAX}_{\leq_s}\ w : w \in \mathbf{suff}(qz) \cap \mathbf{pref}(P) : w)$$

It follows that ${\gamma_f}^*(q, z) \leq_s qz$. □

Property 4.62 (Minimality of Moore machine M_1): Moore machine M_1 is minimal.
Proof:
The following proof is by contradiction. Since all states of M_1 are useful, assume that there exist two states

$$q_0, q_1 : q_0 \in \mathbf{pref}(P) \wedge q_1 \in \mathbf{pref}(P) \wedge q_0 \neq q_1 \wedge |q_0| \geq |q_1|$$

such that

$$(\forall\ w : w \in V^* : Output({\gamma_f}^*(q_0, w)) = Output({\gamma_f}^*(q_1, w)))$$

(That is, we assume that M_1 is not minimal.) Choose $w_0 : q_0 w_0 \in P$. Then ${\gamma_f}^*(q_0, w_0) = q_0 w_0$ and $q_0 w_0 \in Output({\gamma_f}^*(q_0, w_0))$. In this case (from the assumptions)

$$\begin{array}{ll} & q_0 w_0 \in Output({\gamma_f}^*(q_0, w_0)) \\ \equiv & \quad \{\text{ assumption about } q_0 \text{ and } q_1 \} \\ & q_0 w_0 \in Output({\gamma_f}^*(q_1, w_0)) \\ \equiv & \quad \{\text{ definition of } Output \} \end{array}$$

$$
\begin{array}{ll}
 & q_0w_0 \in \mathbf{suff}({\gamma_f}^*(q_1, w_0)) \cap P \\
\Rightarrow & \quad \{ \text{definition of } \mathbf{suff}\text{; Property 4.61; transitivity of } \leq_s \} \\
 & q_0w_0 \leq_s {\gamma_f}^*(q_1, w_0) \leq_s q_1w_0 \\
\Rightarrow & \quad \{ \text{transitivity of } \leq_s \} \\
 & q_0w_0 \leq_s q_1w_0 \\
\equiv & \quad \{ \text{property of } \leq_s \} \\
 & q_0 \leq_s q_1 \\
\Rightarrow & \quad \{ |q_0| \geq |q_1| \} \\
 & q_0 = q_1
\end{array}
$$

This is a contradiction. We conclude that Moore machines M_0 and M_1 are minimal. □

4.3.4 Linear search

In this section, we return to our algorithmic derivation of the Aho-Corasick algorithms.

There are other ways of implementing the assignment in Forward reference 4.48 than the one presented in Section 4.3.2. The presence of the $\mathbf{MAX}_{\leq_s}$ on the right side of the assignment hints that a linear search could be used. Rather than using a single assignment of the form $q := \gamma_f(q, r\rceil 1)$ (as was done in Section 4.3.2), we can try to compute the $\mathbf{MAX}_{\leq_s}$ quantification by means of a linear search of the form

---

$\mathbf{do}\ \neg B(q, r\rceil 1) \rightarrow q := f(q)\ \mathbf{od}$

---

where f is a so-called *failure function* from states to states. The failure function will be used to step through decreasing (under $\leq_s$) values of q (from the maximum) until the value of the quantification is found. This may slow down the actual scanning of the subject string, but with a suitable choice of f linearity can still be maintained.

The advantages of this approach (over the use of γ_f in Section 4.3.2) lie in the lower storage requirements and in the preprocessing phase. The storage requirements decrease from $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ for γ_f to $\mathcal{O}(|\mathbf{pref}(P)|)$ for a failure function. Correspondingly, the precomputation can be performed in $\mathcal{O}(|\mathbf{pref}(P)|)$ time, as opposed to $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ time for the preprocessing of function γ_f.

Both the AC and KMP failure function algorithms make use of such a failure function, albeit in slightly different ways. Here, we give the common part of the derivations. The differences are dealt with in the two following sections.

In order to derive a specification for the linear search guard, and for the failure function, we manipulate the right side of the assignment to q (from Forward reference 4.48) into a suitable form.

We start with the first two lines of the derivation on page 63 (that derivation was used to obtain the right side of the update of q).

$$
\begin{array}{ll}
& enc((enc^{-1}(q)(r{\restriction}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}) \\
= & \quad \{ \text{definition of } enc^{-1} \} \\
& enc(((\mathbf{suff}(q) \cap \mathbf{pref}(P))(r{\restriction}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}) \\
= & \quad \{ \text{definition of } enc \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in ((\mathbf{suff}(q) \cap \mathbf{pref}(P))(r{\restriction}1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\} : w) \\
= & \quad \{ \text{for strings } w, v : w \in A \cup \{v\} \equiv w \in A \vee w = v \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(q) \cap \mathbf{pref}(P))(r{\restriction}1) \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{ \text{domain split} \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(q) \cap \mathbf{pref}(P))(r{\restriction}1) \cap \mathbf{pref}(P) : w)\, \mathbf{max}_{\leq_s} \varepsilon \\
= & \quad \{ \text{change of bound variable: } w = w'(r{\restriction}1) \} \\
& (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\restriction}1) \in \mathbf{pref}(P) : w'(r{\restriction}1))\, \mathbf{max}_{\leq_s} \varepsilon
\end{array}
$$

A linear search cannot be used to easily compute the above expression directly. In the next two sections, the expression will be further manipulated for the specific linear searches.

Forward reference 4.63: Linear search will be used in Sections 4.3.5 and 4.3.6 to compute

$$(\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\restriction}1) \in \mathbf{pref}(P) : w'(r{\restriction}1))\, \mathbf{max}_{\leq_s} \varepsilon$$

□

The use of linear search is expressed in the following program detail.

Algorithm detail 4.64 (LS): Using linear search to update the state variable q. □

4.3.5 The Aho-Corasick failure function algorithm

In order to simplify the linear search, we would like to compute the following $\mathbf{MAX}_{\leq_s}$ quantification (as an intermediate step in computing the one in Forward reference 4.63):

$$(\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\restriction}1) \in \mathbf{pref}(P) : w')$$

There is a potential problem with this approach: when the linear search computes ε as the value of this quantification, an **if-fi** statement is required to decide which of the following two situations gave rise to the ε (and therefore what the value of the quantification in Forward reference 4.63 is):

- $\neg(\exists\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\restriction}1) \in \mathbf{pref}(P) : w')$. The $\mathbf{MAX}_{\leq_s}$ quantification has an empty range, and therefore value ε (the unit of $\mathbf{max}_{\leq_s}$).
- $(r{\restriction}1) \in \mathbf{pref}(P)$. The $\mathbf{MAX}_{\leq_s}$ quantification does not have an empty range, and the value of the quantification is ε.

The linear search will make use of the following failure function.

Definition 4.65 (Failure function f_f): Function $f_f \in \mathbf{pref}(P) \longrightarrow \mathbf{pref}(P)$ is defined as

$$f_f(q) = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q) \setminus \{q\} \cap \mathbf{pref}(P) : w)$$

Note that $f_f(\varepsilon) = \varepsilon$ since ε is the unit of $\mathbf{max}_{\leq_s}$. □

Remark 4.66: The subscript f in the failure function f_f is for *forward*. In Chapter 5, we will use a *reverse* failure function f_r which will be defined analogously. In that chapter, a precomputation algorithm for f_r is given; that algorithm could easily be modified to compute f_f. □

Example 4.67 (Failure function): The failure function corresponding to our example keyword set is:

w	ε	h	s	he	hi	sh	her	his	she
$f_f(w)$	ε	ε	ε	ε	ε	h	ε	s	he

□

Using f_f, the resulting linear search is:

$q' := q;$
$\mathbf{do}\ q' \neq \varepsilon \wedge q'(r{\upharpoonright}1) \notin \mathbf{pref}(P) \rightarrow q' := f_f(q')\ \mathbf{od};$
$\{\ (q' = \varepsilon \wedge \neg(\exists\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) : w'(r{\upharpoonright}1) \in \mathbf{pref}(P)))$
$\quad \vee\ q' = (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\upharpoonright}1) \in \mathbf{pref}(P) : w')\ \}$
$\mathbf{if}\ q' = \varepsilon \wedge (r{\upharpoonright}1) \notin \mathbf{pref}(P) \rightarrow q := \varepsilon$
$[\!]\ \ q' \neq \varepsilon \vee (r{\upharpoonright}1) \in \mathbf{pref}(P) \rightarrow q := q'(r{\upharpoonright}1)$
$\mathbf{fi}$
$\{\ q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P) : w)\ \}$

The second conjunct in the guard of the repetition can be evaluated cheaply using the forward trie τ_f, since $q'(r{\upharpoonright}1) \notin \mathbf{pref}(P) \equiv \tau_f(q', (r{\upharpoonright}1)) = \perp$. However, by extending the forward trie τ_f, we can use it to evaluate both of the repetition guard conjuncts[4].

Definition 4.68 (Extended forward trie corresponding to P): The extended forward trie is function $\tau_{ef} \in \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P) \cup \{\perp\}$ defined by

$$\tau_{ef}(w,a) = \begin{cases} wa & \text{if } wa \in \mathbf{pref}(P) \\ \varepsilon & \text{if } w = \varepsilon \wedge a \notin \mathbf{pref}(P) \\ \perp & \text{otherwise} \end{cases}$$

□

[4]This is essentially an application of the sentinal technique often used with linear search.

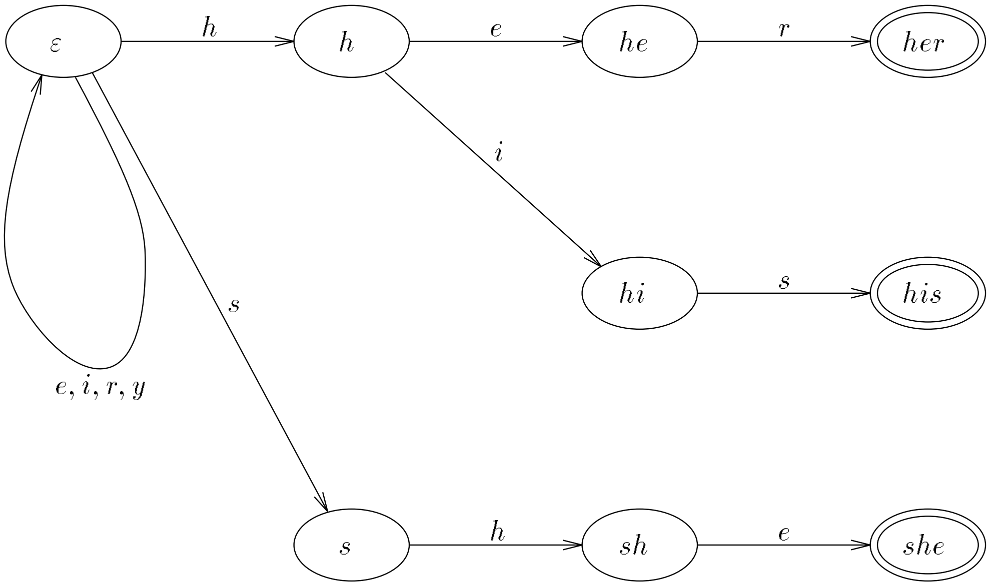


Figure 4.8: Example of function τ_{ef}.

Note that $\tau_f(q, a) = \tau_{ef}(q, a)$ except when $q = \varepsilon \wedge a \notin \mathbf{pref}(P)$ where $\tau_f(q, a) = \perp$ and $\tau_{ef}(q, a) = \varepsilon$.

Property 4.69 (Extended forward trie): Both conjuncts of the linear search guard can be combined since

$$q' \neq \varepsilon \wedge q'(r\lceil 1) \notin \mathbf{pref}(P) \equiv \tau_{ef}(q', r\lceil 1) = \perp$$

As a side effect of the introduction of τ_{ef}, the **if**-**fi** statement after the linear search can be replaced by the single assignment statement $q := \tau_{ef}(q', r\lceil 1)$. □

Example 4.70 (Extended forward trie): Function τ_{ef} corresponding to our example keyword set $P = \{her, his, she\}$ is shown in Figure 4.8. Visually, we can see that τ_{ef} is simply an extension of the forward trie τ_f, by comparing this figure with Figure 4.5 given in Example 4.27. □

Algorithm detail 4.71 (AC-FAIL): Introduction of the extended forward trie τ_{ef} and the failure function f_f to implement the linear search updating state variable q. □

We can now eliminate variable q' from the linear search, to obtain the following algorithm:

Algorithm 4.72 (P$_+$, E, AC, LS, AC-FAIL):

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := \mathit{Output}(q) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  do $\tau_{ef}(q, r{\upharpoonright}1) = \perp \rightarrow q := f_f(q)$ **od**;
  $q := \tau_{ef}(q, r{\upharpoonright}1);$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $O_e := O_e \cup \mathit{Output}(q) \times \{r\}$
od{ PM_e }

□

This algorithm is the Aho-Corasick failure function pattern matching algorithm [AC75, Section 2, Algorithm 1]. In Aho and Corasick's original paper, this algorithm is derived first; it is then used as a starting point to derive the optimized AC algorithm.

This algorithm still has $\mathcal{O}(|S|)$ running time complexity [Aho90] but is less efficient than Algorithm 4.53(P$_+$, E, AC, AC-OPT). Function τ_{ef} can be stored more efficiently than function γ_f, requiring $\mathcal{O}(|\mathbf{pref}(P)|)$ space. Precomputation of extended forward trie τ_{ef} and failure function f_f is discussed in [WZ92, Part II, Section 6].

Since (in this algorithm) the failure function f_f is never applied to ε, we can restrict its domain to $f_f \in \mathbf{pref}(P) \setminus \{\varepsilon\} \longrightarrow \mathbf{pref}(P)$. The function with the restricted domain is slightly cheaper to precompute than the full function. In the next section, we will make use of the full signature of f_f.

4.3.6 The Knuth-Morris-Pratt algorithm

In this section, we would like to use a simpler linear search (compared to that used in the previous section) to compute the following expression (from Forward reference 4.63):

$$(\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \land w'(r{\upharpoonright}1) \in \mathbf{pref}(P) : w'(r{\upharpoonright}1))\, \mathbf{max}_{\leq_s} \varepsilon$$

We would like to rewrite the above quantification into

$$(\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \land w'(r{\upharpoonright}1) \in \mathbf{pref}(P) : w')(r{\upharpoonright}1)$$

In this case, the quantification could be calculated by linear search, but (as in the previous section) we would still need to distinguish between two cases when the value of the quantification is ε (see page 70). In order to avoid this, we extend operator $\mathbf{max}_{\leq_s}$ (from being a binary operator on V^* to a binary operator on $V^* \cup \{\perp_s\}$ for some new element $\perp_s$) and we make this new element the unit of $\mathbf{max}_{\leq_s}$. (Making it the unit will allow us to identify the empty range case of the above quantification; this required an **if**-**fi** statement in our first linear search algorithm in the previous section.)

Definition 4.73 (Extension of $\mathbf{max}_{\leq_s}$): Extend $\mathbf{max}_{\leq_s}$ to be an associative and commutative binary operator on $V^* \cup \{\perp_s\}$, with $(\forall\ w : w \in V^* : w\,\mathbf{max}_{\leq_s} \perp_s = w)$ — that is, $\perp_s$ is the unit of $\mathbf{max}_{\leq_s}$. We also define $\perp_s$ to be the zero of string concatenation — that is, $(\forall\ w : w \in V^* : (w\perp_s = \perp_s) \wedge (\perp_s w = \perp_s))$. By Notation 2.6 we have $(\mathbf{MAX}_{\leq_s}\, w : w \in \emptyset : w) = \perp_s$. □

The property that $\perp_s$ is the zero of string concatenation will be used later in the following derivation. We can now rewrite the expression from Forward reference 4.63

$$\begin{aligned} & (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\upharpoonright}1) \in \mathbf{pref}(P) : w'(r{\upharpoonright}1))\,\mathbf{max}_{\leq_s} \varepsilon \\ = & \qquad \{\text{ Definition 4.73 — } \perp_s \text{ is zero of concatenation and the unit of } \mathbf{max}_{\leq_s} \} \\ & (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w'(r{\upharpoonright}1) \in \mathbf{pref}(P) : w')(r{\upharpoonright}1)\,\mathbf{max}_{\leq_s} \varepsilon \end{aligned}$$

We are left with an expression containing a simpler $\mathbf{MAX}_{\leq_s}$ quantification (the quantification is the same as the one given at the beginning of Section 4.3.5). It is therefore easier to implement a linear search to compute the value of this quantification. The linear search will traverse elements of $(\mathbf{suff}(q) \cap \mathbf{pref}(P)) \cup \{\perp_s\}$.

This straight-forward linear search yields the KMP algorithm. Before presenting the linear search, we note that we will have a failure function with the same definition as was given in Definition 4.65. The only change is: with the extension of $\mathbf{max}_{\leq_s}$, we have $f_f(\varepsilon) = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(\varepsilon) \setminus \{\varepsilon\} : w) = (\mathbf{MAX}_{\leq_s}\, w : w \in \emptyset : w) = \perp_s$ instead of $f_f(\varepsilon) = \varepsilon$. (This version of the failure function f_f is sometimes called the *extended failure function.*)

To compute the desired $\mathbf{MAX}_{\leq_s}$ quantification we can simply use the linear search:

$q' := q;$
do $q' \neq \perp_s$ **cand** $q'(r{\upharpoonright}1) \notin \mathbf{pref}(P) \rightarrow q' := f_f(q')$ **od**;
$\{\ q' = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge w(r{\upharpoonright}1) \in \mathbf{pref}(P) : w)\ \}$
$q := q'(r{\upharpoonright}1)\,\mathbf{max}_{\leq_s} \varepsilon$
$\{\ q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q(r{\upharpoonright}1)) \cap \mathbf{pref}(P) : w)\ \}$

Remark 4.74: Strictly speaking, the conditional conjunction in the repetition guard could also be written as a normal (unconditional) conjunction because $\perp_s$ is the zero of concatenation. As we shall see in Algorithm 4.84, conditional conjunction is necessary when certain coding tricks are used. □

Algorithm detail 4.75 (KMP-FAIL): The extended failure function f_f is introduced to implement the linear search for the update of q. □

Eliminating variable q' in the linear search leads to algorithm

Algorithm 4.76 (P$_+$, E, AC, LS, KMP-FAIL):

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  do $q \neq \bot_s$ **cand** $q(r{\upharpoonright}1) \notin$ **pref**$(P) \rightarrow q := f_f(q)$ **od**;
  $q := q(r{\upharpoonright}1)\ \mathbf{max}_{\leq_s} \varepsilon;$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $O_e := O_e \cup Output(q) \times \{r\}$
od$\{\ PM_e\ \}$

□

This algorithm does not appear in the literature. In some cases, the linear search in the algorithm above performs one more iteration than the one in Algorithm 4.72, meaning that it is slightly less efficient.

4.3.6.1 Adding indices

Historically, the KMP algorithm was designed using indexing within strings; this stems from efficiency concerns. Some of the most common uses of the KMP algorithm are in file-search programs and text editors, in which pointers to memory containing a string are a preferable method of accessing strings. In order to show the equivalence of this more abstract version of KMP, and the classically presented version we will now convert Algorithm 4.76 to make use of indexing within strings. To facilitate the use of indexing, we have to restrict the problem to the one keyword case, as stated in problem detail

Problem detail 4.77 (OKW): $P = \{p\}$ □

Convention 4.78 (Shadow variables): Most shadow predicates and functions will be 'hatted' for easy identification. Variables i and j are so named (and not hatted) to conform to the original publication of the algorithms. □

We now introduce three shadow variables, and invariants that are maintained between the shadow variables and the existing program variables:

- $i : q = p_1 \ldots p_{i-1}$ where $i = 1 \equiv q = \varepsilon$ and $i = 0 \equiv q = \bot_s$. With this convention we mirror the coding trick from the original KMP algorithm.
- $j : u = S_1 \ldots S_{j-1} \wedge r = S_j \ldots S_{|S|}$. Also $r{\upharpoonright}1 = S_j$ if $1 \leq j \leq |S|$.
- $\widehat{O_e} : O_e = (\cup\ x : x \in \widehat{O_e} : \{(p, S_x \cdots S_{|S|})\})$.

Naturally, we must define new predicates and a new failure function $\widehat{f_f}$ on these shadow variables.

Definition 4.79 (Indexing extended failure function): Define $\widehat{f_f} \in [1, |p|+1] \longrightarrow [0, |p|]$ as

$$\widehat{f_f}(i) = |f_f(p_1 \dots p_{i-1})| + 1$$

and define $|\bot_s| = -1$. □

Example 4.80 (Indexing extended failure function): For this single-keyword example, we assume the keyword $p = hehshe$. In this case, our failure function f_f is given as:

w	ε	h	he	heh	$hehs$	$hehsh$	$hehshe$
$f_f(w)$	$\bot_s$	ε	ε	h	ε	h	he

The corresponding indexing failure function $\widehat{f_f}$ is:

i	1	2	3	4	5	6	7
$\widehat{f_f}(i)$	0	1	1	2	1	2	3

□

The invariant relating u and q ($q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)$) can be rewritten to relate j and i:

$$p_1 \dots p_{i-1} = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(S_1 \dots S_{j-1}) \cap \mathbf{pref}(p) : w))$$

Definition 4.81 (New postcondition $\widehat{PM_e}$): Postcondition PM_e can be rewritten in terms of the shadow variables:

$$\widehat{PM_e} : \widehat{O_e} = (\cup\; j : 1 \leq j \leq |S| + 1 \wedge p \in \mathbf{suff}(S_1 \cdots S_{j-1}) : \{j\})$$

□

We can also note the following equivalences and correspondences:

- Since $q \in \mathbf{pref}(p)$ we have $q(r\lceil 1) \notin \mathbf{pref}(p) \equiv S_j \neq p_i$ when $i \leq |p| \wedge j \leq |S|$. Similarly $q \neq \bot_s \equiv 0 < i$ and $q = p \equiv i = |p| + 1$.
- Assignment $q := q(r\lceil 1)\, \mathbf{max}_{\leq_s} \varepsilon$ corresponds to $i := i + 1$. It is here that we use the coding trick alluded to in Remark 4.74.
- Assignment $u, r := u(r\lceil 1), r\lfloor 1$ corresponds to $j := j + 1$.
- The guard can be rewritten using the following equivalence $r \neq \varepsilon \equiv j \leq |S|$.
- Assignment $O_e := O_e \cup Output(q) \times \{r\}$ corresponds to

---

$\textbf{if } i = |p| + 1 \rightarrow \widehat{O_e} := \widehat{O_e} \cup \{j\}$
$[\!] \;\; i \neq |p| + 1 \rightarrow \textbf{skip}$
$\textbf{fi}$

The complete algorithm (written without the invariants relating shadow to non-shadow variables) is now:

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
$i := 1;\ j := 1;$
$\textbf{if } i = |p| + 1 \rightarrow \widehat{O_e} := \{j\}$
$[\!] \;\; i \neq |p| + 1 \rightarrow \widehat{O_e} := \emptyset$
$\textbf{fi};$
$\textbf{do } j \leq |S| \rightarrow$
$\quad \textbf{do } 0 < i \textbf{ cand } S_j \neq p_i \rightarrow q := f_f(q);\ i := \widehat{f}_f(i) \textbf{ od};$
$\quad q := q(r\upharpoonright 1) \,\textbf{max}_{\leq_s} \varepsilon; i := i + 1;$
$\quad u, r := u(r\upharpoonright 1), r\downharpoonright 1;\ j := j + 1;$
$\quad O_e := O_e \cup Output(q) \times \{r\};$
$\quad \textbf{if } i = |p| + 1 \rightarrow \widehat{O_e} := \widehat{O_e} \cup \{j\}$
$\quad [\!] \;\; i \neq |p| + 1 \rightarrow \textbf{skip}$
$\quad \textbf{fi}$
$\textbf{od}\{\ PM_e \wedge \widehat{PM}_e\ \}$

We have introduced algorithm detail:

Algorithm detail 4.82 (INDICES)**:** Represent substrings by indices into the complete strings. □

Remark 4.83: While we have introduced (INDICES) as an algorithm detail, it could also be considered as a problem detail since it is being used to derive an algorithm which satisfies a postcondition given in terms of indices. We will continue to call it an algorithm detail, since we will use it as a pure algorithm detail in Section 4.5. □

Removing the non-shadow variables leaves us with the following algorithm:

Algorithm 4.84 (P_+, E, AC, LS, KMP-FAIL, OKW, INDICES)**:**

$i := 1;\ j := 1;$
$\textbf{if } i = |p| + 1 \rightarrow \widehat{O_e} := \{j\}$
$[\!] \;\; i \neq |p| + 1 \rightarrow \widehat{O_e} := \emptyset$

fi;
do $j \leq |S| \rightarrow$
  do $0 < i$ **cand** $S_j \neq p_i \rightarrow i := \widehat{f_f}(i)$ **od**;
  $i := i + 1$;
  $j := j + 1$;
  if $i = |p| + 1 \rightarrow \widehat{O_e} := \widehat{O_e} \cup \{j\}$
  $[\!]\ i \neq |p| + 1 \rightarrow$ **skip**
  fi
od$\{\ \widehat{PM_e}\ \}$

$\square$

The above algorithm is the classic Knuth-Morris-Pratt algorithm [KMP77, Section 2, p. 326]. This algorithm has $\mathcal{O}(|S|)$ running time, and it has been shown by Perrin [Perr90, p. 32] that the number of applications of $\widehat{f_f}$ (the total number of iterations of the outer repetition and the inner repetition) is never greater than $2 \cdot |S|$. Storage of $\widehat{f_f}$ requires $\mathcal{O}(|p|)$ space. Precomputation of function $\widehat{f_f}$ can easily be derived by converting, in a similar way to above, the precomputation of function f_f (as discussed in [WZ92, Part II, Section 6]) into using indices.

4.3.7 An alternative derivation of Moore machine M_0

An interesting solution to the pattern matching problem involves using an automaton for the language V^*P. Usually, a nondeterministic finite automaton is constructed. The automaton is then simulated, processing input string S, and considering all paths through the automaton. Whenever a final state is entered (after processing string u, a prefix of S), a keyword match has been found (since $u \in V^*P$, equivalently $\mathbf{suff}(u) \cap P \neq \emptyset$, by Property 2.59) and the match is registered; see for example [AHU74, p. 327] for a description of this approach.

One particular (ε-transition-free) transition function for the automaton is simply the forward trie for P, augmented with a transition from state ε to itself on all symbols in V (recall that $\mathbf{pref}(P)$ is the state set of the forward trie — see Definition 4.26). This automaton is defined as $(Q_N, V, \delta_N, \emptyset, \{s_N\}, F_N)$, where

- State set $Q_N = \mathbf{pref}(P)$
- The input alphabet V
- Transition function (trie-based) $\delta_N \in Q_N \times V \longrightarrow \mathcal{P}(Q_N)$ is defined (for $q = \varepsilon$) by

$$\delta_N(q, a) = \begin{cases} \{\varepsilon, a\} & \text{if } a \in \mathbf{pref}(P) \\ \{\varepsilon\} & \text{otherwise} \end{cases}$$

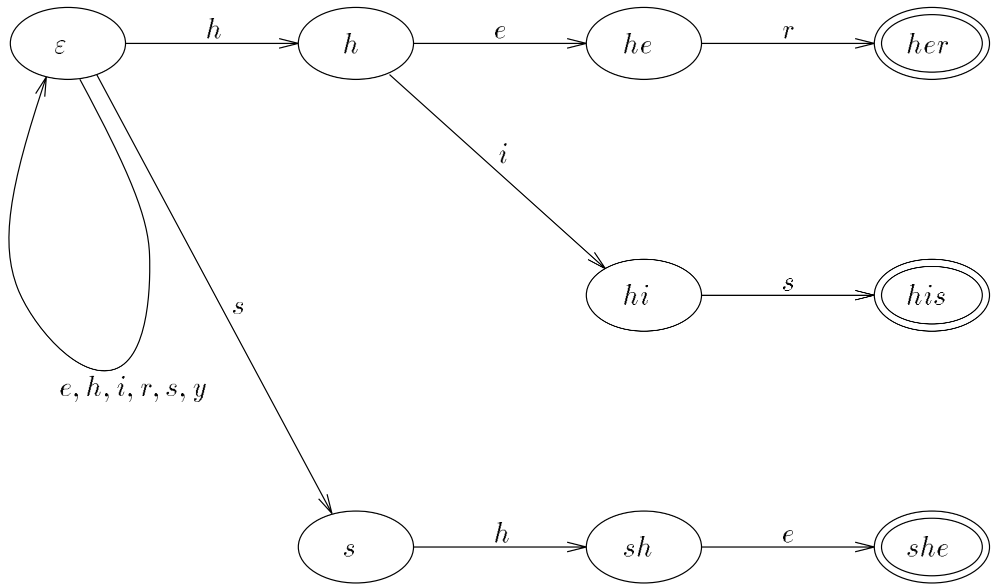


Figure 4.9: Example of function δ_N. Notice the similarity with function τ_{ef}.

and (for $q \neq \varepsilon$)

$$\delta_N(q,a) = \begin{cases} \{qa\} & \text{if } qa \in \mathbf{pref}(P) \\ \emptyset & \text{otherwise} \end{cases}$$

and is extended to $\delta_N^* \in Q_N \times V^* \longrightarrow \mathcal{P}(Q_N)$ in the usual way

- Single start state $s_N = \varepsilon$
- Final state set $F_N = P$

It is useful to see the graphical representation of transition function δ_N.

Example 4.85 (Function δ_N): The function δ_N corresponding to our keyword set $P = \{her, his, she\}$ is shown in Figure 4.9. □

The simulation of this automaton can proceed as follows:

$u, r := \varepsilon, S;\ q_N := \{s_N\};$
$O_e := (q_N \cap F_N) \times \{r\};$
$\{$ invariant: $q_N = \delta_N^*(\varepsilon, u)$ $\}$
do $r \neq \varepsilon \rightarrow$
  $q_N := (\cup\ q : q \in q_N : \delta_N(q, r\upharpoonright 1));$
  $u, r := u(r\upharpoonright 1), u\downharpoonright 1;$

$O_e := O_e \cup (q_N \cap F_N) \times \{r\}$
od$\{ PM_e \}$

Strictly speaking, the automaton is being used as a nondeterministic Moore machine. Each path through the Moore machine is followed simultaneously; the output function is only defined for some of the states (the final states, $F_N = P$ to be precise). The output alphabet Δ_N can be written as $\Delta_N = P \cup \{\perp_N\}$ ($\perp_N$ is output in nonmatching, i.e. non-final states). The output function is $\lambda_N \in Q_N \longrightarrow \Delta_N$ defined as

$$\lambda_N(q) = \begin{cases} q & \text{if } q \in F_N \\ \perp_N & \text{if } q \notin F_N \end{cases}$$

The nondeterministic Moore machine is now $M_N = (Q_N, V, \Delta_N, \delta_N, \lambda_N, \{s_N\})$. In the algorithm, the set O_e is only updated when the output is not $\perp_N$.

The subset construction (with unreachable state removal) can be applied to the nondeterministic Moore machine, to give a deterministic Moore machine M_D. That is, $M_D = (\mathit{useful}_s \circ \mathit{subsetmm})(M_N)$. In the following property, we will prove that $M_D = M_0$ (deterministic Moore machine M_0 was defined in Definition 4.56).

In the derivation that follows, we will use an interesting property of automaton M_N.

Property 4.86 (Transition function δ_N): For all states $q \in Q_N$, the left language of q is V^*q. We write $\overleftarrow{\mathcal{L}}(q) = V^*q$. This follows from the fact that the only cycles in the transition graph are from start state s_N to itself on every $a \in V$. □

Under the subset construction, the state set is $\mathcal{P}(Q_N) = \mathcal{P}(\mathbf{pref}(P))$. The set of reachable states is smaller, as will be shown below. A new output alphabet (under the subset construction) is defined as: $\Delta_D = \mathcal{P}(\Delta_N)$. The set of start-reachable states is

$$\begin{aligned}
& Q_D \\
= & \quad \{ \text{Transformation 2.124 and Property 2.96} \} \\
& \{ q \mid q \in \mathcal{P}(Q_N) \wedge \overleftarrow{\mathcal{L}}_{M_D}(q) \neq \emptyset \} \\
= & \quad \{ \text{Property 2.123 — subset construction} \} \\
& \{ q \mid q \in \mathcal{P}(Q_N) \wedge (\cap\, p : p \in q : \overleftarrow{\mathcal{L}}_{M_N}(p)) \neq \emptyset \} \\
= & \quad \{ \text{Property 2.107 — disjoint left languages in a } \mathit{DMM} \} \\
& \{ \{ p \mid p \in Q_N \wedge w \in \overleftarrow{\mathcal{L}}_{M_N}(p) \} \mid w \in V^* \} \\
= & \quad \{ \text{definition: } Q_N = \mathbf{pref}(P) \text{ and Property 4.86} \} \\
& \{ \{ p \mid p \in \mathbf{pref}(P) \wedge w \in V^*p \} \mid w \in V^* \} \\
= & \quad \{ \text{Property 2.59: } w \in V^*p \equiv p \in \mathbf{suff}(w) \} \\
& \{ \{ p \mid p \in \mathbf{pref}(P) \wedge p \in \mathbf{suff}(w) \} \mid w \in V^* \} \\
= & \quad \{ \text{definition of } \cap\text{: } p \in \mathbf{pref}(P) \wedge p \in \mathbf{suff}(w) \equiv p \in \mathbf{pref}(P) \cap \mathbf{suff}(w) \}
\end{aligned}$$

$$
\begin{array}{ll}
 & \{\,\{\,p \mid p \in \mathbf{pref}(P) \cap \mathbf{suff}(w)\,\} \mid w \in V^*\,\} \\
= & \qquad \{\,\text{set calculus}\,\} \\
 & \{\,\mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^*\,\} \\
= & \qquad \{\,\text{definition of } Q_0\,\} \\
 & Q_0
\end{array}
$$

The deterministic output function $\lambda_D \in Q_D \longrightarrow \mathcal{P}(\Delta_N)$ is

$$
\begin{array}{ll}
 & \lambda_D(q) \\
= & \qquad \{\,\text{Transformation 2.124 — subset construction}\,\} \\
 & \{\,\lambda_N(p) \mid p \in q \wedge \lambda_N(p) \neq \perp_N\,\} \\
= & \qquad \{\,\text{definition of } \lambda_N\,\} \\
 & \{\,p \mid p \in q \wedge p \in F_N\,\} \\
= & \qquad \{\,\text{definition: } F_N = P\,\} \\
 & \{\,p \mid p \in q \wedge p \in P\,\} \\
= & \qquad \{\,\text{definition of } \cap\,\} \\
 & \{\,p \mid p \in q \cap P\,\} \\
= & \qquad \{\,\text{set calculus}\,\} \\
 & q \cap P \\
= & \qquad \{\,\text{definition } \lambda_0\,\} \\
 & \lambda_0(q)
\end{array}
$$

Lastly, the deterministic transition function $\delta_D \in Q_D \times V \longrightarrow Q_D$ is

$$
\begin{array}{ll}
 & \delta_D(q, a) \\
= & \qquad \{\,\text{Transformation 2.124 — subset construction}\,\} \\
 & (\cup\ p : p \in q : \delta_N(p, a)) \\
= & \qquad \{\,\text{definition of } \delta_N,\ \varepsilon \in q\,\} \\
 & (\cup\ p : p \in q \wedge pa \in \mathbf{pref}(P) : \{pa\}) \cup \{\varepsilon\} \\
= & \qquad \{\,\text{set calculus}\,\} \\
 & (qa \cap \mathbf{pref}(P)) \cup \{\varepsilon\} \\
= & \qquad \{\,\text{definition of } \delta_0\,\} \\
 & \delta_0(q, a)
\end{array}
$$

From these derivations it follows that $M_D = M_0$.

Remark 4.87: Notice that the number of states of the Moore machine does not grow during the subset construction. Perrin mentions the Aho-Corasick and Knuth-Morris-Pratt Moore machines as examples of ones which do not suffer from exponential blowup (i.e. the number of states grows exponentially) during the subset construction [Perr90].

Indeed, we have shown a stronger result: the AC and KMP Moore machine (we use the singular since they are isomorphic) does not suffer from any increase in the number of states under the subset construction (with start-unreachable states removed). □

4.4 The Commentz-Walter algorithms

In this section, we discuss a number of algorithms that can be derived from the naïve Algorithm 4.18, viz. the Commentz-Walter (CW) algorithms [Com79a, Com79b] and a multiple keyword version of the Boyer-Moore (BM) algorithm. The original single keyword version of the BM algorithm [BM77], and those variants considered in [HS91], will be discussed in Section 4.5. All of the algorithms derived in this section are also derived (with precomputation algorithms) in [WZ95]. In that paper, some of the algorithm details are given different names, and part of the algorithm graph has a slightly different structure.

We will be using Algorithm 4.18(P_+, S_+, RT) as the starting point for algorithms in this section. For easy cross-referencing, we duplicate that algorithm here:

Algorithm 4.88 (P_+, S_+, RT):

$u, r := \varepsilon, S;\ O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $l, v := u, \varepsilon;\ O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\};$
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l{\upharpoonright}1) \neq \perp \rightarrow$
    $l, v := l{\downharpoonright}1, (l{\upharpoonright}1)v;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od
od$\{$ *PM* $\}$

□

This algorithm traverses the subject string from left to right but does matching from right to left. As soon as a match fails, the starting point for matching is 'shifted' to the right by the assignment $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1$ and the matching starts again. The essential algorithm detail added in this section is that of shifts of the form $u, r := u(r{\upharpoonright}k), r{\downharpoonright}k$ for k possibly greater than 1 (provided no matches are missed, of course). 'Safe' shift distances can be determined from the symbols inspected during a match and some precomputed tables specific to the patterns. This idea was introduced in the original BM single keyword algorithm [BM77], which turns out to be extremely efficient in practice and has become very popular. The algorithm was extended to multiple keywords by Commentz-Walter [Com79a, Com79b], much in the same way as the Aho-Corasick algorithms relate to the Knuth-Morris-Pratt algorithm. In Chapter 5, we will extend the CW algorithm to handle patterns that are arbitrary regular expressions. That extension answers an open question first posed by A.V. Aho in 1980 [Aho80, p. 342].

It turns out that the CW algorithm is little-used in practice, if at all, due to a somewhat inaccessible description and the problem of correctly carrying out the rather intricate precomputations. The algorithm deserves better: extensive benchmarking, reported in Chapter 13, shows that the CW algorithm significantly outperforms the better-known AC algorithms in many cases.

The algorithms presented in this section all use the same algorithm skeleton for scanning the subject string, but differ in the shift distances used and the way these are computed. We now present the derivation of the common part; specific shift distances are treated in Sections 4.4.2–4.4.8. The solid lines and solid circles of Figure 4.10 indicate the part of the taxonomy which we will be considering in this section.

In the next section, we will outline a general method of computing a 'safe' shift distance.

4.4.1 Safe shift distances and predicate weakening

We begin by characterizing the 'ideal' shift distance that can be used in the assignment $u, r := u(r\upharpoonright k), r\downharpoonright k$. Ideally, we would like to shift to the next keyword match to the right (of the current position), a distance of $(\mathbf{MIN}\ n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\upharpoonright n)) \cap P \neq \emptyset : n)$. (Note that this quantification can have an empty range if there is no 'next match', and therefore take value $+\infty$. For this reason, we extend the take and drop operators such that $r\upharpoonright(+\infty) = r$ and $r\downharpoonright(+\infty) = \varepsilon$.) This ideal shift distance can be explained intuitively as the smallest shift distance $n \geq 1$ such that a suffix of $u(r\upharpoonright n)$ is a keyword and therefore a match will be found while scanning $u(r\upharpoonright n)$ from right to left.

Any smaller shift is also appropriate, and we define a safe shift distance as follows.

Definition 4.89 (Safe shift distance): A shift distance k satisfying

$$1 \leq k \leq (\mathbf{MIN}\ n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\upharpoonright n)) \cap P \neq \emptyset : n)$$

is called a *safe shift distance.* □

The use of a safe shift distance is embodied in the following algorithm detail.

Algorithm detail 4.90 (CW): A safe shift distance, k, is used in the assignment

$$u, r := u(r\upharpoonright k), r\downharpoonright k$$

of Algorithm 4.18(P$_+$, S$_+$, RT). □

In [WZ95], this algorithm detail has been renamed SSD (for 'safe shift distance').

Computing the upperbound on k (the maximal safe shift distance) is essentially the same as the problem we are trying to solve, and we aim at easier to compute approximations (from below) of the upperbound. Thanks to the following property, we can weaken the range predicate of the ideal shift to obtain an approximation.

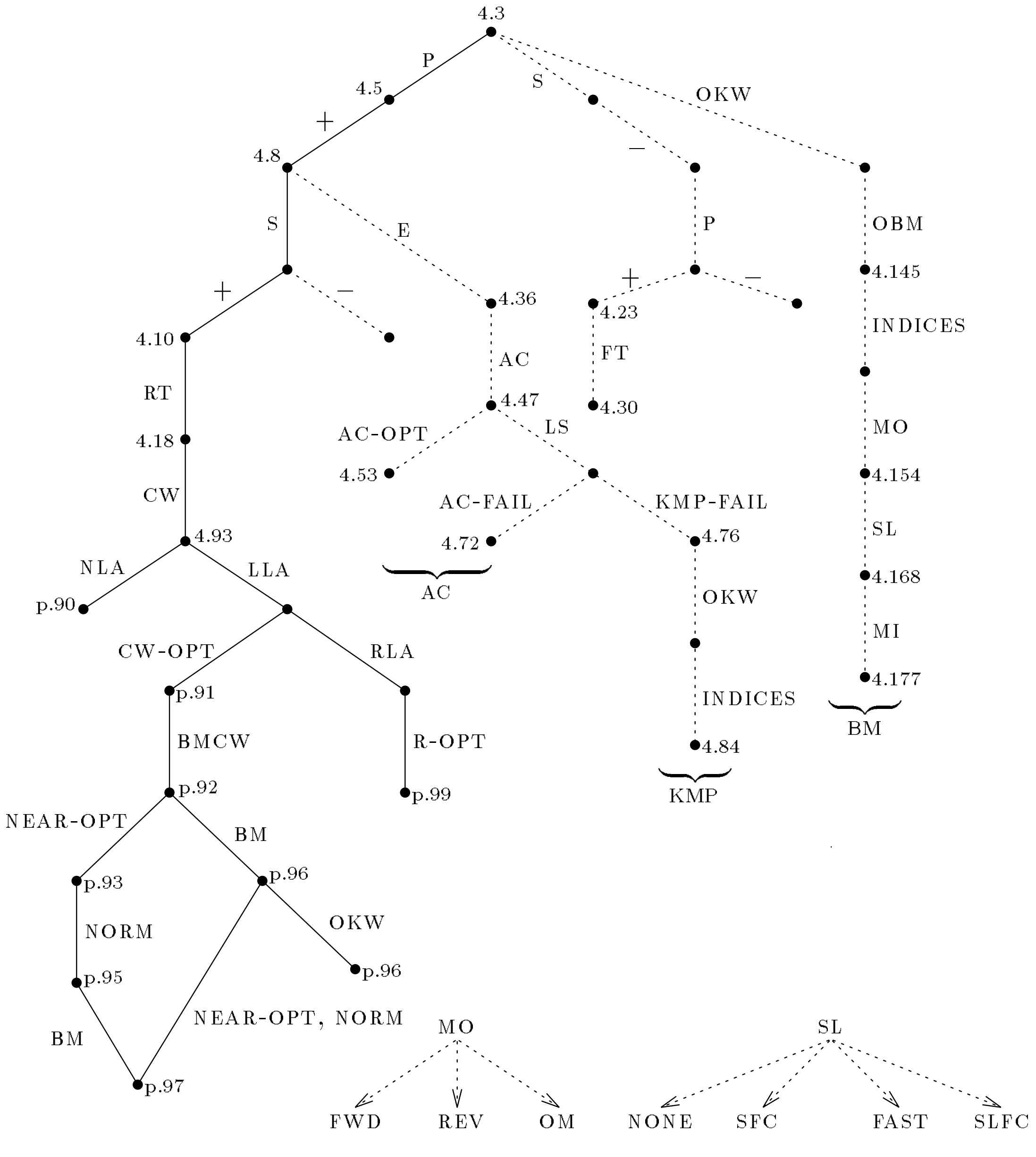


Figure 4.10: The algorithms considered in Section 4.4 are denoted by solid circles, connected by solid lines.

Property 4.91 (Weakening of range predicates): For predicates J and J' such that $J \Rightarrow J'$, we have

$$(\mathbf{MIN}\ i : J(i) : i) \geq (\mathbf{MIN}\ i : J'(i) : i)$$

□

Approximations will be obtained by weakening the predicate $\mathbf{suff}(u(r\rceil n)) \cap P \neq \emptyset$ in the range of the upperbound. Since the ideal predicate $(\mathbf{suff}(u(r\rceil n)) \cap P \neq \emptyset)$ implies its weakenings, the quantification with the weakening in the range will not be greater than the ideal shift (i.e. it will approximate the ideal shift distance from below, and will be a safe shift). The weakest predicate *true* is one such weakening; using it yields a shift distance of 1.

By considering different weakenings, several variants of the CW algorithm (amongst which, the BM algorithm) are obtained. The choice of which weakening to use is frequently a tradeoff between larger shifts (resulting in a more efficient algorithm) and the greater cost of precomputation and storage of the resulting shift tables.

The idea of range predicate weakening turns out to be very useful, and it will also be used in Section 4.5 (to derive the Boyer-Moore family of algorithms) and in Chapter 5 (to derive a generalization of the Commentz-Walter algorithm).

In order to compute a safe shift distance, some additional information will be used. An interesting side-effect of introducing the reverse trie (creating Algorithm 4.18) is that the predicate $u = lv \wedge v \in \mathbf{suff}(P)$ becomes an invariant of the inner repetition (see Forward reference 4.19). Adding $l, v := \varepsilon, \varepsilon$ to the initial assignments in Algorithm 4.18 turns $u = lv \wedge v \in \mathbf{suff}(P)$ into an invariant of the outer repetition too. This additional information (the invariant) will be used in finding weakenings of the ideal shift predicate. Most of the weakenings that we will derive depend only upon l and v; in Section 4.4.8, we will consider a shift that depends upon l, v, and r.

Notation 4.92 (Shift distance k): Due to this dependence on l, v and perhaps r, we can view k as a function and write $k(l, v, r)$ instead of k. In cases where the shift does not depend upon r, we simply write $k(l, v)$. □

This yields the following algorithm scheme for all variants of the CW algorithm from which variants are obtained by substituting a particular function for $k(l, v, r)$.

Algorithm 4.93 (P$_+$, S$_+$, RT, CW):

$u, r := \varepsilon, S;$
$l, v := \varepsilon, \varepsilon;\ O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
$\{$ invariant: $u = lv \wedge v \in \mathbf{suff}(P)$ $\}$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\rceil k(l, v, r)), r\rfloor k(l, v, r);$
  $l, v := u, \varepsilon;\ O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\};$
  $\{$ invariant: $u = lv \wedge v \in \mathbf{suff}(P)$ $\}$

$$\begin{array}{l} \quad\quad \textbf{do}\ l \neq \varepsilon\ \textbf{cand}\ \tau_r(v, l{\upharpoonright}1) \neq \bot \rightarrow \\ \quad\quad\quad l, v := l{\downharpoonleft}1, (l{\upharpoonright}1)v; \\ \quad\quad\quad O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\} \\ \quad\quad \textbf{od} \\ \quad \textbf{od}\{\ PM\ \} \end{array}$$

□

Previous presentations of the Commentz-Walter algorithms [Com79a, Com79b, WZ92] all present phase shifted versions of this algorithm scheme. In those papers, the phase shifted version is chosen to simplify some of the definitions, at the expense of complicating the algorithm. The algorithm above is considerably simpler. Other algorithm skeletons are possible, as in [FS93] where a single repetition containing an **if**-**fi** construct is used.

We start the weakening of predicate $\mathbf{suff}(u(r{\upharpoonright}n)) \cap P \neq \emptyset$ with some general steps (before proceeding to more specific weakenings) under the assumption that $u = lv \wedge v \in \mathbf{suff}(P) \wedge 1 \leq n \leq |r|$:

$$\begin{array}{ll} & \mathbf{suff}(u(r{\upharpoonright}n)) \cap P \neq \emptyset \\ \equiv & \quad \{\, u = lv \,\} \\ & \mathbf{suff}(lv(r{\upharpoonright}n)) \cap P \neq \emptyset \\ \Rightarrow & \quad \{\, l = (l{\downharpoonleft}1)(l{\upharpoonright}1),\ l{\downharpoonleft}1 \in V^*, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\ & \mathbf{suff}(V^*(l{\upharpoonright}1)v(r{\upharpoonright}n)) \cap P \neq \emptyset \\ \equiv & \quad \{\, 1 \leq n,\ r{\upharpoonright}n = ((r{\upharpoonright}n){\upharpoonright}1)((r{\upharpoonright}n){\downharpoonright}1) = (r{\upharpoonright}1)((r{\upharpoonright}n){\downharpoonright}1) \,\} \\ & \mathbf{suff}(V^*(l{\upharpoonright}1)v(r{\upharpoonright}1)((r{\upharpoonright}n){\downharpoonright}1)) \cap P \neq \emptyset \\ \Rightarrow & \quad \{\, n \leq |r|,\ ((r{\upharpoonright}n){\downharpoonright}1) \in V^{n-1}, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\ & \mathbf{suff}(V^*(l{\upharpoonright}1)v(r{\upharpoonright}1)V^{n-1}) \cap P \neq \emptyset \\ \equiv & \quad \{\, \text{Property 2.59} \,\} \\ & V^*(l{\upharpoonright}1)v(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \end{array}$$

The only reference to r in the last predicate is $r{\upharpoonright}1$. Since $r \neq \varepsilon$ (by the outer repetition guard in Algorithm 4.93) we have $r{\upharpoonright}1 \in V$. We no longer need the upper bound $n \leq |r|$ on n, and it can be dropped from the range of the **MIN** quantification for the shift distance.

String[5] $l{\upharpoonright}1$ is known as the *left lookahead symbol*, v is known as the *recognized suffix* (since $v \in \mathbf{suff}(P)$, by the invariant), and $r{\upharpoonright}1$ is known as the *right lookahead symbol*.

Remark 4.94: In the above derivation, we discarded all but a single symbol of l and r (except in the case $l = \varepsilon$, where we discard all of l). We could have kept more of either string in our weakening (yielding a stronger predicate, and therefore a greater shift distance); unfortunately, this would have given a funtion that is more difficult to precompute, and a shift tables that require more space for storage. For example, we could have kept two symbols of l and two of r, yielding a minimum storage requirement of $\mathcal{O}(|V|^4 \cdot |\mathbf{suff}(P)|)$. The $|\mathbf{suff}(P)|$ term comes from the fact that $v \in \mathbf{suff}(P)$. □

[5] It is a string since it is possible that $l = \varepsilon$.

Forward reference 4.95: In the following section, we will consider a further weakening of the last predicate in the preceding derivation:

$$V^*(l\restriction 1)v(r\upharpoonright 1)V^{n-1} \cap V^*P \neq \emptyset$$

□

In the next section, we consider some general weakening strategies which will be used later to derive particular weakenings. By discussing the strategies first, we are able to develop an informal notation in which we can express the steps in obtaining a weakening, and can be used to develop new weakenings.

4.4.1.1 General weakening strategies

We present a number of weakening strategies, assigning each of them a name (which resembles an algorithm detail). By naming the strategies, we will be able to give a concise description of the steps involved in deriving the weakening. In this dissertation, we will only consider a few of the possible weakenings. The notation introduced in this section is intended to help other weakening-developers convey the general steps involved in deriving their weakenings. It is possible that two different sequences of strategies yield equivalent predicates.

Each of the following strategies is given simply as an implication (or an equivalence), or as a relationship between two **MIN** quantifications. In some cases, a reference to a relevant property is given. In the following descriptions, we assume that $A, B, C \subseteq V^*$ (they are languages), $u, v \in V^*$, $a \in V$, and J and J' are predicates:

DISCARD Discard a conjunct: $J \wedge J' \Rightarrow J$.

DUPLICATE Duplicate a conjunct: $J \equiv J \wedge J$.

SPLIT From Property 2.60:

$$V^*A \cap V^*B \neq \emptyset \equiv V^*A \cap B \neq \emptyset \vee V^*B \cap A \neq \emptyset$$

and

$$V^*aA \cap V^*B \neq \emptyset \equiv V^*aA \cap B \neq \emptyset \vee V^*B \cap A \neq \emptyset$$

DECOUPLE $AuB \cap C \neq \emptyset \Rightarrow AV^{|u|}B \cap C \neq \emptyset$

ABSORB $V^*CA \cap B \neq \emptyset \Rightarrow V^*A \cap B \neq \emptyset$

Q-SPLIT $(\mathbf{MIN}\ i : J(i) \vee J'(i) : i) = (\mathbf{MIN}\ i : J(i) : i)\,\mathbf{min}(\mathbf{MIN}\ i : J'(i) : i)$ (see Property 2.21).

Q-DECOUPLE $(\mathbf{MIN}\ i : J(i) \wedge J'(i) : i) \geq (\mathbf{MIN}\ i : J(i) : i)\,\mathbf{max}(\mathbf{MIN}\ i : J'(i) : i)$ (see Property 2.21).

ENLARGE For $1 \leq k$, $(\mathbf{MIN}\ i : k \leq i \wedge J(i) : i) \geq (\mathbf{MIN}\ i : 1 \leq i \wedge J(i) : i)$

Some of these strategies are used directly in weakening the predicate, while some are used after a weakening has been inserted into a **MIN** quantification. In some of the strategies, it is necessary to specify which conjunct, word, or quantification is being manipulated. An example of this is given below.

Beginning with Forward reference 4.95, we make our first weakening step — a DECOUPLE step. The step can also be given as a hint in the derivation:

$$\begin{array}{ll} & V^*(l{\upharpoonright}1)v(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \\ \Rightarrow & \quad \{\ \text{DECOUPLE } r{\upharpoonright}1\ \} \\ & V^*(l{\upharpoonright}1)vVV^{n-1} \cap V^*P \neq \emptyset \\ \equiv & \quad \{\ VV^{n-1} = V^n\ \} \\ & V^*(l{\upharpoonright}1)vV^n \cap V^*P \neq \emptyset \end{array}$$

Note that the last step involved straightforward manipulation instead of the application of a strategy. Such simple steps are not mentioned in the list of strategies applied.

The last predicate is totally free of r. In Section 4.4.8 we will consider an algorithm which makes use of the right lookahead symbol $r{\upharpoonright}1$.

Forward reference 4.96: In Sections 4.4.2–4.4.6 we will consider further weakenings of the predicate (derived above):

$$V^*(l{\upharpoonright}1)vV^n \cap V^*P \neq \emptyset$$

□

In the following section, we consider the $l = \varepsilon$ case (and the no-lookahead case) separately. This will allow us to assume $l \neq \varepsilon$ in Sections 4.4.2–4.4.8.

4.4.1.2 The $l = \varepsilon$ and the no-lookahead cases

In the $l = \varepsilon$ case, the predicate in Forward reference 4.96 is equivalent to $V^*vV^n \cap V^*P \neq \emptyset$. We apply a SPLIT step, yielding

$$V^*vV^n \cap P \neq \emptyset \vee V^*P \cap vV^n \neq \emptyset$$

We can use this predicate, and one more step, to arrive at a practical shift distance. Starting with the ideal shift distance:

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r{\upharpoonright}n)) \cap P \neq \emptyset : n) \\ \geq & \quad \{\ \text{weakening steps above}\ \} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge (V^*vV^n \cap P \neq \emptyset \vee V^*P \cap vV^n \neq \emptyset) : n) \\ = & \quad \{\ \text{Q-SPLIT}\ \} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*vV^n \cap P \neq \emptyset : n)\,\mathbf{min}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*P \cap vV^n \neq \emptyset : n) \end{array}$$

In order to make this shift distance more concise, we define two auxiliary functions.

Definition 4.97 (Functions d_1 and d_2): Functions $d_1, d_2 \in \textbf{suff}(P) \longrightarrow \mathbb{N}$ are defined by

$$\begin{array}{rcl} d_1(x) & = & (\textbf{MIN}\ n : 1 \leq n \wedge V^*xV^n \cap P \neq \emptyset : n) \\ d_2(x) & = & (\textbf{MIN}\ n : 1 \leq n \wedge V^*P \cap xV^n \neq \emptyset : n) \end{array}$$

□

Functions d_1 and d_2 were named by Commentz-Walter [Com79a, Com79b].

Remark 4.98: Note that the quantification in d_1 can have an empty range, meaning that it can take value $+\infty$. On the other hand, the quantification in the definition of d_2 never has an empty range, and therefore d_2 never takes value $+\infty$. Indeed, function d_2 is bounded above by $(\textbf{MIN}\ p : p \in P : |p|)\ \textbf{max}\ 1$, and so the shift distances are never greater than $(\textbf{MIN}\ p : p \in P : |p|)\ \textbf{max}\ 1$. In Chapter 13, we show that this upperbound can have a significant effect on the practical performance of the Commentz-Walter algorithm variants. □

Forward reference 4.99: Functions d_1 and d_2 will also be used in Chapter 5 to derive a generalization of the Commentz-Walter algorithm. In that chapter, precomputation algorithms for d_1 and d_2 are also presented. □

Example 4.100 (Functions d_1 and d_2): For keyword set $\{her, his, she\}$ we compute d_1 and d_2:

w	ε	e	r	s	er	he	is	her	his	she
$d_1(w)$	1	1	$+\infty$	2	$+\infty$	1	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$d_2(w)$	3	3	3	2	3	1	2	3	2	1

□

Using functions d_1 and d_2, our $l = \varepsilon$ shift distance is $d_1(v)\ \textbf{min}\ d_2(v)$. In the following sections, we assume $l \neq \varepsilon$.

4.4.1.2.1 The no-lookahead shift function We can discard all references to the left lookahead symbol, by using the $l = \varepsilon$ shift distance for the $l \neq \varepsilon$ case too. This weakening step is referred to as discarding the lookahead symbol. The corresponding shift function is given as follows.

Definition 4.101 (Shift function k_{nla}): Shift function k_{nla} is defined as:

$$k_{nla}(l, v) = d_1(v)\ \textbf{min}\ d_2(v)$$

□

Remark 4.102: This shift function yields the smallest shift distances of all shift functions to be considered in this section. □

Since the various k shift functions are usually expressed in terms of more elementary functions, they are not usually tabulated (the k functions are computed on-the-fly). The basic functions, however, are usually tabulated.

Algorithm detail 4.103 (NLA)**:** Calculating the shift distance using k_{nla} is algorithm detail (NLA). □

The use of shift function k_{nla} results in algorithm (P$_+$, S$_+$, RT, CW, NLA), which does not appear in the literature. This algorithm is of no great practical interest, since the precomputation is barely cheaper than any of the other variants (for example, shift function k_{cw} — see Section 4.4.5), and the resulting shift distances are less than in any of the other variants. The algorithm does, however, use only $\mathcal{O}(|\mathbf{suff}(P)|)$ storage for functions d_1 and d_2.

4.4.2 A shift function without further weakening

For our first shift function, we do not weaken the predicate in Forward reference 4.96 any further — we simply apply SPLIT, and Q-SPLIT. This weakening, and most of the following ones, will include the left lookahead symbol (one of the weakenings will not). The use of the lookahead symbol is given in the following algorithm detail

Algorithm detail 4.104 (LLA)**:** The left lookahead symbol ($l{\restriction}1$) is used in determining the shift distance. □

Applying SPLIT to the predicate in Forward reference 4.96 yields:

$$V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \vee V^*P \cap vV^n \neq \emptyset$$

We manipulate the resulting **MIN** quantification as follows:

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \leq n \wedge (V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \vee V^*P \cap vV^n \neq \emptyset) : n) \\ = & \quad \{\ \text{Q-SPLIT}\ \} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\restriction}1)vV^n \cap P \neq \emptyset : n) \\ & \ \mathbf{min}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*P \cap vV^n \neq \emptyset : n) \end{array}$$

The second term of the infix **min** is simply $d_2(v)$. To give the shift function concisely, we define another auxiliary function.

Definition 4.105 (Function d_{opt}): Functions $d_{opt} \in V \times \mathbf{suff}(P) \longrightarrow \mathbb{N}$ is defined by

$$d_{opt}(a, x) = (\mathbf{MIN}\ n : 1 \leq n \wedge V^*axV^n \cap P \neq \emptyset : n)$$

□

Remark 4.106: Note that the quantification in the definition of this function can have an empty range, and therefore d_{opt} can take value $+\infty$. □

Due to the size of the resulting tables, we do not present an example of d_{opt} here.
Given this function, we can define the shift distance:

Definition 4.107 (Shift function k_{opt}): Shift function k_{opt} is defined as

$$k_{opt}(l,v) = \begin{cases} d_{opt}(l{\upharpoonright}1, v) \min d_2(v) & l \neq \varepsilon \\ d_1(v) \min d_2(v) & l = \varepsilon \end{cases}$$

□

The use of the particular shift derived above is given in the following algorithm detail.

Algorithm detail 4.108 (CW-OPT)**:** Calculating the shift distance using function k_{opt} is algorithm detail (CW-OPT). □

The resulting algorithm is (P$_+$, S$_+$, RT, CW, LLA, CW-OPT). From the informal description in their article, it appears that Fan and Su present a version of this algorithm [FS93]. The algorithm was derived independently in [WZ95], where the precomputation of the three auxiliary functions can also be found. This algorithm promises to be particularly efficient, although it is not one of the ones benchmarked in Chapter 13. The disadvantage to the use of shift function k_{opt} is that it requires storage $\mathcal{O}(|V| \cdot |\mathbf{suff}(P)|)$, whereas some of the other shift functions presented in this chapter require less storage.

4.4.3 Towards the CW and BM algorithms

In this section, we weaken the range predicate from function d_{opt} further, decoupling the lookahead symbol and the recognized suffix. We perform a type of DECOUPLE step, using following function:

Definition 4.109 (Function MS): Function $MS \in \mathbf{suff}(P) \longrightarrow \mathcal{P}(V)$ is defined as

$$MS(v) = \{\, a \mid av \in \mathbf{suff}(P) \,\}$$

□

Example 4.110 (Function MS): Computing MS for keyword set $\{her, his, she\}$ yields:

w	ε	e	r	s	er	he	is	her	his	she
$MS(w)$	$\{e,r,s\}$	$\{h\}$	$\{e\}$	$\{i\}$	$\{h\}$	$\{s\}$	$\{h\}$	$\emptyset$	$\emptyset$	$\emptyset$

□

We will need a conjunct of the postcondition of the inner repetition of Algorithm 4.93: $(l{\upharpoonright}1)v \notin \mathbf{suff}(P)$; this follows from the negation of the inner repetition guard. Recall that we are considering the $l \neq \varepsilon$ case, so we may assume the termination condition of the inner repetition. It follows that $(l{\upharpoonright}1) \notin MS(v)$, equivalently $(l{\upharpoonright}1) \in V \setminus MS(v)$. We begin with the range predicate of function d_{opt}:

$$
\begin{array}{ll}
 & V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \\
\equiv & \qquad \{ \text{DUPLICATE} \} \\
 & V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \wedge V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \\
\equiv & \qquad \{ \text{DECOUPLE } v \text{ in first conjunct} \} \\
 & V^*(l{\restriction}1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*(l{\restriction}1)vV^n \cap P \neq \emptyset \\
\Rightarrow & \qquad \{ \text{definition of } MS;\ (l{\restriction}1)v \notin \mathbf{suff}(P) \} \\
 & V^*(l{\restriction}1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset
\end{array}
$$

In the following sections, we will further manipulate the shift predicate in the last line above. We can directly use the above predicate, by defining the following shift function.

Definition 4.111 (Function d_{bmcw}): Define $d_{bmcw} \in V \times \mathbf{suff}(P) \longrightarrow \mathbb{N}$ by $d_{bmcw}(a, x) =$

$$(\mathbf{MIN}\ n : 1 \leq n \wedge V^*aV^{|x|+n} \cap P \neq \emptyset \wedge V^*(V \setminus MS(x))xV^n \cap P \neq \emptyset : n)$$

□

Example 4.112 (Function d_{bmcw}): Due to the space required for the table, an example of this function is not given here. □

Remark 4.113: Note that the quantification in the definition of function d_{bmcw} can have an empty range, meaning that it can take value $+\infty$. □

The resulting shift distance is given as follows.

Definition 4.114 (Shift function k_{bmcw}): Shift function k_{bmcw} is defined as

$$k_{bmcw}(l, v) = \begin{cases} d_{bmcw}(l{\restriction}1, v) \mathbin{\mathbf{min}} d_2(v) & l \neq \varepsilon \\ d_1(v) \mathbin{\mathbf{min}} d_2(v) & l = \varepsilon \end{cases}$$

□

Remark 4.115: The shift distance given by k_{bmcw} is never greater than that given by k_{opt}. That is, $k_{bmcw} \leq k_{opt}$. □

Using this shift distance is given in the following algorithm detail.

Algorithm detail 4.116 (BMCW): Calculating the shift distance using function k_{bmcw} is algorithm detail (BMCW). □

The resulting algorithm (P$_+$, S$_+$, RT, CW, LLA, CW-OPT, BMCW) does not appear in the literature. The algorithm includes the (CW-OPT) detail, since shift distance is derived from the one given in detail (CW-OPT). It is given in [WZ95], where the precomputation of the auxiliary function is discussed. Shift function k_{bmcw} requires the same amount of storage as k_{opt}. Function k_{bmcw} is interesting because it combines the best of the Boyer-Moore and the normal Commentz-Walter algorithms (both to be presented later). An algorithm given by Baeza-Yates and Régnier (in [B-YR90]) appears to be related to this one. Although it is not yet clear how their shift distance is obtained, it appears that it yields smaller shifts than those given with Algorithm detail (BMCW).

4.4.4 A more easily precomputed shift function

In this section, we weaken the range predicate derived in the previous section, applying an ABSORB step. Starting from the predicate in the previous section, we derive:

$$
\begin{array}{ll}
 & V^*(l{\upharpoonright}1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset \\
\Rightarrow & \quad \{\ \text{ABSORB in second conjunct}\ \} \\
 & V^*(l{\upharpoonright}1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*vV^n \cap P \neq \emptyset
\end{array}
$$

We can now define another auxiliary function, using this predicate in its range.

Definition 4.117 (Function d_{nopt}): Function $d_{nopt} \in V \times \mathbf{suff}(P) \longrightarrow \mathbb{N}$ is defined by

$$d_{nopt}(a, x) = (\mathbf{MIN}\ n : 1 \leq n \wedge V^*aV^{|x|+n} \cap P \neq \emptyset \wedge V^*xV^n \cap P \neq \emptyset : n)$$

□

Remark 4.118: As with function d_{opt}, the quantification in function d_{nopt} can have an empty range, and the function can take value $+\infty$. □

Due to the size of the resulting tables, we do not present an example of function d_{nopt} here. Given d_{nopt}, we can define the shift distance:

Definition 4.119 (Shift function k_{nopt}): Shift function k_{nopt} is defined as

$$k_{nopt}(l, v) = \begin{cases} d_{nopt}(l{\upharpoonright}1, v)\ \mathbf{min}\ d_2(v) & l \neq \varepsilon \\ d_1(v)\ \mathbf{min}\ d_2(v) & l = \varepsilon \end{cases}$$

□

Remark 4.120: Note that (due to the weakenings of the range predicate) the shift distance given by k_{nopt} is never greater than that given by k_{bmcw}. That is, $k_{nopt} \leq k_{bmcw}$. □

The use of the particular shift derived above is given in the following algorithm detail.

Algorithm detail 4.121 (NEAR-OPT): Calculating the shift distance using function k_{nopt} is algorithm detail (NEAR-OPT). □

The resulting algorithm (P$_+$, S$_+$, RT, CW, LLA, CW-OPT, BMCW, NEAR-OPT) does not appear in the literature. Note that the sequence of details includes the sequence of details from the previous section, since k_{nopt} derived from k_{bmcw}. The storage requirements for this shift function are the same as the requirements for k_{opt}. The advantage of using k_{nopt} is that the precomputation is cheaper.

4.4.5 The standard Commentz-Walter algorithm

We can further weaken the range predicate used in the previous section in the definition of function d_{nopt}. We apply Q-DECOUPLE, followed by ENLARGE. We start our derivation with d_{nopt}.

$$\begin{array}{ll}
& d_{nopt}(l\upharpoonright 1, v) \\
= & \quad \{\text{ definition of } d_{nopt}\ \} \\
& (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l\upharpoonright 1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*vV^n \cap P \neq \emptyset : n) \\
\geq & \quad \{\ \text{Q-DECOUPLE}\ \} \\
& (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l\upharpoonright 1)V^{|v|+n} \cap P \neq \emptyset : n) \\
& \mathbf{max}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*vV^n \cap P \neq \emptyset : n) \\
= & \quad \{\text{ definition of function } d_1\ \} \\
& (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l\upharpoonright 1)V^{|v|+n} \cap P \neq \emptyset : n)\,\mathbf{max}\,d_1(v) \\
= & \quad \{\text{ changing bound variable: } n' = |v| + n\ \} \\
& (\mathbf{MIN}\ n' : 1 + |v| \leq n' \wedge V^*(l\upharpoonright 1)V^{n'} \cap P \neq \emptyset : n' - |v|)\,\mathbf{max}\,d_1(v) \\
\geq & \quad \{\ \text{ENLARGE}\ \} \\
& (\mathbf{MIN}\ n' : 1 \leq n' \wedge V^*(l\upharpoonright 1)V^{n'} \cap P \neq \emptyset : n' - |v|)\,\mathbf{max}\,d_1(v)
\end{array}$$

We use the following auxiliary function to simplify the **MIN** quantification in the last expression.

Definition 4.122 (Function $char_{cw}$): Define $char_{cw} \in \mathbb{N} \times V \longrightarrow \mathbb{N}$ by

$$char_{cw}(i, a) = (\mathbf{MIN}\ n : 1 \leq n \wedge V^*aV^n \cap P \neq \emptyset : n - i)$$

Note that we could have given a more specific signature, since the first argument to $char_{cw}$ is always in the range $[0, (\mathbf{MAX}\ p : p \in P : |p|)]$. □

Example 4.123 (Function $char_{cw}$): Due to the space required for the tables, an example of $char_{cw}$ is not given here. The interested reader can easily construct an example of $char_{cw}$ using Example 4.130 and Property 4.131. □

Remark 4.124: Note that it is possible for the quantification in the definition of $char_{cw}$ to have an empty range, and therefore function $char_{cw}$ can take value $+\infty$. □

We can now give the standard Commentz-Walter shift function.

Definition 4.125 (Shift function k_{cw}): Shift function k_{cw} is defined as

$$k_{cw}(l, v) = \begin{cases} (char_{cw}(|v|, l\upharpoonright 1)\,\mathbf{max}\,d_1(v))\,\mathbf{min}\,d_2(v) & l \neq \varepsilon \\ d_1(v)\,\mathbf{min}\,d_2(v) & l = \varepsilon \end{cases}$$

□

Remark 4.126: Note that (due to the weakenings of the range predicate) the shift distance given by k_{cw} is never greater than that given by k_{nopt}. That is, $k_{cw} \leq k_{nopt}$. □

The particular shift derived above is given in the following algorithm detail.

Algorithm detail 4.127 (NORM): Calculating the shift distance using function k_{cw} is algorithm detail (NORM). □

The resulting algorithm (P_+, S_+, RT, CW, LLA, CW-OPT, BMCW, NEAR-OPT, NORM) is the normal Commentz-Walter algorithm (cf. [Com79a, Section II] and [Com79b, Sections II.1 and II.2]). The storage requirements for this shift function are $\mathcal{O}(|\mathbf{suff}(P)|)$ for functions d_1 and d_2, and $\mathcal{O}((\mathbf{MAX}\ p : p \in P : |p|) \cdot |V|)$ for $char_{cw}$. (The definition of function $char_{cw}$ makes it obvious that it can be stored in $\mathcal{O}(|V|)$ space with a small penalty for computing the function.) As a result, k_{cw} can be stored more economically than k_{opt}, k_{nopt}, or k_{nopt}. The precomputation required for k_{cw} is also cheaper, the tradeoff being a smaller shift distance.

4.4.6 A derivation of the Boyer-Moore algorithm

In this section, we derive the multiple keyword Boyer-Moore algorithm starting with the weakening in Section 4.4.3. We begin our derivation with d_{bmcw}:

$$
\begin{array}{ll}
 & d_{bmcw}(l{\upharpoonright}1, v) \\
= & \qquad \{ \text{definition of } d_{bmcw} \} \\
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\upharpoonright}1)V^{|v|+n} \cap P \neq \emptyset \wedge V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset : n) \\
\geq & \qquad \{ \text{Q-DECOUPLE} \} \\
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\upharpoonright}1)V^{|v|+n} \cap P \neq \emptyset : n) \\
 & \mathbf{max}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset : n)
\end{array}
$$

We continue our derivation with the the first operand of the infix **max**:

$$
\begin{array}{ll}
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\upharpoonright}1)V^{|v|+n} \cap P \neq \emptyset : n) \\
= & \qquad \{ \text{changing bound variable: } n' = n + v \} \\
 & (\mathbf{MIN}\ n' : 1 + |v| \leq n' \wedge V^*(l{\upharpoonright}1)V^{n'} \cap P \neq \emptyset : n' - |v|) \\
\geq & \qquad \{ \text{ENLARGE} \} \\
 & (\mathbf{MIN}\ n' : 1 \leq n' \wedge V^*(l{\upharpoonright}1)V^{n'} \cap P \neq \emptyset : n' - |v|) \\
\geq & \qquad \{ V^*(l{\upharpoonright}1)V^n \cap P \neq \emptyset \Rightarrow V^*(l{\upharpoonright}1)V^n \cap V^*P \neq \emptyset \} \\
 & (\mathbf{MIN}\ n' : 1 \leq n' \wedge V^*(l{\upharpoonright}1)V^{n'} \cap V^*P \neq \emptyset : n' - |v|) \\
= & \qquad \{ \text{non-empty range predicate} \} \\
 & ((\mathbf{MIN}\ n' : 1 \leq n' \wedge V^*(l{\upharpoonright}1)V^{n'} \cap V^*P \neq \emptyset : n') - |v|)
\end{array}
$$

To present the resulting shift distance concisely, we define the following auxiliary functions.

Definition 4.128 (Function $char_{bm}$): Function $char_{bm} \in V \longrightarrow \mathbb{N}$ is defined as

$$char_{bm}(a) = (\mathbf{MIN}\ n : 1 \leq n \wedge V^*aV^n \cap V^*P \neq \emptyset : n)$$

□

Definition 4.129 (Function d_{bm}): Define $d_{bm} \in \mathbf{suff}(P) \longrightarrow \mathbb{N}$ by

$$d_{bm}(x) = (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(V \setminus MS(x))xV^n \cap P \neq \emptyset : n)$$

□

In [WZ95], function d_{bm} is called d_{vi}.

Example 4.130 (Function $char_{bm}$): Recall that we take our alphabet to be $\{e, h, i, r, s, y\}$.

a	e	h	i	r	s	y
$char_{bm}(a)$	1	1	1	3	2	3

□

Functions $char_{cw}$ and $char_{bm}$ are related by the following interesting property.

Property 4.131 (Functions $char_{cw}$ and $char_{bm}$): Note that:

$$char_{bm}(a) = char_{cw}(0, a)\,\mathbf{min}(\mathbf{MIN}\ p : p \in P : |p|)\,\mathbf{min}\,1$$

From this property, and the above example, it is possible to construct an example of $char_{cw}$.
□

Example 4.132 (Function d_{bm}): Computing d_{bm} for keyword set $\{her, his, she\}$ yields:

w	ε	e	r	s	er	he	is	her	his	she
$d_{bm}(w)$	1	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

□

Definition 4.133 (BM shift function k_{bm}): The BM shift function k_{bm} is defined as:

$$k_{bm}(l, v) = \begin{cases} ((char_{bm}(l{\upharpoonright}1) - |v|)\,\mathbf{max}\,d_{bm}(v))\,\mathbf{min}\,d_2(v) & l \neq \varepsilon \\ d_1(v)\,\mathbf{min}\,d_2(v) & l = \varepsilon \end{cases}$$

□

Remark 4.134: The shift distance given by k_{bm} is never greater than that given by k_{bmcw}. That is, $k_{bm} \leq k_{bmcw}$. The shift distance given by k_{bm} is not comparable to that given by functions k_{nopt} or k_{cw}. □

Algorithm detail 4.135 (BM): Calculating the shift distance using function k_{bm} is algorithm detail (BM). □

The resulting algorithm is the Boyer-Moore algorithm (P$_+$, S$_+$, RT, CW, LLA, CW-OPT, BMCW, BM). Adding problem detail (OKW) (restricting P to one keyword) yields the well-known BM algorithm, (P$_+$, S$_+$, RT, CW, LLA, CW-OPT, BMCW, BM, OKW), which appears in the literature as [BM77, Section 4]. Precomputation of functions d_{bm} and $char_{bm}$ is discussed in [WZ95].

4.4.7 A weakened Boyer-Moore algorithm

Finally, we derive a weaker variant of the BM algorithm. This variant incorporates the weakenings involved in both the normal Commentz-Walter and the Boyer-Moore algorithms. The resulting shift function is given as follows:

Definition 4.136 (Weak BM shift function k_{wbm}): The BM shift function k_{wbm} is defined as:

$$k_{wbm}(l,v) = \begin{cases} ((char_{bm}(l{\upharpoonright}1) - |v|) \mathbf{max}\, d_1(v)) \mathbf{min}\, d_2(v) & l \neq \varepsilon \\ d_1(v) \mathbf{min}\, d_2(v) & l = \varepsilon \end{cases}$$

□

We do not introduce a new detail, since this algorithm combines details NEAR-OPT, NORM, and BM.

The resulting algorithm is a weak Boyer-Moore algorithm (P$_+$, S$_+$, RT, CW, LLA, CW-OPT, BMCW, NEAR-OPT, NORM, BM). (Since there are two root-paths to this algorithm, the last three algorithm details could also have been ordered as BM, NEAR-OPT, NORM.) This shift function yields a shift distance which is no greater than that given by the Commentz-Walter and Boyer-Moore shift functions.

4.4.8 Using the right lookahead symbol

In this section, we consider a shift function which uses the right lookahead symbol. Since we use the symbol, we introduce the following algorithm detail.

Algorithm detail 4.137 (RLA): The right lookahead symbol $(r{\restriction}1)$ is used in determining the shift distance. □

We shall begin with the predicate given in Forward reference 4.95,

$$V^*(l{\upharpoonright}1)v(r{\restriction}1)V^{n-1} \cap V^*P \neq \emptyset$$

We will apply the following steps:

1. DUPLICATE.
2. DECOUPLE $r{\restriction}1$ and $(l{\upharpoonright}1)v$.
3. ABSORB.
4. Q-DECOUPLE.
5. Implicit SPLIT and Q-SPLIT.

This is shown in the following derivation:

$$
\begin{array}{ll}
 & V^*(l{\upharpoonleft}1)v(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \\
\equiv & \quad \{\ \textsc{duplicate}\ \} \\
 & V^*(l{\upharpoonleft}1)v(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \wedge V^*(l{\upharpoonleft}1)v(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \\
\Rightarrow & \quad \{\ \textsc{decouple}\ r{\upharpoonright}1 \text{ (first conjunct) and } (l{\upharpoonleft}1)v \text{ (second conjunct)}\ \} \\
 & V^*(l{\upharpoonleft}1)vVV^{n-1} \cap V^*P \neq \emptyset \wedge V^*VV^{|v|}(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset \\
\Rightarrow & \quad \{\ \textsc{absorb}\ \} \\
 & V^*(l{\upharpoonleft}1)vV^n \cap V^*P \neq \emptyset \wedge V^*(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset
\end{array}
$$

We can now use the last predicate in a **MIN** quantification:

$$
\begin{array}{ll}
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\upharpoonleft}1)vV^n \cap V^*P \neq \emptyset \wedge V^*(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset : n) \\
\geq & \quad \{\ \textsc{q-decouple}\ \} \\
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*(l{\upharpoonleft}1)vV^n \cap V^*P \neq \emptyset : n) \\
 & \ \mathbf{max}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset : n) \\
= & \quad \{\ \text{Section 4.4.2 — definitions of } d_{opt}, d_2\text{; implicit } \textsc{split}, \textsc{q-split}\text{; } l \neq \varepsilon\ \} \\
 & (d_{opt}(l{\upharpoonleft}1, v)\,\mathbf{min}\,d_2(v))\,\mathbf{max}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*(r{\upharpoonright}1)V^{n-1} \cap V^*P \neq \emptyset : n) \\
= & \quad \{\ \text{changing of bound variable: } n' = n - 1\ \} \\
 & (d_{opt}(l{\upharpoonleft}1, v)\,\mathbf{min}\,d_2(v))\,\mathbf{max}(\mathbf{MIN}\ n' : 0 \leq n' \wedge V^*(r{\upharpoonright}1)V^{n'} \cap V^*P \neq \emptyset : n' + 1) \\
= & \quad \{\ \text{non-empty range predicate}\ \} \\
 & (d_{opt}(l{\upharpoonleft}1, v)\,\mathbf{min}\,d_2(v))\,\mathbf{max}(\mathbf{MIN}\ n' : 0 \leq n' \wedge V^*(r{\upharpoonright}1)V^{n'} \cap V^*P \neq \emptyset : n') + 1
\end{array}
$$

We can now define an auxiliary function.

Definition 4.138 (Function $char_{rla}$): Function $char_{rla} \in V \longrightarrow \mathbb{N}$ is defined by

$$char_{rla}(a) = (\mathbf{MIN}\ n : 0 \leq n \wedge V^*aV^n \cap V^*P \neq \emptyset : n) + 1$$

□

Example 4.139 (Function $char_{rla}$):

a	e	h	i	r	s	y
$char_{rla}(a)$	1	2	2	1	1	4

□

This function can be used in the following shift distance:

Definition 4.140 (Shift function k_{ropt}): The optimized shift function with right lookahead is:

$$k_{ropt}(l, v, r) = \begin{cases} (d_{opt}(l{\upharpoonleft}1, v)\,\mathbf{min}\,d_2(v))\,\mathbf{max}\,char_{rla}(r{\upharpoonright}1) & l \neq \varepsilon \\ d_1(v)\,\mathbf{min}\,d_2(v) & l = \varepsilon \end{cases}$$

□

Algorithm detail 4.141 (R-OPT): Calculating the shift distance using function k_{ropt} is referred to as algorithm detail (R-OPT). □

The resulting algorithm is (P_+, S_+, RT, CW, LLA, RLA, R-OPT). This algorithm does not appear in the literature. It is not difficult to see that function k_{ropt} will always yield a shift distance at least as large as k_{opt}. Function $char_{rla}$ requires $\mathcal{O}(|V|)$ storage. The precomputation of $char_{rla}$ is similar to that for $char_{bm}$ (see [WZ95]).

4.5 The Boyer-Moore family of algorithms

Since the appearance of the original Boyer-Moore algorithm [BM77], many variations and improvements have been published. Most of these have been classified and discussed by Hume and Sunday in [HS91], which also contains extensive benchmarking results. The material we present in this section supplements their work, as we provide derivations and correctness arguments for most of the algorithms in their paper. Figure 4.11 gives the part of the taxonomy graph which corresponds to this section.

The Boyer-Moore algorithm derivation in the previous section only accounted for one method of traversing the string variable u, in increasing order of v. In practice, when $P = \{p\}$ (P is a singleton set) other methods of comparing v to keyword p can be used. We therefore introduce problem detail (OKW) ($P = \{p\}$, originally given on page 75). Starting with the original problem specification, we derive the Boyer-Moore algorithm and its variants. The derivation presented here has a number of similarities with the one given in the previous section, in particular, the technique of predicate weakening (introduced in Section 4.4.1) will again be used to derive shift distances. Different weakening strategies (which will not be introduced explicitly as they were in Section 4.4.1.1) can be used in this section, thanks to problem detail (OKW).

To make the following presentation more readable, we define a 'perfect match' (as opposed to a failed, or partial, match) predicate and an auxiliary function.

Definition 4.142 (Perfect match predicate *PerfMatch*): We define a 'perfect match' predicate

$$PerfMatch((l,v,r)) \equiv (lvr = S \wedge v = p)$$

Notice that p is an implicit parameter of *PerfMatch*. □

We can rewrite the pattern matching postcondition in terms of predicate *PerfMatch* as:

$$O = (\cup\ l,v,r : PerfMatch((l,v,r)) : \{(l,v,r)\})$$

To make the following presentation more readable, we introduce an auxiliary function.

Definition 4.143 (Shift function *shift*): Define right shift function $shift \in (V^*)^3 \times \mathbb{N} \longrightarrow (V^*)^3$ by

$$shift(l,v,r,k) = (l(vr\restriction k), (v(r\restriction k))\lfloor k, r\lfloor k)$$

□

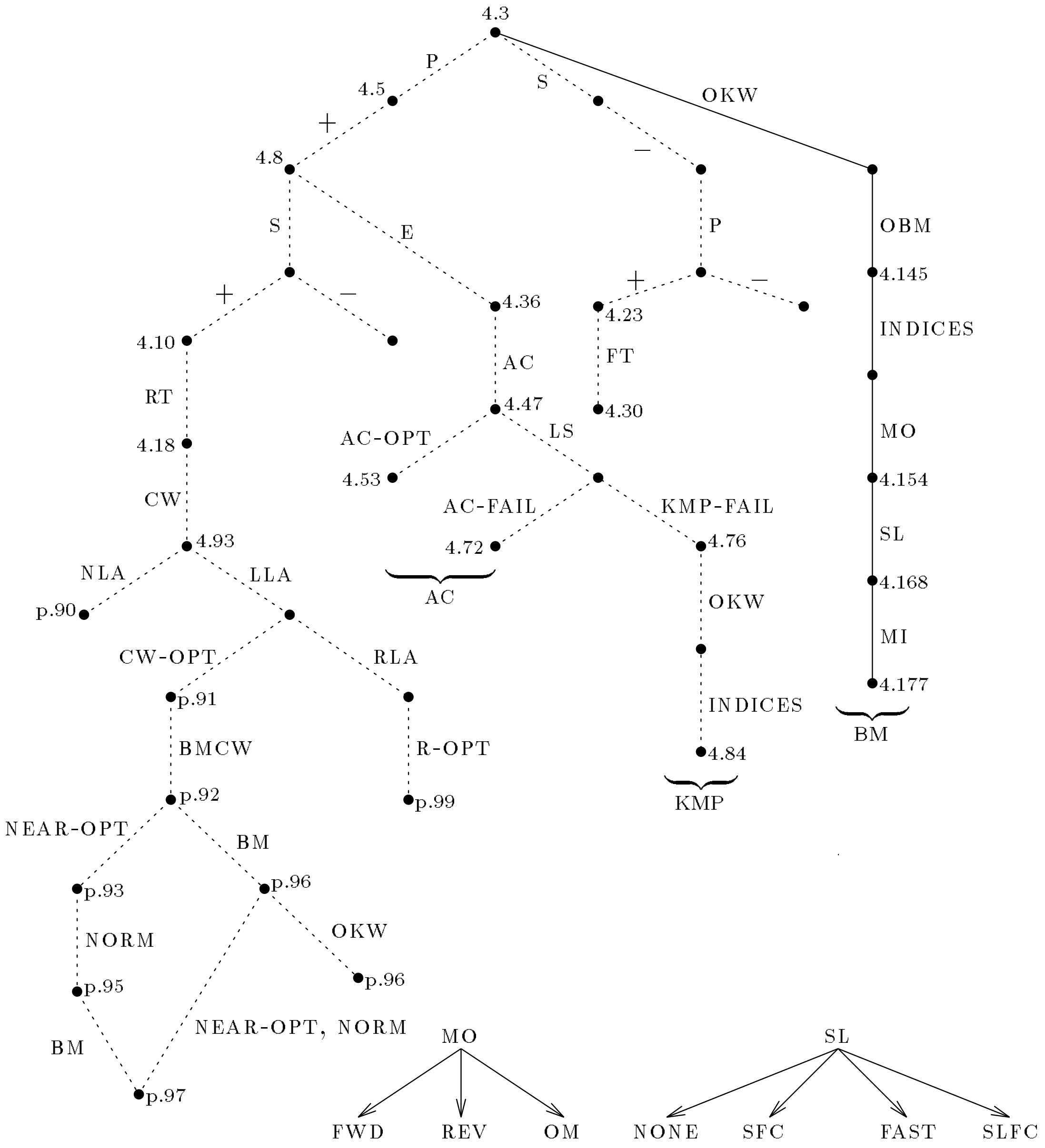


Figure 4.11: The algorithms considered in Section 4.5 are denoted by solid circles, connected by solid lines. Additional algorithm details considered in this section are denoted by the two smaller graphs at the bottom of the figure.

We can now introduce the 'ordinary Boyer-Moore' algorithm detail.

Algorithm detail 4.144 (OBM)**:** Use function *shift* and maintain invariant $I_1(l,v,r) \equiv$

$$\begin{array}{rcl} O & = & (\cup\ l',v',r' : \mathit{PerfMatch}((l',v',r')) \wedge (l'v' <_p lv) : \{(l',v',r')\}) \\ & & \wedge\ (lvr = S) \wedge (|v| \leq |p|) \wedge (|v| < |p| \Rightarrow r = \varepsilon) \end{array}$$

□

Using this algorithm detail, we obtain a first (deterministic) solution.

Algorithm 4.145 (OKW, OBM)**:**

$l,v,r := \varepsilon, S\upharpoonright|p|, S\downharpoonright|p|;\ O := \emptyset;$
$\{$ invariant: $I_1(l,v,r)$ $\}$
do $|v| = |p| \rightarrow$
  if $v = p \rightarrow O := O \cup \{(l,v,r)\}$
  $[\!]$ $v \neq p \rightarrow$ **skip**
  fi;
  $(l,v,r) := \mathit{shift}(l,v,r,1)$
od$\{$ PM $\}$

□

This algorithm does not take into account how we evaluate $v = p$. Comparing symbols of v and p from left to right or right to left are two possible methods of evaluating $v = p$. The methods that we consider in this section all involve indexing in strings v and p. For this reason, we introduce algorithm detail (INDICES) (originally given on page 77).

To take advantage of the indexing, we define 'match orders', which will enable us to consider a number of different ways of comparing v and p.

Definition 4.146 (Match order): A match order is a bijective function $mo \in [1,|p|] \longrightarrow [1,|p|]$. This function is used to determine the order in which the individual symbols of v and p are compared. □

The usefulness of match orders is expressed in the following property.

Property 4.147 (Match order): Since mo is bijective, we now have

$$(v = p) \equiv (\forall\ i : 1 \leq i \leq |p| : v_{mo(i)} = p_{mo(i)})$$

□

The match order algorithm detail is:

Algorithm detail 4.148 (MO)**:** The symbols of v and p are compared in a fixed order determined by a bijective function $mo \in [1,|p|] \longrightarrow [1,|p|]$. □

The particular match order used in an algorithm determines the third position of the algorithm name. Three of the most common match orders, (which represent particular instances of detail (MO) and appear in the smaller graph of Figure 4.11 with (MO) as root) are

Algorithm detail 4.149 (FWD): The forward (or identity) match order given by $mo(i) = i$. □

Algorithm detail 4.150 (REV): The reverse match order given by $mo(i) = |p| - i + 1$. This is the original Boyer-Moore match order. □

Algorithm detail 4.151 (OM): Let $Prob \in [1, |p|] \longrightarrow \mathbb{R}$ be the probability distribution of the symbols of p in input string S; the domain of function $Prob$ consists of indices into p. Let an 'optimal mismatch' match order be any permutation mo such that

$$(\forall\ i, j : 1 \leq i \leq |p| \wedge 1 \leq j \leq i : Prob(mo(j)) \leq Prob(mo(i)))$$

This match order is described as 'optimal' because it compares symbols of p in order of ascending probability of occurring in S. In this way, the least probable symbols of p are compared first, so on the average one can expect to find any mismatch as early as possible. □

Example 4.152 (Match orders): We assume a single keyword *hehshe* (taken from Example 4.80). The (FWD) match order is $mo(i) = i$, while the (REV) match order is $mo(i) = 6 - i + 1 = 7 - i$. If we assume that symbol h is the least probable, followed by s and finally e, we obtain an (OM) match order $mo = \{(1,1), (2,3), (3,5), (4,4), (5,6), (6,2)\}$. Another possible (OM) match order is $mo = \{(1,3), (2,1), (3,5), (4,4), (5,2), (6,6)\}$. □

Comparing v and p using match order mo is done by (program) function *match* specified by

---

$\{\ |v| = |p|\ \}$
$i := match(v, p, mo)$
$\{\ I_2(v, p, mo, i) : (1 \leq i \leq |p| + 1) \wedge (i \leq |p| \Rightarrow v_{mo(i)} \neq p_{mo(i)})$
$\qquad \wedge (\forall\ j : 1 \leq j < i : v_{mo(j)} = p_{mo(j)})\ \}$

Property 4.153 (Postcondition of *match*): From $I_2(v, p, mo, i)$ it follows that $(v = p) \equiv (i = |p| + 1)$, and that if $i \leq |p|$ then $v_{mo(i)}$ is the first (in the given order) mismatching symbol. □

An implementation of *match* is

---

func $match(v, p, mo) \rightarrow$
  $i := 1$;
  do $i \leq |p|$ **cand** $v_{mo(i)} = p_{mo(i)} \rightarrow i := i + 1$ **od**;
  return i
cnuf

Adding indexing, mo, i, and $match$ to Algorithm 4.145(OKW, OBM) results in

Algorithm 4.154 (OKW, OBM, INDICES, MO):

$l, v, r := \varepsilon, S\upharpoonright|p|, S\downharpoonright|p|;\ O := \emptyset$;
{ invariant: $I_1(l, v, r)$ }
do $|v| = |p| \rightarrow$
  $i := match(v, p, mo)$;
  { $I_2(v, p, mo, i)$ }
  if $i = |p| + 1 \rightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $i \neq |p| + 1 \rightarrow$ **skip**
  fi;
  $(l, v, r) := shift(l, v, r, 1)$
od{ PM }

□

Remark 4.155: In some versions of the Boyer-Moore algorithms *match* is only executed after a successful comparison of a symbol of p which is least frequent in S, and the corresponding symbol of v. In the taxonomy in [HS91] this comparison is called the *guard* and the symbol of p is called the *guard symbol*. We do not consider it here since it can be viewed as additionally requiring that $p_{mo(1)}$ is a symbol of p with minimal frequency in S. □

Remark 4.156: A number of variants of the Boyer-Moore algorithm are considered in [CR94, Chapter 4]. Some of these variants use information from previous match attempts to reduce the number of symbol comparisons (in v and p) that occur in subsequent match attempts. While such improvements appear to lead to algorithms which are efficient in practice, the approach is not considered further in this dissertation. □

4.5.1 Larger shifts without using *match* information

It may be possible to make an additional shift (immediately before *match* is invoked) providing no matches are missed. In this section, we consider making such a shift provided that it can be cheaply implemented. On certain computer architectures, some of the shifts described below can be implemented with as few as three machine instructions (see Chapter 9 for more on implementation issues). A shift of not greater than $(\mathbf{MIN}\ k : 0 \leq k \leq |r| \wedge \mathit{PerfMatch}(\mathit{shift}(l, v, r, k)) : k)\ \mathbf{min}\ |r|$ will be safe. The $\mathbf{min}\ |r|$ is used to ensure that $|v| = |p|$ is maintained, as required for the precondition of *match*.

Remark 4.157: This definition of a safe shift is similar to the one given in Definition 4.89, rewritten to make use of function *shift* and predicate *PerfMatch*. In Definition 4.89, a lower bound of 1 is placed on the bound variable in the **MIN** quantification, while in this section we use a lower bound of 0. The difference depends upon the position (in the algorithm) of the shift: a lower bound of 0 is used for a shift immediately before a match attempt, while a shift of 1 is used for a shift after a match attempt. That is, the lower bound 0 is used here because there may be a keyword match at the current position in the input string (a shift of distance 0), and function *match* is still to be executed. In Section 4.5.2, we will again consider safe shift distances of at least one symbol. □

The shift can be performed with the statement

$\{\ |v| = |p|\ \}$
$(l, v, r) := \mathit{shift}(l, v, r, (\mathbf{MIN}\ k : 0 \leq k \leq |r| \wedge \mathit{PerfMatch}(\mathit{shift}(l, v, r, k)) : k)\ \mathbf{min}\ |r|)$
$\{\ |v| = |p| \wedge (r = \varepsilon \vee \mathit{PerfMatch}((l, v, r)))\ \}$

The safe shift can be implemented in another way, as in the following definition.

Definition 4.158 (Skip loop): We define the following algorithm fragment to be a *skip loop*:

$\{\ |v| = |p|\ \}$
do $1 \leq |r| \wedge \neg \mathit{PerfMatch}((l, v, r)) \rightarrow$
    $(l, v, r) := \mathit{shift}(l, v, r, (\mathbf{MIN}\ k : 1 \leq k \leq |r| \wedge \mathit{PerfMatch}(\mathit{shift}(l, v, r, k)) : k)\ \mathbf{min}\ |r|)$
od
$\{\ |v| = |p| \wedge (r = \varepsilon \vee \mathit{PerfMatch}((l, v, r)))\ \}$

This implementation of the safe shift is known as a *skip loop* in the taxonomy of Hume and Sunday [HS91]. □

Remark 4.159: Since at most one step is taken by the skip loop (in its present form), this could have been implemented with an **if**-**fi** construct, however, the **do**-**od** construct will prove to be more useful when the shift distance is approximated from below. □

Convention 4.160 (Skip loop shift distance): The shift distance in the skip loop is also known as a *skip length*. □

As in Section 4.4, we note that calculating the **MIN** quantification is essentially as difficult as the problem we are trying to solve. Since any smaller shift length suffices, we consider weakenings of predicate *PerfMatch* in the **MIN** range predicate. Some weakenings are: $J_0((l,v,r)) \equiv true$, $J_1((l,v,r)) \equiv (v_1 = p_1)$, $J_2((l,v,r)) \equiv (v_{|p|} = p_{|p|})$, and $J_3((l,v,r)) \equiv (v_j = p_j)$ (for some $j : 1 \leq j \leq |p|$); the predicates J_1, J_2 and J_3 require that $p \neq \varepsilon$. These particular weakenings were chosen to be cheap to evaluate, requiring as few as a single machine instruction on some machines.

Remark 4.161: Predicates J_1 and J_2 are special cases of J_3. We can of course take the conjunction of any of these weakenings and still have a weakening of *PerfMatch*. □

Evaluating predicate *PerfMatch* in the skip loop guard is equivalent to an invocation of function *match*. Fortunately, we can make use of the weakenings of *PerfMatch* here as well. Since $PerfMatch(l,v,r) \Rightarrow J_3(l,v,r)$ (and therefore $\neg J_3(l,v,r) \Rightarrow \neg PerfMatch(l,v,r)$), we will use a weakening in place of *PerfMatch* in the skip loop guard.

For each weakening of *PerfMatch*, we assume that the weakening is used in place of *PerfMatch* in the guard, and we consider the resulting shift length as calculated with the quantified **MIN** under the assumption that the guard holds. In the case of J_0, the entire skip loop is equivalent to **skip**.

We consider the shift length for J_3 before returning to J_1 and J_2 as special cases. We need to compute

$$(\mathbf{MIN}\ k : 1 \leq k \leq |r| \wedge PerfMatch(shift(l,v,r,k)) : k)$$

In order to easily compute this we will weaken the range predicate, removing lookahead. It is known (from the **do**-**od** guard) that $\neg J_3((l,v,r))$ holds. The derivation proceeds as follows (assuming $1 \leq k \leq |r|$, $|v| = |p|$, $\neg J_3((l,v,r))$ and fixed $j : 1 \leq j \leq |p|$):

$$\begin{array}{ll}
 & PerfMatch(shift(l,v,r,k)) \\
\equiv & \quad \{ \text{definition of } shift \} \\
 & PerfMatch((l(vr\upharpoonright k),(v(r\upharpoonright k))\downharpoonright k, r\downharpoonright k)) \\
\Rightarrow & \quad \{ \text{definition of } PerfMatch \} \\
 & (v(r\upharpoonright k))\downharpoonright k = p \\
\equiv & \quad \{ \text{definition of = on strings, } k \leq |r|,\ |v| = |p| \} \\
 & (\forall\ h : 1 \leq h \leq |p| : ((v(r\upharpoonright k))\downharpoonright k)_h = p_h) \\
\equiv & \quad \{ \text{rewrite } \downharpoonright \text{ into indexing} \} \\
 & (\forall\ h : 1 \leq h \leq |p| : (v(r\upharpoonright k))_{h+k} = p_h) \\
\Rightarrow & \quad \{ \text{discard lookahead at } r,\ |v| = |p| \} \\
 & (\forall\ h : 1 \leq h \leq |p| - k : v_{h+k} = p_h) \\
\equiv & \quad \{ \text{change of bound variable: } h' = h + k \} \\
 & (\forall\ h' : 1 + k \leq h' \leq |p| : v_{h'} = p_{h'-k}) \\
\Rightarrow & \quad \{ \text{one point rule — quantification at } h' = j;\ j \leq |p| \}
\end{array}$$

$$
\begin{array}{ll}
& 1+k \leq j \Rightarrow v_j = p_{j-k} \\
\equiv & \quad \{\text{assumption: } \neg J_3((l,v,r))\} \\
& 1+k \leq j \Rightarrow v_j = p_{j-k} \wedge v_j \neq p_j \\
\Rightarrow & \quad \{\text{transitivity of } =\} \\
& 1+k \leq j \Rightarrow p_j \neq p_{j-k}
\end{array}
$$

We will use the last line to present one skip distance, and the the third last line ($1 + k \leq j \Rightarrow v_j = p_{j-k}$) to present another (greater) skip distance. The final predicate is free of r, and so the upperbound of $|r|$ on k can be dropped.

Remark 4.162: As in the Commentz-Walter algorithm variant presented in Section 4.4.8, we could have kept some right lookahead in the weakenings. Although this would have yielded a family of efficient algorithms, we do not consider such weakenings in this dissertation. □

In order to present the two skip distances concisely, we define an auxiliary function and a constant.

Definition 4.163 (Function sl_1 and constant sl_2): Given $j : 1 \leq j \leq |p|$ (the index used in predicate J_3), we can define function $sl_1 \in V \longrightarrow \mathbb{N}$ and constant $sl_2 \in \mathbb{N}$

$$
\begin{array}{lll}
sl_1(a) & = & (\mathbf{MIN}\ k : 1 \leq k \wedge (1+k \leq j \Rightarrow a = p_{j-k}) : k) \\
sl_2 & = & (\mathbf{MIN}\ k : 1 \leq k \wedge (1+k \leq j \Rightarrow p_j \neq p_{j-k}) : k)
\end{array}
$$

Note that sl_1 and sl_2 both depend implicitly on j. □

It follows from the derivation of the range predicates of these two functions, that sl_1 yields a greater skip distance than sl_2. The disadvantage to using sl_1 is that $\mathcal{O}(|V|)$ storage space is required to tabulate it.

Remark 4.164: In Section 4.5.2 we will show how each of sl_1 and sl_2 can be obtained from two functions computed for a different purpose. □

Remark 4.165: It is not too difficult to see that the skip distances of both sl_1 and sl_2 are bounded above by j. □

Example 4.166 (Functions sl_1 and sl_2): Assuming keyword $hehshe$, $j = 4$, and alphabet $\{e, h, i, r, s, y\}$, we have $sl_2 = 1$ and

a	e	h	i	r	s	y
$sl_1(a)$	2	1	4	4	4	4

□

Either $sl_1(v_j)\,\mathbf{min}\,|r|$ or $sl_2\,\mathbf{min}\,|r|$ can be used as the skip distance. In the remaining presentations of the algorithms, we will use sl_1 exclusively. If a conjunct of any of J_0, J_1, J_2, or J_3 is used as a weakening of *PerfMatch*, the appropriate skip length can be approximated as the **max** of the individual skip lengths. A particularly interesting skip length is that arising from predicate J_1 (in which $j = 1$). In this case, $sl_1(a) = 1$ (for all $a \in V$) and $sl_2 = 1$ and a skip length of 1 is always used, regardless of p or v.

Assuming J is a weakening of *PerfMatch* we introduce the following program detail.

Algorithm detail 4.167 (SL)**:** Comparison of v and p is preceded by a *skip loop* based upon weakening J of *PerfMatch* and some appropriate skip length. □

Assuming some fixed $j : 1 \le j \le |p|$ we use J_3 as an example of a weakening of *PerfMatch* in

Algorithm 4.168 (OKW, OBM, INDICES, MO, SL)**:**

$$
\begin{array}{l}
l, v, r := \varepsilon, S\!\upharpoonright\!|p|, S\!\downharpoonright\!|p|;\ O := \emptyset;\\
\{\ \text{invariant: } I_1(l, v, r)\ \}\\
\mathbf{do}\ |v| = |p| \to\\
\quad \{\ |v| = |p|\ \}\\
\quad \mathbf{do}\ 1 \le |r| \wedge \neg J_3((l, v, r)) \to\\
\quad\quad (l, v, r) := \mathit{shift}(l, v, r, sl_1(v_j)\,\mathbf{min}\,|r|)\\
\quad \mathbf{od};\\
\quad \{\ |v| = |p| \wedge (J_3((l, v, r)) \vee r = \varepsilon)\ \}\\
\quad i := \mathit{match}(v, p, mo);\\
\quad \{\ I_2(v, p, mo, i)\ \}\\
\quad \mathbf{if}\ i = |p| + 1 \to O := O \cup \{(l, v, r)\}\\
\quad [\!]\ \ i \ne |p| + 1 \to \mathbf{skip}\\
\quad \mathbf{fi};\\
\quad (l, v, r) := \mathit{shift}(l, v, r, 1)\\
\mathbf{od}\mathit{BCPM}\ \}
\end{array}
$$

□

Note that the efficiency, but not the correctness, of this algorithm is diminished by omitting the skip loop. Although the skip look looks costly to implement, it is usually compiled into a few machine instructions. In [HS91], it is shown that the use of a skip loop can yield significant improvements to the running time of most BM variants.

We proceed by presenting four instances of detail (SL) (each based on a weakening of *PerfMatch*)[6]:

> **Algorithm detail 4.169** (NONE)**:** The predicate J_0 (defined as *true*) is used as the weakening of *PerfMatch* in the skip loop. Notice that in this case the skip loop is equivalent to statement **skip**. □

[6] The names of the details are taken from the taxonomy in [HS91].

Algorithm detail 4.170 (SFC): The predicate J_1 (defined as $v_1 = p_1$) is used as the weakening of *PerfMatch*, along with the corresponding skip length of 1, in the skip loop. □

Algorithm detail 4.171 (FAST): The predicate J_2 (defined as $v_{|p|} = p_{|p|}$) is used as the weakening of *PerfMatch*, along with the corresponding skip length, in the skip loop. □

Algorithm detail 4.172 (SLFC): Let p_j be a symbol of p with minimal frequency in S. Predicate J_3 (defined as $v_j = p_j$) is used as the weakening of *PerfMatch*, along with the corresponding skip length, in the skip loop. □

4.5.2 Making use of *match* information

Up to now information from previous matching attempts was not used in the computation of the shift distance (in fact, there was no computation and the shift distance in the update of (l, v, r) at the end of the outer repetition defaulted to 1). In this section, we will take into account the information from the immediately preceding matching attempt, in much the same way as was done in the Commentz-Walter algorithm (see Section 4.4).

We would like to compute a safe shift distance of

$$(\mathbf{MIN}\ k : 1 \leq k \leq |r| \wedge \mathit{PerfMatch}(\mathit{shift}(l, v, r, k)) : k)$$

and again we proceed with a weakening of the range predicate (see Definition 4.89). In the following derivation we will make use of part of the weakening derivation in Section 4.5.1, and we will also assume $1 \leq k \leq |r|$, $|v| = |p|$ and the postcondition of *match*, namely $I_2(v, p, mo, i)$.

$$\begin{array}{ll}
 & \mathit{PerfMatch}(\mathit{shift}(l, v, r, k)) \\
\Rightarrow & \quad \{\ \text{derivation on page 105}\ \} \\
 & (\forall\ h' : 1 + k \leq h' \leq |p| : v_{h'} = p_{h'-k}) \\
\equiv & \quad \{\ \text{change of bound variable: } h' = mo(h)\ \} \\
 & (\forall\ h : 1 \leq h \leq |p| \wedge 1 + k \leq mo(h) : v_{mo(h)} = p_{mo(h)-k}) \\
\equiv & \quad \{\ I_2(v, p, mo, i)\ \} \\
 & (\forall\ h : 1 \leq h \leq |p| \wedge 1 + k \leq mo(h) : v_{mo(h)} = p_{mo(h)-k}) \\
 & \ \wedge (i \leq |p| \Rightarrow v_{mo(i)} \neq p_{mo(i)}) \wedge (\forall\ h : 1 \leq h < i : v_{mo(h)} = p_{mo(h)}) \\
\Rightarrow & \quad \{\ \text{one point rule (first quantification) — } h = i\ \} \\
 & (\forall\ h : 1 \leq h \leq |p| \wedge 1 + k \leq mo(h) : v_{mo(h)} = p_{mo(h)-k}) \\
 & \ \wedge (i \leq |p|\ \mathbf{cand}\ 1 + k \leq mo(i) \Rightarrow v_{mo(i)} \neq p_{mo(i)} \wedge v_{mo(i)} = p_{mo(i)-k}) \\
 & \ \wedge (\forall\ h : 1 \leq h < i : v_{mo(h)} = p_{mo(h)}) \\
\Rightarrow & \quad \{\ \text{combine } \forall \text{ quantifications, with restricted range, since } 1 \leq i \leq |p| + 1\ \} \\
 & (\forall\ h : 1 \leq h < i \wedge 1 + k \leq mo(h) : v_{mo(h)} = p_{mo(h)-k} \wedge v_{mo(h)} = p_{mo(h)})
\end{array}$$

$$
\begin{array}{ll}
& \wedge\ (i \leq |p| \ \mathbf{cand}\ 1+k \leq mo(i) \Rightarrow v_{mo(i)} \neq p_{mo(i)} \wedge v_{mo(i)} = p_{mo(i)-k}) \\
\Rightarrow & \quad \{\text{ transitivity of } = \text{ in quantification, to eliminate dependence upon } v\ \} \\
& (\forall\ h : 1 \leq h < i \wedge 1+k \leq mo(h) : p_{mo(h)} = p_{mo(h)-k}) \\
& \wedge\ (i \leq |p| \ \mathbf{cand}\ 1+k \leq mo(i) \Rightarrow v_{mo(i)} \neq p_{mo(i)} \wedge v_{mo(i)} = p_{mo(i)-k}) \\
\Rightarrow & \quad \{\text{ transitivity of } = \text{ in implication }\} \\
& (\forall\ h : 1 \leq h < i \wedge 1+k \leq mo(h) : p_{mo(h)} = p_{mo(h)-k}) \\
& \wedge\ (i \leq |p| \ \mathbf{cand}\ 1+k \leq mo(i) \Rightarrow p_{mo(i)} \neq p_{mo(i)-k} \wedge v_{mo(i)} = p_{mo(i)-k})
\end{array}
$$

Definition 4.173 (Predicate I_3): The last predicate in the preceding derivation will be denoted by $I_3(i,k)$ (here we have chosen to make parameters mo, p and v implicit). □

Based upon previous *match* information, a safe shift distance is

$$(\mathbf{MIN}\ k : 1 \leq k \wedge I_3(i,k) : k)$$

Notice that this shift distance still depends on implicit parameters mo and p.

In much of the literature, I_3 is broken up into three conjuncts:

$$
\begin{array}{lcl}
I_3'(i,k) & \equiv & (\forall\ h : 1 \leq h < i \wedge 1+k \leq mo(h) : p_{mo(h)} = p_{mo(h)-k}) \\
I_3''(i,k) & \equiv & (i \leq |p| \ \mathbf{cand}\ 1+k \leq mo(i) \Rightarrow v_{mo(i)} = p_{mo(i)-k}) \\
I_3'''(i,k) & \equiv & (i \leq |p| \ \mathbf{cand}\ 1+k \leq mo(i) \Rightarrow p_{mo(i)} \neq p_{mo(i)-k})
\end{array}
$$

In order to concisely present a shift distance, we define three auxiliary functions.

Definition 4.174 (Functions s_1, $char_1$, and $char_2$): Define functions $s_1 \in [1, |p|+1] \longrightarrow \mathbb{N}$, $char_1 \in [1, |p|+1] \longrightarrow \mathbb{N}$, and $char_2 \in [1, |p|+1] \longrightarrow \mathbb{N}$ as

$$
\begin{array}{lcl}
s_1(i) & = & (\mathbf{MIN}\ k : 1 \leq k \wedge I_3'(i,k) : k) \\
char_1(i) & = & (\mathbf{MIN}\ k : 1 \leq k \wedge I_3''(i,k) : k) \\
char_2(i) & = & (\mathbf{MIN}\ k : 1 \leq k \wedge I_3'''(i,k) : k)
\end{array}
$$

Function $char1$ implicitly uses v, and requires $\mathcal{O}(|V| \cdot (|p|+1))$ space for tabulation. □

All three of these functions are bounded above by $|p|$.

It should be noted that, when $I_2(v, p, mo, i)$ holds, $I_3''(i,k) \Rightarrow I_3'''(i,k)$. This means that $char_1$ always yields a greater shift distance than $char_2$. As with sl_1 and sl_2, either $char_1$ or $char_2$ can be used; the choice is a tradeoff between the $\mathcal{O}(|V| \cdot (|p|+1))$ space for $char_1$ and the $\mathcal{O}(|p|+1)$ space for $char_2$. In the following presentation, we will use only $char_1$.

Example 4.175 (Functions s_1 and $char_2$): Using keyword $hehshe$ and the (FWD) match order, we obtain

i	1	2	3	4	5	6	7
$s_1(i)$	1	1	2	2	4	4	4

and

i	1	2	3	4	5	6	7
$char_2(i)$	1	1	1	1	1	1	1

The shift distance provided by $char_2$ does not look promising. This is due to the particular keyword choice. Keywords in which a single symbol is repeated (such as $hhhh$) yield greater shift distances. □

Using functions s_1 and $char_1$ yields a new, possibly smaller, shift distance

$$s_1(i) \, \mathbf{max} \, char_1(i)$$

This is known as the *match information* detail.

Algorithm detail 4.176 (MI): Use information from the preceding match attempt by computing the shift distance using functions s_1 and either $char_1$ or $char_2$. □

Adding this detail, and integer variable *distance* for clarity, results in the following Boyer-Moore algorithm skeleton[7] (cf. [HS91, Section 4, p. 1224]):

Algorithm 4.177 (OKW, OBM, INDICES, MO, SL, MI):

$l, v, r := \varepsilon, S{\upharpoonright}|p|, S{\downharpoonright}|p|;\ O := \emptyset;$
$\{$ invariant: $I_1(l, v, r)$ $\}$
do $|v| = |p| \rightarrow$
  $\{\ |v| = |p|\ \}$
  do $1 \leq |r| \wedge \neg J_3((l, v, r)) \rightarrow$
    $(l, v, r) := \mathit{shift}(l, v, r, sl_1(v_j) \, \mathbf{min} \, |r|)$
  od;
  $\{\ |v| = |p| \wedge (J_3((l, v, r)) \vee r = \varepsilon)\ \}$
  $i := \mathit{match}(v, p, mo);$
  $\{\ I_2(v, p, mo, i)\ \}$
  if $i = |p| + 1 \rightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $i \neq |p| + 1 \rightarrow$ **skip**
  fi;
  $\mathit{distance} := s_1(i) \, \mathbf{max} \, char_1(i);$
  $(l, v, r) := \mathit{shift}(l, v, r, \mathit{distance})$
od$\{$ PM $\}$

□

Remark 4.178: Precomputation of functions s_1, $char_1$, and $char_2$ is briefly discussed in the original taxonomy [WZ92]. □

[7] Details (MO) and (SL) still have to be instantiated — weakening J_3 is used for the skip loop.

Given fixed $j : 1 \leq j \leq |p|$ we can easily compute the function sl_1 and constant sl_2 from Section 4.5.1. This can be done for any particular mo. The functions are

$$\begin{array}{lcl} sl_1(v_j) & = & char_1(mo^{-1}(j)) \\ sl_2 & = & char_2(mo^{-1}(j)) \end{array}$$

Example 4.179 (Computing function sl_2): Our example of sl_2 used $j = 4$, so

$$sl_2 = char_2(mo^{-1}(4)) = char_2(4) = 1$$

□

4.6 Conclusions

The highlights of this taxonomy fall into two categories: general results of the derivation method and specific results of the taxonomy. The general results are:

- The method of refinement used in each of the derivations presents the algorithms in an abstract, easily digested format. This presentation allows a correctness proof of an algorithm to be developed simultaneously with the algorithm itself.

- The presentation method proves to be more than just a method of deriving algorithms: the derivations themselves serve to classify the algorithms in the taxonomy. This is accomplished by dividing the derivation at points where either problem or algorithm details are introduced. A sequence of such details identifies an algorithm. By prefix-factoring these sequences, common parts of two algorithm derivations are presented simultaneously.

- The taxonomy of all algorithms considered can be depicted as a graph: the root represents the original (naïve) solution $O := (\cup\, l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\})$; edges represent the addition of a detail; and the internal vertices and leaves represent derived algorithms. This graph is shown in Figure 4.1 in Section 4.1. The utility of this graph is that it can be used as a table of contents to the taxonomy. Being interested in only a subset of the algorithms, for example the Aho-Corasick (AC) algorithms, does not necessitate reading all of the derivations; only the root-leaf paths that lead to the AC algorithms need to be read for a complete overview of these algorithms.

- The presentation was more than just a taxonomy. Instead of using completed derivations of known algorithms, which are frequently derived in different styles, all of the algorithms were derived in a common framework. This made it easier to determine similarities or differences between algorithms for the purpose of classifying them.

- The pattern matching overviews presented in [Aho90, CR94, Step94] are excellent introductions to many of the algorithms presented in this chapter. Unfortunately, they do not present all variants of the algorithms, nor does it present them in a fashion that allows one to contrast the algorithms with one another. The taxonomy in this chapter accomplished precisely this goal, of presenting algorithms in one framework for comparison. In deriving the algorithms for this taxonomy every attempt was made to thoroughly explore all of the possible variants. Our taxonomy is a thorough introduction to all variants of the four principal pattern matching algorithms presented in [Aho90, CR94, Step94].

Results concerning particular algorithms can be summarized as follows:

- As stated in [AC75], the AC algorithm is intended to be a generalization of the original Knuth-Morris-Pratt (KMP) algorithm — making use of automata theory. The classical derivations of the two (using automata and indices, respectively) do not serve to highlight their similarities, or differences.

  When derived in the same framework, it becomes apparent that the AC algorithm cannot be specialized to arrive at KMP; this can be seen from the derivation of the AC algorithm subtree of the taxonomy tree. The linear search (introduced in Section 4.3.4) used in the failure function AC algorithm (Algorithm 4.72) is quite different from the linear search used in the abstract KMP algorithm (Algorithm 4.76). Indices could have been introduced in Algorithm 4.72, although this does not yield the classically presented KMP algorithm. The relationship between the AC and KMP algorithms is in fact that they have a common ancestor algorithm (P_+, E, AC, LS).

- The abstract intermediate KMP algorithm (Algorithm 4.76) is in fact a new algorithm, albeit a variant of the AC algorithm. The running time of this new algorithm does not appear to be any better than Algorithm 4.72. The transformation (by adding indices) of Algorithm 4.76 into the classically presented KMP algorithm (Algorithm 4.84) was demonstrated to be straightforward.

- The original Aho-Corasick article [AC75] presented the 'optimal' version of the algorithm after the failure function version of the algorithm. Although the optimal algorithm was explained in [AC75] as using a transition function γ_f which is a composition of the extended forward trie τ_{ef} and failure function f_f, our derivation proceeded much more smoothly by deriving an algorithm which is a common ancestor of both the optimal and the failure function algorithms.

- The pattern matching Moore machine with transition function γ_f is the minimal Moore machine performing the Aho-Corasick pattern matching transduction.

- 'Predicate weakening' (of Sections 4.4 and 4.5) was instrumental in deriving various algorithms (and their correctness proofs) from the Commentz-Walter (CW) algorithm, in particular the Boyer-Moore (BM) algorithm. The CW algorithm has not emerged as a popular string pattern matching algorithm partly due to the difficulty in understanding it. The derivation presented in Section 4.4 arrives at the CW algorithm through a series of small transformations, starting with a naïve (quadratic running time) algorithm. This derivation makes the CW algorithm considerably easier to understand. Predicate weakening was also heavily used in deriving the 'match-order' variant of the BM algorithm.

- Commentz-Walter's intention was to combine the BM algorithm with automata theory, to produce an algorithm dealing with multiple keywords. The relationship between the two algorithms has previously remained obscured by the styles of presentation of the two algorithms (indices in BM, and automata in CW). As seen from Section 4.4 the BM algorithm can indeed be arrived at in the same framework (as the CW algorithm) as a special case. The publication of the Hume-Sunday taxonomy [HS91] motivated us to also derive the BM algorithm in an entirely different manner — making use of the concept of 'match-orders'.

- In both papers by Commentz-Walter describing her algorithm (in particular the technical report [Com79b]), the differences between methods of determining a safe shift amount were not made explicit. Indeed, that some of these shift functions were distinct was not mentioned in all cases. The derivation of the CW algorithm given in this chapter clearly defines the differences between the shift functions.

- In the BM algorithm, the functions contributing to a shift have been presented in several separate papers since the introduction of the original algorithm. Until the publication of the taxonomy by [HS91] it was difficult to examine the contribution of each shift function. Both Section 4.5 and [HS91] present a shift as consisting of components that can be readily replaced by an equivalent component, for example: the 'skip' loops, or the 'match-orders'. [HS91] emphasized effects on running-time of each component. Our taxonomy has emphasized the derivation of each of these components from a common specification.

Chapter 5

A new *RE* pattern matching algorithm

This chapter presents a Boyer-Moore type algorithm for regular expression pattern matching, answering an open problem posed by A.V. Aho in 1980 [Aho80, p. 342]. The new algorithm handles patterns specified by regular expressions — a generalization of the Boyer-Moore and Commentz-Walter algorithms, both considered in Chapter 4.

Like the Boyer-Moore and Commentz-Walter algorithms, the new algorithm makes use of shift functions which can be precomputed and tabulated. The precomputation algorithms are derived, and it is shown that the required shift functions can be precomputed from Commentz-Walter's d_1 and d_2 shift functions.

In certain cases, the Boyer-Moore (respectively Commentz-Walter) algorithm has greatly outperformed the Knuth-Morris-Pratt (respectively Aho-Corasick) algorithm (as discussed in Chapter 13). In testing, the algorithm presented in this chapter also frequently outperforms the regular expression generalization of the Aho-Corasick algorithm.

An early version of this algorithm was presented in [WW94]. The research reported in this chapter was performed jointly with Richard E. Watson of the Department of Mathematics, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada; he can be reached at `watsona@sfu.ca`.

5.1 Introduction

The pattern matching problem is: given a non-empty language L (over an alphabet[1] V) and an input string S (also over alphabet V), find all substrings of S that are in L. Several restricted forms of this problem have been solved (all of which are discussed in detail in Chapter 4, and in [Aho90, WZ92]):

- The Knuth-Morris-Pratt (Section 4.3.6 and [KMP77]) and Boyer-Moore (Sections 4.4.6 and 4.5 and [BM77]) algorithms solve the problem when L consists of a single word (the single keyword pattern matching problem).

- The Aho-Corasick (Section 4.3 and [AC75]) and Commentz-Walter (Section 4.4 and [Com79a, Com79b]) algorithms solve the problem when L is a finite set of

[1] Throughout this chapter we assume a fixed alphabet V.

(key)words (the multiple keyword pattern matching problem). The Aho-Corasick and Commentz-Walter algorithms are generalizations of the Knuth-Morris-Pratt and Boyer-Moore algorithms respectively.

- The case where L is a regular language (the regular expression pattern matching problem) can be solved as follows: a finite automaton is constructed for the language V^*L; each time the automaton enters a final state (while processing the input string S) a matching substring has been found. This algorithm is detailed in [Aho90, AHU74], and is a generalization of the algorithm presented in Section 4.3.7. Since it is also a generalization of the Knuth-Morris-Pratt and Aho-Corasick algorithms, we will refer to it as the GAC (generalized AC) algorithm. Until now, most practical algorithms solving the regular expression pattern matching problem are variants of the GAC algorithm.

Although the Knuth-Morris-Pratt and Aho-Corasick algorithms have better worst-case running time than the Boyer-Moore and Commentz-Walter algorithms (respectively), the latter two algorithms are known to be extremely efficient in practice (see Chapter 13 and [HS91, Wat94a]). Interestingly, to date no generalization (to the case where L is an arbitrary regular language) of the Boyer-Moore and Commentz-Walter algorithms has been discovered. In 1980, A.V. Aho stated the following open problem:

> It would also be interesting to know whether there exists a Boyer-Moore type algorithm for regular expression pattern matching. [Aho80, p. 342].

In this chapter, we present such an algorithm. As with the Boyer-Moore and Commentz-Walter algorithms, the new algorithm requires shift tables. The precomputation of these shift tables is discussed, and shown to be related to the shift tables used by the Commentz-Walter algorithm. Finally, the new algorithm is specialized to obtain a variant of the Boyer-Moore (single keyword) algorithm — showing that it is indeed a generalization of the Boyer-Moore algorithm. The algorithm has been implemented, and in practice it frequently displays better performance than the GAC algorithm.

The derivation of the new algorithm closely parallel the development of the Commentz-Walter algorithm (see Section 4.4), especially in the use of predicate weakening to find a practically computed shift distance. In the Commentz-Walter algorithm, information from previous match attempts is used to make a shift of at least one symbol; the shift functions are finite, and can therefore be tabulated. In the new algorithm, we also use information from previous match attempts; directly using the information may yield shift functions which are infinite, and therefore impossible to precompute. The main result in the development of the algorithm is a weakening step which allows us to use finite shift functions in place of the (possibly) infinite ones — thereby yielding a practical algorithm.

It should be noted that there does exist another regular expression pattern matching algorithm (due to R. Baeza-Yates [GB-Y91]) with good performance; that algorithm requires some precomputation on the input string, and is therefore suited to a different kind of problem than the one presented in this chapter.

This chapter is structured as follows:

- Section 5.2 gives the problem specification, and a simple first algorithm.
- Section 5.3 presents the essential idea of greater shift distances while processing the input text, as in the Boyer-Moore algorithm.
- Section 5.4 derives algorithms required for the precomputation of the shift functions used in the pattern matching algorithm.
- Section 5.5 specializes the new pattern matching algorithm to obtain the Boyer-Moore algorithm.
- Section 5.6 provides some data on the performance of the new algorithm versus the GAC algorithm.
- Section 5.7 discusses some techniques for further improving the performance of the algorithm.
- Section 5.8 presents the conclusions of this chapter.

5.2 Problem specification and a simple first algorithm

We begin this section with a precise specification of the regular language pattern matching problem.

Definition 5.1 (Regular pattern matching problem): Given an alphabet V, an input string $S \in V^*$, a regular expression E (the pattern expression), and regular language $L \subseteq V^*$ such that $L = \mathcal{L}_{RE}(E)$, establish postcondition

$$RPM: \ O = (\cup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap L) \times \{r\})$$

□

In the remainder of this chapter, we will use language L instead of regular expression E in order to make the algorithm derivation more readable. Note that the encoding of set O is precisely the same as was used in Chapter 4.

We pattern our naïve first algorithm after Algorithms 4.10 and 4.18 (from Chapter 4, pages 51 and 85). In this algorithm, the prefixes (u) of S and the suffixes (v) of u are considered in order of increasing length[2].

[2]Other orders of evaluation can also be used. This order is only chosen so as to arrive at an algorithm generally resembling the Boyer-Moore algorithm.

Algorithm 5.2:

$u, r := \varepsilon, S;$
if $\varepsilon \in L \rightarrow O := \{(\varepsilon, \varepsilon, S)\}$
$[\!]\ \varepsilon \notin L \rightarrow O := \emptyset$
fi;
do $r \neq \varepsilon \rightarrow$
    $u, r := u(r\upharpoonright 1), r\downharpoonleft 1;$
    $l, v := u, \varepsilon;$
    if $v \in L \rightarrow O := O \cup \{(l, v, r)\}$
    $[\!]\ v \notin L \rightarrow$ **skip**
    fi;
    do $l \neq \varepsilon$ **cand** $(l\upharpoonright 1)v \in$ **suff**$(L) \rightarrow$
        $l, v := l\downharpoonleft 1, (l\upharpoonright 1)v;$
        if $v \in L \rightarrow O := O \cup \{(l, v, r)\}$
        $[\!]\ v \notin L \rightarrow$ **skip**
        fi
    od
od$\{$ RPM $\}$

□

Remark 5.3: Note that we have used **if-fi** constructs in giving the initialization and updates to variable O (compare this with the use of $\times$ and $\cap$ in presenting the initialization and updates in Algorithm 4.10). This is done to facilitate the introduction of a finite automaton in the next section. □

Remark 5.4: The number of iterations of the inner repetition is $\mathcal{O}(|S| \cdot ((\mathbf{MAX}\ w : w \in L : |w|)\ \mathbf{min}\ |S|))$. This is not the same as the running time, as we have not taken the cost of operations such as $\varepsilon \in L$ and $v \in L$ into account. The implementation of guard conjunct $(l\upharpoonright 1)v \in$ **suff**(L) and expression $v \in L$ (in the update of variable O) remain unspecified. □

In order to make the algorithm more practical, we introduce a finite automaton, in much the same way that the reverse trie was introduced in Algorithm detail 4.17.

5.2.1 A more practical algorithm using a finite automaton

Since L is a regular language, we construct (from E) a (possibly nondeterministic) ε-*free* finite automaton $M = (Q, V, \delta, \emptyset, I, F)$ accepting L^R (the reverse language[3] of L). The transition function will be taken to have signature $\delta \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(Q)$.

[3] The reverse is used, since we will be using automaton M to consider the symbols of substring v in right to left order instead of left to right order; this is analogous to the way in which the *reverse* trie was used with Algorithm 4.18.

We will not explicitly present the automaton here (except in the examples); we assume that it was constructed (from regular expression E) by one of the well-known algorithms, for example those presented in Chapter 6. Since M is ε-*free*, we have the property that $\varepsilon \in L \equiv I \cap F \neq \emptyset$ (see Remark 2.94). Finite automata with ε-transitions could have been used; they are only excluded in order to simplify the definitions given here.

In a manner analogous to that in which the reverse trie was introduced into Algorithm 4.10, we now make use of the automaton M. We introduce a new variable C (the current state set) ranging over $\mathcal{P}(Q)$ with the invariant

$$C = \{ q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \}$$

String v is reversed in the conjunct since v is processed in reverse. Given this invariant, the conditional conjunct of the inner repetition guard in Algorithm 5.2, $(l{\upharpoonright}1)v \in \mathbf{suff}(L)$, is equivalent to $\delta(C, l{\upharpoonright}1) \neq \emptyset$. The new algorithm is:

Algorithm 5.5:

$u, r := \varepsilon, S;$
if $I \cap F \neq \emptyset \rightarrow O := \{(\varepsilon, \varepsilon, S)\}$
$[\!]$ $I \cap F = \emptyset \rightarrow O := \emptyset$
fi;
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $l, v, C := u, \varepsilon, I;$
  if $C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $C \cap F = \emptyset \rightarrow$ **skip**
  fi;
  { invariant: $u = lv \wedge C = \{ q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \}$ }
  do $l \neq \varepsilon$ **cand** $\delta(C, l{\upharpoonright}1) \neq \emptyset \rightarrow$
    $l, v, C := l{\downharpoonright}1, (l{\upharpoonright}1)v, \delta(C, l{\upharpoonright}1);$
    $\{ C \cap F \neq \emptyset \equiv v \in L \}$
    if $C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\}$
    $[\!]$ $C \cap F = \emptyset \rightarrow$ **skip**
    fi
  od
od{ RPM }

□

Remark 5.6: There are a number of choices in the implementation of the finite automaton M. In particular, if a deterministic finite automaton is used then the algorithm variable C would always be a singleton set (and the algorithm could be modified so that C ranges over Q instead of $\mathcal{P}(Q)$). The use of a deterministic automaton requires more costly precomputation (of the automaton), but enables the algorithm to process input string S faster. A nondeterministic automaton would involve cheaper precomputation, but the input string would be processed more slowly as all paths in the automaton are simulated. A hybrid solution is to begin with a nondeterministic automaton, and then construct (and tabulate) a deterministic automaton on-the-fly, as the nondeterministic automaton is simulated. The performance of various types of finite automata will be considered again in Chapter 14. In this chapter, we continue to use a possibly nondeterministic finite automaton. □

In order to make some of the derivations in subsequent sections more readable, we define some constants.

Definition 5.7 (Constants relating to automaton M): For each state $q \in Q$, we define constant m_q to be the length of a shortest word in $\overleftarrow{\mathcal{L}}(q)$. Define m to be the length of a shortest word in L. Intuitively, m_q is the length of a shortest path from a start state to state q in M, while m is the length of a shortest path from a start state to a final state in M. □

Example 5.8 (Pattern and corresponding FA): As an example of a regular language pattern, and a corresponding finite automaton, consider the language $L = \{bd, de\}\{c\}^*\{b\} \cup \{bda\}$ (over alphabet $V = \{a, b, c, d, e\}$). In this case, an automaton M (which is shown in Figure 5.1) accepts the language $L^R = \{b\}\{c\}^*\{db, ed\} \cup \{adb\}$. Coincidentally, automaton M is a DFA. The left languages of each of the states (for the automaton in Figure 5.1) are as follows:

$$
\begin{array}{rcl}
\overleftarrow{\mathcal{L}}(0) & = & \{\varepsilon\} \\
\overleftarrow{\mathcal{L}}(1) & = & \{a\} \\
\overleftarrow{\mathcal{L}}(2) & = & \{b\}\{c\}^* \\
\overleftarrow{\mathcal{L}}(3) & = & \{ad\} \cup \{b\}\{c\}^*\{d\} \\
\overleftarrow{\mathcal{L}}(4) & = & \{b\}\{c\}^*\{e\} \\
\overleftarrow{\mathcal{L}}(5) & = & \{adb\} \cup \{b\}\{c\}^*\{db\} \\
\overleftarrow{\mathcal{L}}(6) & = & \{b\}\{c\}^*\{ed\}
\end{array}
$$

Additionally, $m = 3$, $m_0 = 0$, $m_1 = m_2 = 1$, $m_3 = m_4 = 2$, and $m_5 = m_6 = 3$. Language L and automaton M will be used as our running example throughout this chapter.

Within examples, we will use names (such as L, V, and M) to refer to the concrete objects defined above, as opposed to the abstract objects used elsewhere in this chapter. □

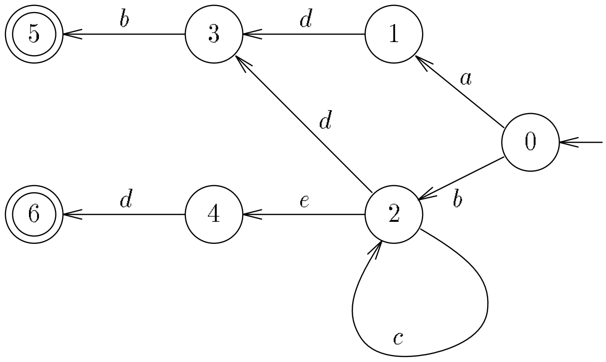


Figure 5.1: A finite automaton accepting the regular language $L^R = \{b\}\{c\}^*\{db, ed\} \cup \{adb\}$.

5.3 Greater shift distances

Upon termination of the inner repetition, we know (by the invariant of the inner repetition) that $C = \{\, q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \,\}$. This implies $(\forall\ q : q \in C : v^R \in \overleftarrow{\mathcal{L}}(q))$, and equivalently

$$(\forall\ q : q \in C : v \in \overleftarrow{\mathcal{L}}(q)^R)$$

In a manner analogous to the Commentz-Walter and Boyer-Moore algorithm derivations (Sections 4.4 and 4.5), this information can be used on a subsequent iteration of the outer repetition to make a shift k of more than one symbol by replacing the assignment $u, r := u(r{\upharpoonleft}1), r{\downharpoonleft}1$ by $u, r := u(r{\upharpoonleft}k), r{\downharpoonleft}k$.

In order to make use of this information (which relates v and C) on the first iteration of the outer repetition, we make the invariant of the inner repetition an invariant of the outer repetition as well, by adding the (redundant) initialization $l, v, C := u, \varepsilon, I$ before the outer repetition[4]:

Algorithm 5.9:

$u, r := \varepsilon, S;$
if $I \cap F \neq \emptyset \rightarrow O := \{(\varepsilon, \varepsilon, S)\}$
$[\!]\ \ I \cap F = \emptyset \rightarrow O := \emptyset$
fi;
$l, v, C := u, \varepsilon, I;$
$\{$ invariant: $u = lv \wedge C = \{\, q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \,\}\ \}$

[4] This does not change the nature of the algorithm, other than creating a new outer repetition invariant. A similar initialization was added to the Commentz-Walter algorithm skeleton, Algorithm 4.93.

$$
\begin{array}{l}
\textbf{do } r \neq \varepsilon \rightarrow \\
\quad u, r := u(r\upharpoonright 1), r\downharpoonright 1; \\
\quad l, v, C := u, \varepsilon, I; \\
\quad \textbf{if } C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\} \\
\quad [\!] \;\; C \cap F = \emptyset \rightarrow \textbf{skip} \\
\quad \textbf{fi}; \\
\quad \{ \text{ invariant: } u = lv \wedge C = \{ q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \} \ \} \\
\quad \textbf{do } l \neq \varepsilon \textbf{ cand } \delta(C, l\upharpoonright 1) \neq \emptyset \rightarrow \\
\quad\quad l, v, C := l\downharpoonright 1, (l\upharpoonright 1)v, \delta(C, l\upharpoonright 1); \\
\quad\quad \{ \ C \cap F \neq \emptyset \equiv v \in L \ \} \\
\quad\quad \textbf{if } C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\} \\
\quad\quad [\!] \;\; C \cap F = \emptyset \rightarrow \textbf{skip} \\
\quad\quad \textbf{fi} \\
\quad \textbf{od} \\
\textbf{od}\{ \ RPM \ \}
\end{array}
$$

□

5.3.1 A more efficient algorithm by computing a greater shift

We wish to use a shift distance which is possibly greater than 1 by replacing the assignment $u, r := u(r\upharpoonright 1), r\downharpoonright 1$ by $u, r := u(r\upharpoonright k), r\downharpoonright k$ (for $1 \leq k$). As with the Commentz-Walter and Boyer-Moore algorithms, we would like an ideal shift distance — the shift distance to the nearest match to the right (in input string S). Formally, this distance is given by: $(\textbf{MIN } n : 1 \leq n \leq |r| \wedge \textbf{suff}(u(r\upharpoonright n)) \cap L \neq \emptyset : n)$. Any shift distance less than this is also acceptable, and we define a safe shift distance (similar to that given in Definition 4.89).

Definition 5.10 (Safe shift distance): A shift distance k satisfying

$$1 \leq k \leq (\textbf{MIN } n : 1 \leq n \leq |r| \wedge \textbf{suff}(u(r\upharpoonright n)) \cap L \neq \emptyset : n)$$

is a safe shift distance. We call the upperbound (the quantification) the *maximal safe shift distance* or the *ideal shift distance*. □

Using a safe shift distance, the update of u, r then becomes $u, r := u(r\upharpoonright k), r\downharpoonright k$. In order to compute a safe shift distance, we will weaken predicate $\textbf{suff}(u(r\upharpoonright n)) \cap L \neq \emptyset$ (which we call the *ideal shift predicate*) in the range of the maximal safe shift distance quantification. This technique of using predicate weakening to find a more easily computed shift distance was introduced in Section 4.4.1. The weakest predicate *true* yields a shift distance of 1. We begin by finding a weakening of the ideal shift predicate which is stronger than *true*, but still precomputable; later we will present precomputation algorithms for the resulting approximation.

In the following weakening, we will first remove the dependency (of the ideal shift preciate) on l, then r, then v, leaving a weakening that only depends upon C and n

(and, of course, language L). The particular weakening that we derive will prove to yield precomputable shift tables. Assuming $1 \leq n \leq |r|$ and the (implied) invariant $u = lv \wedge (\forall\ q : q \in C : v \in \overleftarrow{\mathcal{L}}(q)^R)$, we begin with the ideal shift predicate:

$$\begin{array}{ll}
& \mathbf{suff}(u(r{\upharpoonright}n)) \cap L \neq \emptyset \\
\equiv & \quad \{\text{ invariant: } u = lv\ \} \\
& \mathbf{suff}(lv(r{\upharpoonright}n)) \cap L \neq \emptyset \\
\Rightarrow & \quad \{\text{ discard lookahead to } l\text{: } l \in V^*\text{, monotonicity of } \mathbf{suff} \text{ and } \cap\ \} \\
& \mathbf{suff}(V^*v(r{\upharpoonright}n)) \cap L \neq \emptyset \\
\Rightarrow & \quad \{\text{ domain of } r \text{ and } n\text{: } n \leq |r|\text{, so } (r{\upharpoonright}n) \in V^n\ \} \\
& \mathbf{suff}(V^*vV^n) \cap L \neq \emptyset \\
\equiv & \quad \{\text{ Property 2.59 }\} \\
& V^*vV^n \cap V^*L \neq \emptyset \\
\Rightarrow & \quad \{\text{ invariant: } (\forall\ q : q \in C : v \in \overleftarrow{\mathcal{L}}(q)^R)\text{, monotonicity of } \cap\ \} \\
& (\forall\ q : q \in C : V^*\overleftarrow{\mathcal{L}}(q)^R V^n \cap V^*L \neq \emptyset)
\end{array}$$

The predicate is now free of l, v, r and S and depends only on current state set C, automaton M, and language L (and therefore E). The fact that it is free of r allows us to drop the conjunct $n \leq |r|$ from the quantification giving the shift distance. We will continue this derivation from the last line.

Remark 5.11: As in the Commentz-Walter algorithms, we could have weakened the predicate in the above derivation to use one character of lookahead, $l{\upharpoonright}1$. With a single character of lookahead, it seems particularly difficult to arrive at easily precomputed shift functions, and that approach is not pursued in this dissertation. □

Forward reference 5.12: The fact that the language L and the languages $\overleftarrow{\mathcal{L}}(q)$ can be infinite (for a given $q \in Q$) makes evaluation of this predicate, $(\forall\ q : q \in C : V^*\overleftarrow{\mathcal{L}}(q)^R V^n \cap V^*L \neq \emptyset)$, difficult. In the following subsection, we will introduce the essential ingredient of this algorithm derivation, by deriving a more practical range predicate. □

5.3.2 Deriving a practical range predicate

In order to further weaken the predicate in Forward reference 5.12 (and find a more easily computed weakening), we aim at finite languages L_q (corresponding to each $q \in Q$) such that $V^*\overleftarrow{\mathcal{L}}(q)^R \subseteq V^*L_q$ and a finite language L' such that $V^*L \subseteq V^*L'$. This is the essential ingredient which reduces the shift functions from being infinite (not precomputable) to finite (which can be precomputed and tabulated). Possible definitions of such languages are as follows.

Definition 5.13 (Languages L_q and L'): Define the following languages (for all $q \in Q$)

$$\begin{array}{rcl} L_q & = & \mathbf{suff}(\overleftarrow{\mathcal{L}}(q)^R) \cap V^{m_q \,\mathbf{min}\, m} \\ L' & = & \mathbf{suff}(L) \cap V^m \end{array}$$

□

Remark 5.14: The definitions given here were chosen for their simplicity; other definitions are possible, but these particular ones lead to a generalization of the Boyer-Moore algorithm. □

Languages L_q satisfy an important property that we will require later.

Property 5.15 (Languages L_q): Assuming $M \in FA$ we have:

$$\mathit{Useful}_s(M) \equiv (\forall\ q : q \in Q : \overleftarrow{\mathcal{L}}(q) \neq \emptyset) \equiv (\forall\ q : q \in Q : L_q \neq \emptyset)$$

□

In the following property, we show that this definition of L_q satisfies the required property.

Property 5.16 (Languages L_q): For all $q \in Q$: $V^*\overleftarrow{\mathcal{L}}(q)^R \subseteq V^*L_q$ and $V^*L \subseteq V^*L'$.
Proof:
We can see that the definition of L_q satisfies a required property by considering a particular word w:

$$\begin{array}{ll} & w \in \overleftarrow{\mathcal{L}}(q)^R \\ \Rightarrow & \quad \{\text{ definition of } m_q\text{: } |w| \geq m_q \geq m_q \,\mathbf{min}\, m \,\} \\ & (\exists\ x, y : w = xy : x \in V^* \wedge y \in \mathbf{suff}(\overleftarrow{\mathcal{L}}(q)^R) \wedge |y| = m_q \,\mathbf{min}\, m) \\ \equiv & \quad \{\text{ definition of } \cap \text{ and } V^{m_q \,\mathbf{min}\, m} \,\} \\ & (\exists\ x, y : w = xy : x \in V^* \wedge y \in \mathbf{suff}(\overleftarrow{\mathcal{L}}(q)^R) \cap V^{m_q \,\mathbf{min}\, m}) \\ \equiv & \quad \{\text{ definition of } L_q \,\} \\ & (\exists\ x, y : w = xy : x \in V^* \wedge y \in L_q) \\ \equiv & \quad \{\text{ definition of concatenation of languages } \} \\ & w \in V^*L_q \end{array}$$

We conclude that $\overleftarrow{\mathcal{L}}(q)^R \subseteq V^*L_q$. It follows that $V^*\overleftarrow{\mathcal{L}}(q)^R \subseteq V^*V^*L_q$, and (since $V^*V^* = V^*$) $V^*\overleftarrow{\mathcal{L}}(q)^R \subseteq V^*L_q$. □

A similar proof applies to the L, L' case. Note that L_q and L' (for all $q \in Q$) are finite languages.

Example 5.17 (L_q and L'): Given our running example, we can see that

$$L' = \{bda, bdb, deb, dcb, ecb, ccb\}$$

and (for all states $0, \ldots, 6$ in finite automaton M):

$$\begin{array}{lcl} L_0 & = & \{\varepsilon\} \\ L_1 & = & \{a\} \\ L_2 & = & \{b\} \\ L_3 & = & \{da, db, cb\} \\ L_4 & = & \{eb, cb\} \\ L_5 & = & \{bda, bdb, dcb, ccb\} \\ L_6 & = & \{deb, ecb, ccb\} \end{array}$$

□

We can continue our previous derivation of a useable range predicate, from Forward reference 5.12.

$$\begin{array}{ll} & (\forall\ q : q \in C : V^* \overleftarrow{\mathcal{L}}(q)^R V^n \cap V^* L \neq \emptyset) \\ \Rightarrow & \quad \{\text{ Property 5.16 }\} \\ & (\forall\ q : q \in C : V^* L_q V^n \cap V^* L' \neq \emptyset) \\ \equiv & \quad \{\text{ existentially quantify over all } w \in L_q\ \} \\ & (\forall\ q : q \in C : (\exists\ w : w \in L_q : V^* w V^n \cap V^* L' \neq \emptyset)) \end{array}$$

We now have a usable weakening of the range predicate of the ideal shift distance.

Recalling Property 2.22, we can now proceed with our derivation (of an approximation), beginning with the ideal shift distance:

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r{\restriction}n)) \cap L \neq \emptyset : n) \\ \geq & \quad \{\text{ weakening of range predicate (see derivation above), free of } r \text{ so drop } n \leq |r|\ \} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge (\forall\ q : q \in C : (\exists\ w : w \in L_q : V^* w V^n \cap V^* L' \neq \emptyset)) : n) \\ \geq & \quad \{\text{ Property 2.22 — conjunctive } (\forall)\ \mathbf{MIN} \text{ range predicate; } |C| \text{ is finite}\ \} \\ & (\mathbf{MAX}\ q : q \in C : (\mathbf{MIN}\ n : 1 \leq n \wedge (\exists\ w : w \in L_q : V^* w V^n \cap V^* L' \neq \emptyset) : n))\ \mathbf{max}\ 1 \\ = & \quad \{\text{ Property 2.22 — disjunctive } (\exists)\ \mathbf{MIN} \text{ range predicate; } |L_q| \text{ is finite}\ \} \\ & (\mathbf{MAX}\ q : q \in C : (\mathbf{MIN}\ w : w \in L_q : (\mathbf{MIN}\ n : 1 \leq n \wedge V^* w V^n \cap V^* L' \neq \emptyset : n)))\ \mathbf{max}\ 1 \end{array}$$

The second step (above) warrants further explanation. In the case that $M \in FA$ has no start states ($I = \emptyset$), variable C will always be $\emptyset$. Since this would yield a shift distance of $-\infty$ we use $\mathbf{max}\ 1$ to ensure that the shift distance is at least 1.

In the case where a particular $L_q = \emptyset$, the outermost **MIN** quantification can take the value $+\infty$ — yielding a shift distance which is not safe. As mentioned in Property 5.15, this can be avoided by requiring that $Useful_s(M)$ holds. In practice, this is not necessary, since it is not possible that $q \in C \wedge L_q = \emptyset$ (this can be seen by inspection of the second conjunct of our algorithm invariant).

We now continue with the inner **MIN** quantification above:

$$
\begin{array}{ll}
& (\mathbf{MIN}\ n : 1 \le n \wedge V^*wV^n \cap V^*L' \neq \emptyset : n) \\
= & \quad \{\text{Property 2.60}\} \\
& (\mathbf{MIN}\ n : 1 \le n \wedge (V^*wV^n \cap L' \neq \emptyset \vee wV^n \cap V^*L' \neq \emptyset) : n) \\
= & \quad \{\text{Property 2.21; } \wedge \text{ distributes over } \vee\} \\
& (\mathbf{MIN}\ n : 1 \le n \wedge V^*wV^n \cap L' \neq \emptyset : n)\, \mathbf{min} (\mathbf{MIN}\ n : 1 \le n \wedge wV^n \cap V^*L' \neq \emptyset : n)
\end{array}
$$

This last line above can be written more concisely with the introduction of a pair of auxiliary functions.

Definition 5.18 (Functions d_1, d_2): We define two auxiliary functions $d_1, d_2 \in V^* \longrightarrow \mathbb{N}$ as:

$$
\begin{array}{lcl}
d_1(x) & = & (\mathbf{MIN}\ n : 1 \le n \wedge V^*xV^n \cap L' \neq \emptyset : n) \\
d_2(x) & = & (\mathbf{MIN}\ n : 1 \le n \wedge xV^n \cap V^*L' \neq \emptyset : n)
\end{array}
$$

Since both d_1 and d_2 are only applied to elements of L_q (for all $q \in Q$), we could have given their domains as $(\cup\ q : q \in Q : L_q)$. In Section 5.4.1, we will give an even more useful characterization of their domains. □

Note that these functions are almost identical to those defined in Definition 4.97 — only the domains are different.

Using the auxiliary functions, the last line of the derivation above can be written as $d_1(w)\, \mathbf{min}\, d_2(w)$. The approximation of the ideal shift distance is then:

$$(\mathbf{MAX}\ q : q \in C : (\mathbf{MIN}\ w : w \in L_q : d_1(w)\, \mathbf{min}\, d_2(w)))\, \mathbf{max}\, 1$$

For readability, we define another auxiliary function.

Definition 5.19 (Function t): Define auxiliary function $t \in Q \longrightarrow \mathbb{N}$ as

$$t(q) = (\mathbf{MIN}\ w : w \in L_q : d_1(w)\, \mathbf{min}\, d_2(w))$$

□

Remark 5.20: Functions d_1, d_2 and t are easily precomputed as discussed in Section 5.4. □

Using function t gives a shift distance of

$$(\mathbf{MAX}\ q : q \in C : t(q))\, \mathbf{max}\, 1$$

The final algorithm (using function t and introducing variable *distance* for readability) is:

Algorithm 5.21 (An efficient algorithm):

$u, r := \varepsilon, S$;
if $I \cap F \neq \emptyset \rightarrow O := \{(\varepsilon, \varepsilon, S)\}$
$[\!]$ $I \cap F = \emptyset \rightarrow O := \emptyset$
fi;
$l, v, C := u, \varepsilon, I$;
$\{$ invariant: $u = lv \wedge C = \{ q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \} \}$
do $r \neq \varepsilon \rightarrow$
  $distance := (\mathbf{MAX}\ q : q \in C : t(q))\ \mathbf{max}\ 1$;
  $u, r := u(r\upharpoonright distance), r\downharpoonright distance$;
  $l, v, C := u, \varepsilon, I$;
  if $C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $C \cap F = \emptyset \rightarrow$ **skip**
  fi;
  $\{$ invariant: $u = lv \wedge C = \{ q \mid q \in Q \wedge v^R \in \overleftarrow{\mathcal{L}}(q) \} \}$
  do $l \neq \varepsilon$ **cand** $\delta(C, l\upharpoonright 1) \neq \emptyset \rightarrow$
    $l, v, C := l\downharpoonright 1, (l\upharpoonright 1)v, \delta(C, l\upharpoonright 1)$;
    $\{ C \cap F \neq \emptyset \equiv v \in L \}$
    if $C \cap F \neq \emptyset \rightarrow O := O \cup \{(l, v, r)\}$
    $[\!]$ $C \cap F = \emptyset \rightarrow$ **skip**
    fi
  od
od$\{$ *RPM* $\}$

□

5.4 Precomputation

In this section, we consider the precomputation of languages L_q and L', and functions d_1, d_2, and t. The precomputation is presented as a series of small algorithms — each easier to understand than a single monolithic one. All algorithms are presented and derived in the reverse order of their application. In practice they would be combined into one algorithm, as is described in Section 5.4.9.

5.4.1 Characterizing the domains of functions d_1 and d_2

Since functions d_1 and d_2 are only applied to elements of L_q (for all $q \in Q$), their signatures can be taken as $d_1, d_2 \in (\cup\ q : q \in Q : L_q) \longrightarrow \mathbb{N}$. In order to make the precomputation of the functions easier, we need a different characterization of their domains. To do this in a simple manner, we require that $\mathit{Useful}_f(M)$ holds; intuitively, this means that for every state q in M, there is a path from q to a final state in M. In Chapter 6, several general finite automata construction algorithms are given; many of those algorithms construct automata with this property. Property $\mathit{Useful}_f(M)$ implies another property of M which will prove to be more directly useful.

Property 5.22 (Left languages): From the definition of Useful_f and Property 2.101 it follows that

$$(\forall\ q : q \in Q : \overleftarrow{\mathcal{L}}(q) \subseteq \mathbf{pref}(L^R))$$

□

From the property above, and the domains of d_1 and d_2, we can restrict the domains of d_1 and d_2 as follows (for all $q \in Q$):

$$\begin{array}{ll}
 & L_q \\
= & \quad \{ \text{definition of } L_q \} \\
 & \mathbf{suff}(\overleftarrow{\mathcal{L}}(q)^R) \cap V^{m_q \,\mathbf{min}\, m} \\
\subseteq & \quad \{ \text{Property 5.22; monotonicity of } \mathbf{suff} \} \\
 & \mathbf{suff}(\mathbf{pref}(L^R)^R) \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \text{Property 2.52 — } \mathbf{pref} \text{ and } \mathbf{suff} \text{ are duals; reverse is its own inverse} \} \\
 & \mathbf{suff}(\mathbf{suff}(L)) \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \text{Property 2.51 — idempotence of } \mathbf{suff} \} \\
 & \mathbf{suff}(L) \cap V^{m_q \,\mathbf{min}\, m} \\
\subseteq & \quad \{ m_q \,\mathbf{min}\, m \leq m;\ w \in V^{m_q \,\mathbf{min}\, m} \Rightarrow w \in \mathbf{suff}(V^m) \} \\
 & \mathbf{suff}(\mathbf{suff}(L) \cap V^m) \\
= & \quad \{ \text{definition of } L' \} \\
 & \mathbf{suff}(L')
\end{array}$$

Given this property (of each L_q), we can restrict the domain of functions d_1 and d_2 so that $d_1, d_2 \in \mathbf{suff}(L') \longrightarrow \mathbb{N}$. Since $|L'|$ is finite, then $|\mathbf{suff}(L')|$ is finite as well. Notice that this new signature (for d_1 and d_2) corresponds to that given in Definition 4.97, with (finite) keyword set L' in place of keyword set P.

Example 5.23 (Language L'): In our running example, where

$$L' = \{bda, bdb, deb, dcb, ecb, ccb\}$$

we have

$$\mathbf{suff}(L') = \{\varepsilon, a, b, da, db, eb, cb, bda, bdb, deb, dcb, ecb, ccb\}$$

Given the definitions of d_1, d_2, we can compute the two functions by hand:

w	ε	a	b	da	db	eb	cb	bda	bdb	deb	dcb	ecb	ccb
$d_1(w)$	1	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$d_2(w)$	3	3	2	3	2	2	2	3	2	2	2	2	2

□

Before precomputing d_1, d_2, we concentrate on the precomputation of function t.

5.4.2 Precomputing function t

Assuming that functions d_1 and d_2 and sets L_q (for all $q \in Q$) have been precomputed, we can compute function t as follows (variable tee is used to accumulate shift function t):

Algorithm 5.24 (Computing t):

```
for q : q ∈ Q →
    tee(q) := +∞
rof;
for q, u : q ∈ Q ∧ u ∈ L_q →
    tee(q) := tee(q) min d_1(u) min d_2(u)
rof{ tee = t }
```

□

Notice that we impose no unnecessary order of evaluation in either of the two repetitions. An implementor of this algorithm is free to choose an order of evaluation which is most efficient for the encoding used in the implementation.

Example 5.25 (Function t): In our running example, we obtain the following values for function t (given the values of L_q for all states q, and functions d_1 and d_2): $t(0) = 1, t(1) = 3, t(2) = 2, t(3) = 2, t(4) = 2, t(5) = 2, t(6) = 2$. □

5.4.3 Precomputing functions d_1 and d_2

With the domain of functions d_1 and d_2 restricted to $\mathbf{suff}(L')$, functions d_1 and d_2 are the Commentz-Walter precomputed functions for (finite) keyword set L' [Com79a, Com79b].

We now present two algorithms, computing d_1 and d_2 respectively. The algorithms are fully derived in [WZ95], and are given here without proofs of correctness. The two precomputation algorithms presented below depend upon the *reverse failure function* corresponding to keyword set L'.

Definition 5.26 (Function f_r): Function $f_r \in \mathbf{suff}(L') \setminus \{\varepsilon\} \longrightarrow \mathbf{suff}(L')$ corresponding to L' is defined as

$$f_r(u) = (\mathbf{MAX}_{\leq_p}\, w : w \in \mathbf{pref}(u) \setminus \{u\} \cap \mathbf{suff}(L') : w)$$

□

The reverse failure function is defined analogously to the forward failure function — see Definition 4.65 and Remark 4.66.

In the following two algorithms, we assume that function f_r is precomputed (variables *dee1* and *dee2* are used to accumulate d_1 and d_2, respectively):

Algorithm 5.27 (Computing d_1):

---

for $u : u \in \mathbf{suff}(L') \rightarrow$
  $dee1(u) := +\infty$
rof;
for $u : u \in \mathbf{suff}(L') \setminus \{\varepsilon\} \rightarrow$
  $dee1(f_r(u)) := dee1(f_r(u))\ \mathbf{min}(|u| - |f_r(u)|)$
rof$\{\ dee1 = d_1\ \}$

□

Again, notice that we impose no unnecessary order of evaluation in either of the two repetitions.

Algorithm 5.28 (Computing d_2):

---

for $u : u \in \mathbf{suff}(L') \rightarrow$
  $dee2(u) := +\infty$
rof;
for $u : u \in L' \rightarrow$
  $v := u$;
  do $v \neq \varepsilon \rightarrow$
    $v := f_r(v)$;
    if $|u| - |v| < dee2(v) \rightarrow dee2(v) := |u| - |v|$
    $[\!]\ \ |u| - |v| \geq dee2(v) \rightarrow v := \varepsilon$
    fi
  od
rof;
$n := 1$;
do $\mathbf{suff}(L') \cap V^n \neq \emptyset \rightarrow$
  for $u : u \in \mathbf{suff}(L') \cap V^n \rightarrow$
    $dee2(u) := dee2(u)\ \mathbf{min}\ dee2(u \rfloor 1)$
  rof;
  $n := n + 1$
od$\{\ dee2 = d_2\ \}$

□

Notice that the third (un-nested) repetition is a breadth-first traversal of the set $\mathbf{suff}(L')$, and the second (un-nested) repetition requires that function f_r is precomputed. By the definition of language L', the depth of the traversal is m.

Precomputation using these algorithms has been found to be cheap in practice [Wat94a].

5.4.4 Precomputing function f_r

The following algorithm (taken largely from [WZ92, Section 6, p. 33]) computes function f_r (variable *effr* is used to accumulate f_r):

Algorithm 5.29 (Computing f_r):

---

for $a : a \in V \rightarrow$
  if $a \in \mathbf{suff}(L') \rightarrow \mathit{effr}(a) := \varepsilon$
  $[\!]$ $a \notin \mathbf{suff}(L') \rightarrow$ **skip**
  fi
rof;
$n := 1$;
$\{$ invariant: $(\forall\ u : u \in \mathbf{suff}(L') \wedge |u| \leq n : \mathit{effr}(u) = f_r(u))$ $\}$
do $\mathbf{suff}(L') \cap V^n \neq \emptyset \rightarrow$
  for $u, a : u \in \mathbf{suff}(L') \cap V^n \wedge a \in V \rightarrow$
    if $au \in \mathbf{suff}(L') \rightarrow$
      $u' := \mathit{effr}(u)$;
      do $u' \neq \varepsilon \wedge au' \notin \mathbf{suff}(L') \rightarrow$
        $u' := \mathit{effr}(u')$
      od;
      if $u' = \varepsilon \wedge a \notin \mathbf{suff}(L') \rightarrow \mathit{effr}(au) := \varepsilon$
      $[\!]$ $u' \neq \varepsilon \vee a \in \mathbf{suff}(L') \rightarrow \mathit{effr}(au) := au'$
      fi
    $[\!]$ $au \notin \mathbf{suff}(L') \rightarrow$ **skip**
    fi
  rof;
  $n := n + 1$
od
$\{ n > m \}$
$\{ \mathit{effr} = f_r \}$

□

This algorithm also makes use of a breadth-first traversal (of depth m) of the set $\mathbf{suff}(L')$.

Example 5.30 (Function f_r): Consider the function f_r for our running example:

$w : w \in \mathbf{suff}(L') \setminus \{\varepsilon\}$	a	b	da	db	eb	cb	bda	bdb	deb	dcb	ecb	ccb
$f_r(w)$	ε	ε	ε	ε	ε	ε	b	b	ε	ε	ε	ε

□

5.4.5 Precomputing sets L_q

The languages L_q can easily be precomputed using relation $Reach(M)$ (see Definition 2.95) and two auxiliary functions.

Definition 5.31 (Functions st and emm): The auxiliary functions are $st \in \mathbf{suff}(L') \longrightarrow \mathcal{P}(Q)$ and $emm \in Q \longrightarrow [0, m]$ defined as:

$$\begin{array}{lcl} st(u) & = & \{ q \mid q \in Q \wedge u^R \in \overleftarrow{\mathcal{L}}(q) \} \\ emm(q) & = & m_q \,\mathbf{min}\, m \end{array}$$

□

Relation $Reach(M)$ will be used along with the following property of finite automata.

Property 5.32 (Reachability and left languages): A useful property (of any finite automaton) is that (for all states $q \in Q$):

$$\mathbf{pref}(\overleftarrow{\mathcal{L}}(q)) = (\cup\, p : p \in Q \wedge (p,q) \in Reach(M) : \overleftarrow{\mathcal{L}}(p))$$

□

Given relation $Reach(M)$ and functions emm and st, we now derive an expression for L_q that is easier to compute (than the definition):

$$\begin{array}{ll}
 & L_q \\
= & \quad \{ \text{definition of } L_q \} \\
 & \mathbf{suff}(\overleftarrow{\mathcal{L}}(q)^R) \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \text{Property 2.52 — } \mathbf{pref} \text{ and } \mathbf{suff} \text{ are duals} \} \\
 & \mathbf{pref}(\overleftarrow{\mathcal{L}}(q))^R \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \text{Property 5.32} \} \\
 & (\cup\, p : p \in Q \wedge (p,q) \in Reach(M) : \overleftarrow{\mathcal{L}}(p))^R \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \cap \text{ and } R \text{ distribute over } \cup \} \\
 & (\cup\, p : p \in Q \wedge (p,q) \in Reach(M) : \overleftarrow{\mathcal{L}}(p)^R \cap V^{m_q \,\mathbf{min}\, m}) \\
= & \quad \{ \text{quantify over all words } w : w \in \overleftarrow{\mathcal{L}}(p)^R \cap V^{m_q \,\mathbf{min}\, m} \} \\
 & (\cup\, w, p : p \in Q \wedge (p,q) \in Reach(M) \wedge w \in \overleftarrow{\mathcal{L}}(p)^R \cap V^{m_q \,\mathbf{min}\, m} : \{w\}) \\
= & \quad \{ \text{definition of } \cap \} \\
 & (\cup\, w, p : p \in Q \wedge (p,q) \in Reach(M) \wedge w \in \overleftarrow{\mathcal{L}}(p)^R \wedge : \{w\}) \\
= & \quad \{ L_q \subseteq \mathbf{suff}(L');\ w \in \overleftarrow{\mathcal{L}}(p)^R \equiv w^R \in \overleftarrow{\mathcal{L}}(p) \equiv p \in st(w) \} \\
 & (\cup\, w, p : p \in Q \wedge (p,q) \in Reach(M) \wedge w \in \mathbf{suff}(L') \wedge p \in st(w) \wedge w \in V^{m_q \,\mathbf{min}\, m} : \{w\}) \\
= & \quad \{ w \in V^{m_q \,\mathbf{min}\, m} \equiv |w| = m_q \,\mathbf{min}\, m \equiv |w| = emm(q) \} \\
 & (\cup\, w, p : p \in Q \wedge (p,q) \in Reach(M) \wedge w \in \mathbf{suff}(L') \wedge p \in st(w) \wedge |w| = emm(q) : \{w\})
\end{array}$$

Assuming that relation *Reach* and auxiliary functions *emm* and *st* are precomputed, we can now present an algorithm computing L_q (for all $q \in Q$, using variable *ell* to accumulate the sets L_q):

Algorithm 5.33 (Computing L_q):

$$
\begin{array}{l}
\textbf{for } q : q \in Q \rightarrow \\
\quad ell(q) := \emptyset \\
\textbf{rof}; \\
\textbf{for } p, q, w : (p, q) \in Reach(M) \wedge w \in \textbf{suff}(L') \wedge p \in st(w) \wedge |w| = emm(q) \rightarrow \\
\quad ell(q) := ell(q) \cup \{w\} \\
\textbf{rof}\{ \ (\forall\ q : q \in Q : ell(q) = L_q) \ \}
\end{array}
$$

□

5.4.6 Precomputing function *emm*

Assuming that function *st* and length m have already been computed, the following algorithm computes function *emm* using a breadth-first traversal of $\textbf{suff}(L')$:

Algorithm 5.34 (Computing *emm*):

$$
\begin{array}{l}
\textbf{for } q : q \in Q \rightarrow \\
\quad \textbf{if } q \in I \rightarrow emm(q) := 0 \\
\quad [\!] \ \ q \notin I \rightarrow emm(q) := m \\
\quad \textbf{fi} \\
\textbf{rof}; \\
n := 1; \\
\textbf{do } \textbf{suff}(L') \cap V^n \neq \emptyset \rightarrow \\
\quad \textbf{for } u : u \in \textbf{suff}(L') \cap V^n \rightarrow \\
\quad\quad \textbf{for } q : q \in st(u) \rightarrow \\
\quad\quad\quad emm(q) := emm(q) \ \textbf{min} \ n \\
\quad\quad \textbf{rof} \\
\quad \textbf{rof}; \\
\quad n := n + 1 \\
\textbf{od}\{ \ (\forall\ q : q \in Q : emm(q) = m_q \ \textbf{min} \ m) \ \}
\end{array}
$$

□

5.4.7 Precomputing function *st* and languages L' and $\textbf{suff}(L')$

The following algorithm makes a breadth-first traversal (of depth m) of the transition graph of finite automaton M. It simultaneously computes function *st*, languages L' and $\textbf{suff}(L')$, and m (the length of a shortest word in language L).

Languages L' and $\mathbf{suff}(L')$ are used in most of the precomputation algorithms already presented. While the following algorithm computes language $\mathbf{suff}(L')$, it is also an example of a breadth-first traversal of $\mathbf{suff}(L')$ without having to explicitly compute and store it; instead, the algorithm traverses the transition graph of finite automaton M and implicitly performs a breadth-first traversal of $\mathbf{suff}(L')$.

Algorithm 5.35 (Computing st, L', and $\mathbf{suff}(L')$):

$$
\begin{array}{l}
st(\varepsilon), current, SLprime, n, \mathit{final} := I, \{\varepsilon\}, \{\varepsilon\}, 0, (I \cap F = \emptyset);\\
\{ \text{ invariant: } current = \mathbf{suff}(L') \cap V^n\\
\qquad\qquad \wedge\ SLprime = (\cup\ i : i \leq n : \mathbf{suff}(L') \cap V^i)\\
\qquad\qquad \wedge\ 0 \leq n \leq m\\
\qquad\qquad \wedge\ (\mathit{final} \equiv n = m)\\
\qquad\qquad \wedge\ (\forall\ u : u \in \mathbf{suff}(L') \wedge |u| \leq n : st(u) = \{ q \mid u^R \in \overleftarrow{\mathcal{L}}(q) \}) \ \}\\
\mathbf{do}\ \neg \mathit{final} \rightarrow\\
\quad current' := \emptyset;\\
\quad n := n + 1;\\
\quad \mathbf{for}\ u, a : u \in current \wedge a \in V \rightarrow\\
\qquad \mathbf{if}\ \delta(st(u), a) \neq \emptyset \rightarrow\\
\qquad\quad \{ \ au \in \mathbf{suff}(L') \cap V^n \ \}\\
\qquad\quad st(au) := \delta(st(u), a);\\
\qquad\quad \{ \ (\forall\ q : q \in st(au) : au \in \overleftarrow{\mathcal{L}}(q)^R) \ \}\\
\qquad\quad current' := current' \cup \{au\};\\
\qquad\quad \mathit{final} := \mathit{final} \vee (st(au) \cap F \neq \emptyset)\\
\qquad [\!]\ \delta(st(u), a) = \emptyset \rightarrow \mathbf{skip}\\
\qquad \mathbf{fi}\\
\quad \mathbf{rof};\\
\quad current := current';\\
\quad SLprime := SLprime \cup current\\
\mathbf{od}\\
\{ \ n = m \ \}\\
\{ \ current = \mathbf{suff}(L') \cap V^m = L' \ \}\\
\{ \ SLprime = \mathbf{suff}(L') \ \}\\
\{ \ (\forall\ u : u \in \mathbf{suff}(L') : st(u) = \{ q \mid u^R \in \overleftarrow{\mathcal{L}}(q) \}) \ \}
\end{array}
$$

□

5.4.8 Precomputing relation $Reach(M)$

Relation $Reach(M)$ can be precomputed by a reflexive and transitive closure algorithm. The algorithm is (where I_Q is the identity binary relation on state set Q):

Algorithm 5.36 (Computing $Reach(M)$):

$Rch := \emptyset$;
for $q, a : q \in Q \wedge a \in V \rightarrow$
  $Rch := Rch \cup \{(q,q)\} \cup (\{q\} \times \delta(\{q\}, a))$
rof;
$\{ \; Rch = I_Q \cup \bar{\pi}_2(\delta) \; \}$
$change := true$;
do $change \rightarrow$
  $change := false$;
  for $p, q, r : (p,q) \in Rch \wedge (q,r) \in Rch \rightarrow$
    $change := change \vee (p,r) \notin Rch$;
    $Rch := Rch \cup \{(p,r)\}$
  rof
od$\{ \; Rch = Reach(M) \; \}$

□

5.4.9 Combining the precomputation algorithms

The precomputation algorithms can be combined into a single monolithic algorithm. Such an algorithm is essentially the sequential concatenation of the separate precomputation algorithms. The order in which the algorithms are applied is determined by their dependency graph, which is shown in Figure 5.2. A possible order of execution is obtained by reversing a topological sort of the dependency graph. One such order is: (Algorithms) 5.36, 5.35, 5.34, 5.33, 5.29, 5.27, 5.28, 5.24.

5.5 Specializing the pattern matching algorithm

By restricting the form of the regular expression patterns, we can specialize the pattern matching algorithm to obtain the Boyer-Moore and the Commentz-Walter algorithms. In this section, we specialize to obtain a variant of the Boyer-Moore algorithm that does not use a lookahead symbol.

To obtain the single-keyword pattern matching problem, we require that L be a singleton set; that is $L = \{p\}$ (problem detail (OKW) from Chapter 4), a language consisting of a single keyword.

We can now give a finite automaton accepting L^R.

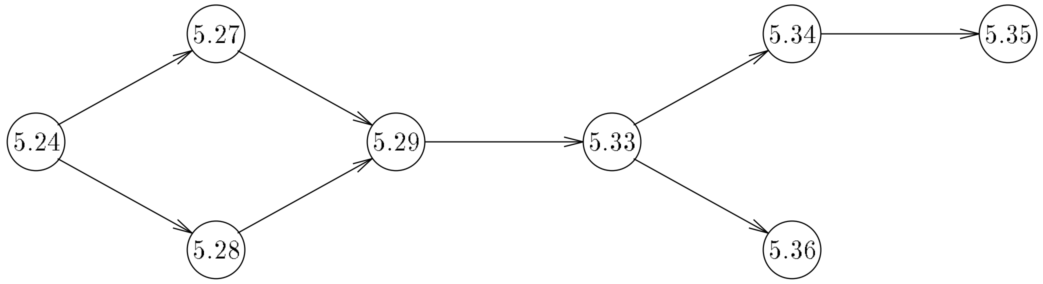


Figure 5.2: The dependency graph of the precomputation algorithms. An arrow from algorithm a to algorithm b indicates that algorithm b must be executed before algorithm a.

Definition 5.37 (Finite automaton accepting L^R): We define deterministic finite automaton $M = (\mathbf{suff}(p), V, \gamma, \emptyset, \{\varepsilon\}, \{p\})$. The states are elements of $\mathbf{suff}(p)$. We define deterministic transition function $\gamma \in \mathbf{suff}(p) \times V \longrightarrow \mathbf{suff}(p) \cup \{\perp\}$ (the special value $\perp$ denotes an undefined transition) as:

$$\gamma(w, a) = \begin{cases} aw & \text{if } aw \in \mathbf{suff}(p) \\ \perp & \text{otherwise} \end{cases}$$

□

$\mathit{Useful}_f(M)$ holds. Given function γ, we have (for every state $w \in \mathbf{suff}(p)$):

$$\overleftarrow{\mathcal{L}}(w) = \{w^R\}$$

Automaton M is deterministic, and the current state-set variable (C in the algorithm) is always a singleton set; call it state $w \in \mathbf{suff}(p)$. Since $\overleftarrow{\mathcal{L}}(w)$ is a singleton set and $|w| \leq |p|$, we have $m_w = |w|$ and $L_w = \overleftarrow{\mathcal{L}}(w)^R = \{w\}$. Additionally, since $m = |p|$, $L' = L = \{p\}$. Clearly, we have $L_w \subseteq \mathbf{suff}(L') = \mathbf{suff}(p)$. Function t is given by $t(w) = d_1(w) \,\mathbf{min}\, d_2(w)$. The shift distance will then be $d_1(w) \,\mathbf{min}\, d_2(w)$ in the update of variables u, r. Elements of $\mathbf{suff}(p)$ (in particular, current state variable w) can be encoded as integer indices (into string p) in the range $[0, |p|]$, as was done in Section 4.3.6.1. By making use of this encoding, and changing the domain of the variables u, r and functions d_1, d_2 to make use of indexing in input string S, we obtain the Boyer-Moore algorithm. The Commentz-Walter algorithm can similarly be obtained as a specialization.

5.6 The performance of the algorithm

Empirical performance data was gathered by implementing this algorithm in a `grep` style pattern matching tool, running under UNIX (on a SUN SPARC STATION 1+) and MS-DOS (on a 20 Mhz 386DX).

On each run, the new algorithm was used in addition to the old (generalized Aho-Corasick) algorithm which constructs a finite automaton accepting the language V^*L. (For both the old and the new algorithms, only deterministic finite automata were used. The time required for precomputation was not measured, but for both algorithms it appeared to be negligible compared to the time required to process the input string.) In the cases where $m \geq 6$ (the length of the shortest word in L is at least 6), and $|L'| \leq 18$, this new algorithm outperforms the other algorithm. These conditions held on approximately 35% of our user-entered regular expression patterns.

In the cases where the new algorithm outperformed the traditional one, the differences in execution speed varied from a 5% improvement to a 150% improvement. In the cases where the new algorithm was outperformed, its execution speed was never less than 30% of the execution speed of the traditional algorithm.

The conditions for obtaining high performance from the new algorithm ($m \geq 6 \wedge |L'| \leq 18$) can easily be determined from automaton M. In a `grep` style pattern matching tool, the automaton M can be constructed for language L^R. If the required conditions are met, the Boyer-Moore type pattern matcher presented in this chapter is used. If the conditions are not met, M can be reversed (so that it accepts language L), and converted to an automaton accepting V^*L. The traditional algorithm can then be used.

5.7 Improving the algorithm

In this section, we briefly consider two approaches to improving the expected performance of the algorithm. In the first approach, we consider the use of a right lookahead symbol, to improve the shift distance. In the second approach, we consider how the choice of a particular *FA* can affect the shift distance.

The use of a right lookahead symbol was first discussed, in the context of the Commentz-Walter algorithm, in Section 4.4.8. Had we retained a single symbol of right lookahead in this chapter, we would have arrived at the weakening

$$(\forall\ q : q \in C : V^* \overleftarrow{\mathcal{L}}(q)^R (r\rceil 1) V^{n-1} \cap V^* L \neq \emptyset)$$

in place of

$$(\forall\ q : q \in C : V^* \overleftarrow{\mathcal{L}}(q)^R V^n \cap V^* L \neq \emptyset)$$

in the derivation on page 123. The introduction of sets L_q and L' would remain unchanged, yielding the weakening

$$(\forall\ q : q \in C : (\exists\ w : w \in L_q : V^* w (r\rceil 1) V^{n-1} \cap V^* L' \neq \emptyset))$$

in place of the one on page 125. We could then use the same techniques as used in Section 4.4.8 to introduce an auxiliary function and give a new shift distance which uses this right lookahead symbol. This is left to the reader.

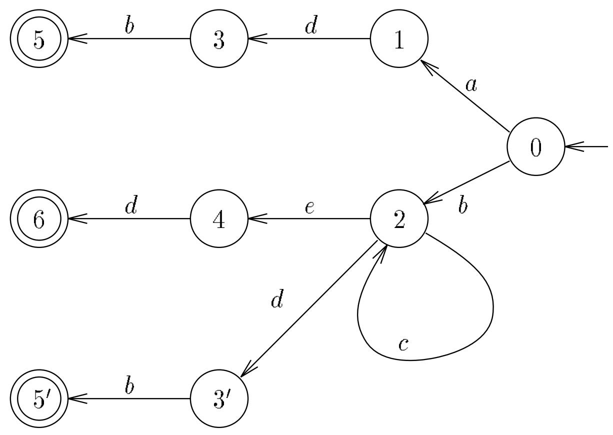


Figure 5.3: Improved automaton, equivalent to the one shown in Figure 5.1.

Another possible technique for improvement in the algorithm involves the following observation: it follows from the definition of function t (Definition 5.19) that smaller sets L_q can lead to greater shift distances. Consider the finite automaton used in the examples — see Figure 5.1. In this automaton we will consider two states in particular: 3 and 5. We see that $L_3 = \{da, db, cb\}$, $L_5 = \{bda, bdb, dcb, ccb\}$, $t(3) = 2$, and $t(5) = 2$ (see Examples 5.17 and 5.25). The relatively low shift distance for state 3 is due to the fact that it is not possible to tell (without modifying the algorithm) from the state number (3) whether the most recent in-transition was from state 1 or state 2.

If we were to split states 3 and 5, producing two new states $3'$ and $5'$, we would obtain the *FA* shown in Figure 5.3. In this new automaton, we have the following changes: $L_3 = \{da\}$, $L_{3'} = \{db, cb\}$, $L_5 = \{bda\}$, and $L_{5'} = \{bdb, dcb, ccb\}$. Correspondingly, the shift function is changed to $t(3) = t(5) = 3$ and $t(3') = t(5') = 2$. Using this alternative automaton, when in state 3 or state 5, the resulting shift is 3 symbols instead of 2.

In order to take advantage of this type of improvement, it is necessary to use finite automata which distinguish as much as possible between different strings in their left languages (as in the way we split state 3 to distinguish between the two in-transitions in the original automaton). The tradeoff is that the alternative automaton requires more storage space. It would be interesting to know quantitatively what the tradeoffs are between automaton size and shift distances.

5.8 Conclusions

We have achieved our aim of deriving an efficient generalized Boyer-Moore type pattern matching algorithm for regular languages. The stepwise derivation began with a simple, intuitive first algorithm; a finite automaton was introduced to make the implementation practical. The idea of shift distances greater than one symbol (as in the Boyer-Moore and Commentz-Walter algorithms) was introduced. The use of predicate weakening was instrumental in deriving a practical approximation to the ideal shift distance.

Using a structural property of finite automata, the approximation was shown to be the composition of several functions, all but two of which are easily computed. The remaining two functions are the Commentz-Walter shift functions; an algorithm computing these functions has previously been derived with correctness arguments in [WZ92, WZ95].

A Boyer-Moore algorithm was derived as a special case of our algorithm, showing our algorithm to be a truly generalized pattern matching algorithm.

Chapter 6

FA construction algorithms

In this chapter, we present a taxonomy of functions and algorithms (collectively called *constructions*) which construct a finite automaton from a regular expression. The taxonomy presented here reuses some ideas of the earlier taxonomy given in [Wat93a]. The taxonomy will be presented in a less rigorous fashion than the one in Chapter 4. We present a number of definitions and theorems without full proofs; in those cases, the proofs are omitted for brevity.

6.1 Introduction and related work

The finite automaton construction problem is: given a regular expression E, construct $M \in FA$ (or, in some cases, $M \in DFA$) such that $\mathcal{L}_{RE}(E) = \mathcal{L}_{FA}(M)$.

The central idea behind our taxonomy is as follows: we present a 'canonical' construction whose correctness is easily seen. The states in a canonically constructed *FA* have internal structure, encoding information — namely the left language and the right language of the state. In subsequent constructions, we may discard some of the information (perhaps identifying states) or we may encode the states (for example, as regular expressions) for greater efficiency. Additionally, we can apply function rem_ε to remove ε-transitions, or the subset construction yielding a *DFA*. We use these techniques to arrive at all of the well-known constructions.

In order to clearly present and derive the constructions, they are given as functions (instead of as algorithms). Since most of the constructions given in the literature (especially in text-books) are given as algorithms, we also select a few and present them again as imperative programs in Section 6.9.

The main finite automata constructions included in the taxonomy are:

- A variant of the Thompson construction as presented in [Thom68], appearing as Construction 6.15. This construction, which is known in this dissertation as the 'canonical' construction, produces a (possibly nondeterministic) finite automaton (possibly with ε-transitions). It is based upon the concept of 'items' which is borrowed from LR parsing [Knut65].

- The ε-*free* item set construction, appearing as Construction 6.19. This construction is the composition of ε-transition removal with the canonical construction.

- The deterministic item set construction, given on page 156 (as Construction (REM-ε, SUBSET, USE-S)) and as Algorithm 6.83. The construction produces a deterministic finite automaton and does not appear in the literature.

- An improvement of the item set construction. This construction (appearing on page 158 (as Construction (REM-ε, SUBSET, USE-S, WFILT)) and not appearing in the literature) produces a deterministic finite automaton, and is also based upon the *DFA* item set construction. Furthermore, it is an improvement of DeRemer's construction (mentioned below). A variant (given in Section 6.9) is also related to the Aho-Sethi-Ullman deterministic finite automaton construction.

- The DeRemer construction as presented in [DeRe74]. This construction (page 159 as Construction (REM-ε, SUBSET, USE-S, XFILT)) produces a deterministic finite automaton. In this chapter, it is derived from the item set construction, although DeRemer made use of LR parsing in his derivation.

- The Berry-Sethi construction as presented in [BS86, Glus61, MY60]. This construction appears as Construction 6.39. It is implicitly given by Glushkov [Glus61] and McNaughton and Yamada [MY60], where it is used as the nondeterministic finite automaton construction underlying a deterministic finite automaton construction. Berry and Sethi explicitly present this algorithm in [BS86], where they relate it to Brzozowski's *DFA* construction [Brzo64].

- The McNaughton-Yamada-Glushkov construction as presented in [MY60, Glus61]. This construction (Construction 6.44) produces a *DFA*.

- The dual of the Berry-Sethi construction. This construction (Construction 6.65) is the 'mirror image' of the Berry-Sethi construction. A variant of this construction was also mentioned in passing by Aho, Sethi, and Ullman [ASU86, Example 3.22, p. 140]; that variant appears in this chapter as Construction 6.68.

- The Aho-Sethi-Ullman construction as presented in [ASU86, Alg. 3.5, Fig. 3.44]. This construction (Construction 6.69 and Algorithm 6.86) produces a deterministic finite automaton. It is, in a sense, the 'mirror image' of a variant of the McNaughton-Yamada-Glushkov construction.

- The Antimirov construction as presented in [Anti94, Anti95]. This construction (Construction 6.55 in this dissertation) yields an ε-*free* finite automaton.

- The Brzozowski construction as presented in [Brzo64]. This construction (Construction 6.57) gives a deterministic finite automaton.

The resulting taxonomy graph is shown in Figure 6.1. This graph will be reproduced in each of the following sections of this chapter, indicating the subpart of the taxonomy considered in the section. Later in this section, we give a list of algorithm and problem details introduced and used in this chapter.

Another taxonomy of constructions (also by the present author) was given in [Wat93a]. That taxonomy is structured around the idea of Σ-algebras, deriving a few constructions that are not covered here. It is also much larger (textually) than the one presented in this chapter.

This chapter is structured as follows:

- In Section 6.2, we introduce 'items' and an alternative (tree-based) definition of regular expressions.
- Section 6.3 presents the 'canonical' finite automata construction.
- Section 6.4 is the beginning of the presentation of the ε-*free* constructions.
- In Section 6.5, we introduce a method of encoding the sets of items which are used as states in Section 6.4.
- Section 6.6 introduces some constructions which use derivatives of regular expressions for states, providing an encoding of the constructions of Section 6.4.
- Section 6.7 gives the duals of some of the constructions given in preceding sections.
- The constructions given in Sections 6.5 and 6.7 make use of auxiliary functions and sets. In Section 6.8 we consider the precomputation of these functions and sets.
- Section 6.9 gives the imperative programs which implement some of the constructions.
- Section 6.10 contains the conclusions of this chapter.

The following is a list the algorithm and problem details with a short description of each:

REM-ε (Algorithm detail 6.18) ε-transition removal function rem_ε is composed onto a construction.

USE-S (Algorithm detail 6.21) Start-unreachable state removal function $useful_s$ is composed onto a construction.

SUBSET (Algorithm detail 6.22) The subset construction is composed onto a construction.

FILT (Algorithm detail 6.25) A filter is used to remove redundant items. Particular filters are:

WFILT (Algorithm detail 6.26) Function $\mathcal{W}$ is used as a filter.

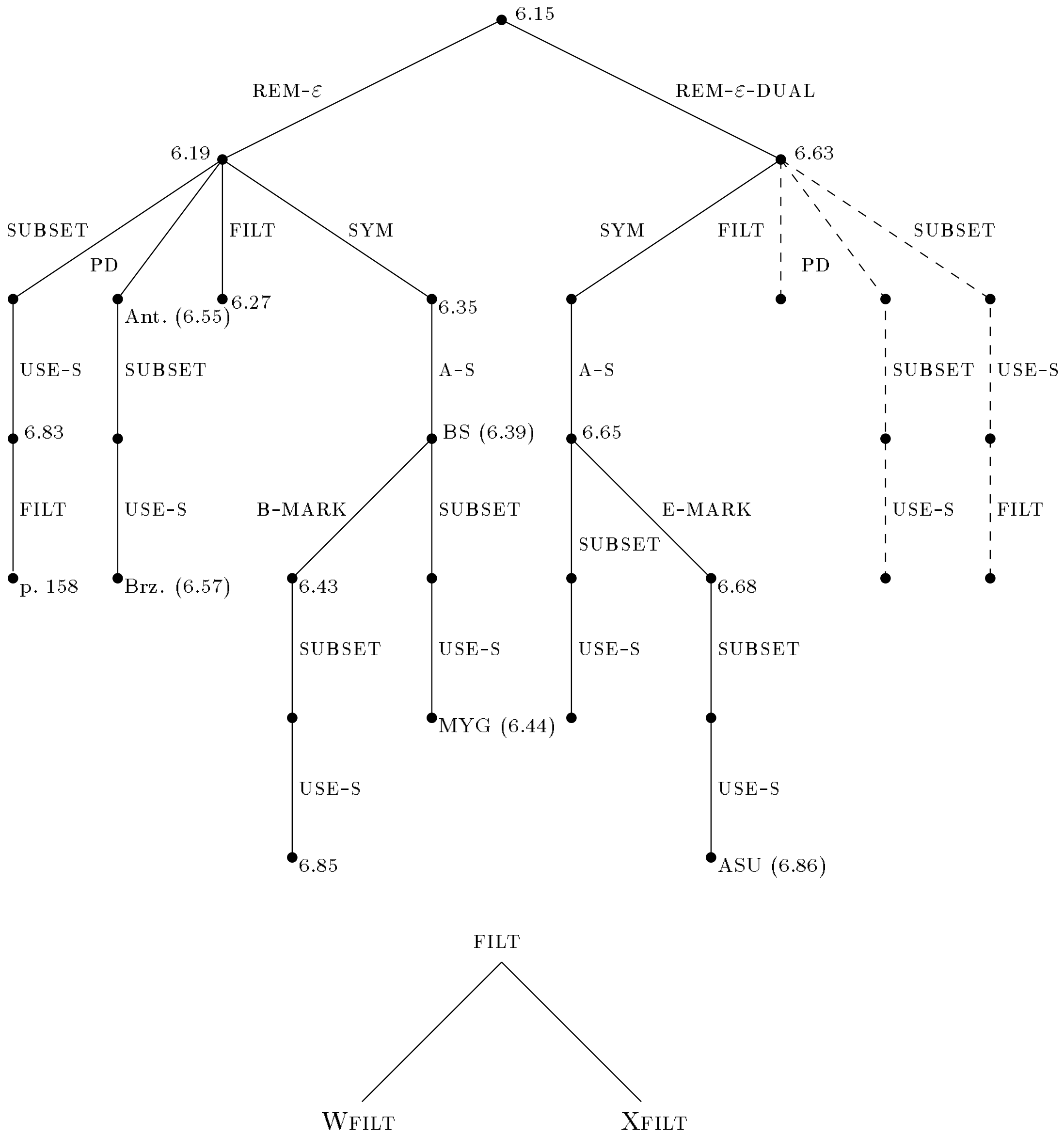


Figure 6.1: A taxonomy of finite automata construction algorithms. The larger graph represents the main part of the taxonomy, while the smaller graph represents the two instantiations of the FILT detail that are discussed in this dissertation. The numbers appearing at some of the vertices correspond to the algorithm or construction numbers in the text of this chapter. In some cases, the algorithm is not presented explicitly, and the page number is given instead. The use of duality is clearly shown by the symmetry in the graph. The algorithms in the dashed-line subtree (on the right of the graph) are not treated in this dissertation, since they are the duals of algorithms in the left half and it is not clear that the duals would be more efficient or enlightening.

XFILT (Algorithm detail 6.31) Function $\mathcal{X}$ is used as a filter.

SYM (Algorithm detail 6.33) States (sets of items) are encoded by elements of $Symnodes_E$.

A-S (Algorithm detail 6.38) Auxiliary sets $Follow_E$, $First_E$, $Last_E$, and predicate *Null* are used to encode the states and transitions.

B-MARK (Algorithm detail 6.42) Certain constructions are greatly simplified by prepending a symbol (the 'begin-marker', usually written \$) to the input regular expression E. In the dual constructions, an 'end-marker' is appended instead.

PD (Algorithm detail 6.54) Partial derivatives are used to encode the states and transition relation in a construction.

REM-ε-DUAL (Algorithm detail 6.62) The dual of the ε-transition removal function is composed onto a construction.

E-MARK (Algorithm detail 6.67) As with detail B-MARK, but an end-marker is appended to the regular expression. This detail is used for the dual constructions.

6.2 Items and an alternative definition of *RE*

This chapter relies upon the tree definitions given in Section 2.5. In order to present the canonical construction, we need the ability to refer to a particular subexpression of an RE, and the place where it occurs in the RE. For example, in $(a \cup a) \in RE$, we would like to be able to distinguish between the two a subexpressions. To do this, we view an RE as synonymous with its corresponding parse tree.

Definition 6.1 (Regular expressions): Define the set of regular expressions over alphabet V as a set of trees. We take $W = (V \cup \{\emptyset, \varepsilon, \cup, \cdot, *, +, ?\}, r)$ as ranked alphabet where symbols in the set $V \cup \{\emptyset, \varepsilon\}$ are all nullary, $\{*, +, ?\}$ are all unary, and $\{\cup, \cdot\}$ are all binary. We can then define the set of regular expressions RE to be the set $Trees(W)$. $\square$

Using this definition, we refer to the set of nodes of the parse tree of $E \in RE$ as $dom(E)$, and the operator at node e is $E(e)$. The two a subexpressions above, in $a \cup a$, would be nodes 1 and 2 respectively.

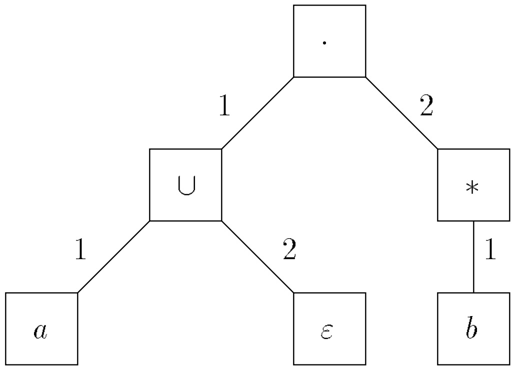


Figure 6.2: Tree view of an example *RE*.

Functions on regular expressions, such as $\mathcal{L}_{RE}$, can by extended to work with the tree versions of regular expressions. Given this equivalence, we will use the two views of regular expressions interchangeably; which one is used at any given time will be clear from the context.

Example 6.2 (Regular expression): We use the regular expression $E = (a \cup \varepsilon) \cdot (b^*)$ as our example. In the tree view, we have $dom(E) = \{0, 1, 2, 1 \diamond 1, 1 \diamond 2, 2 \diamond 1\}$ and $E = \{(0, \cdot), (1, \cup), (2, *), (1 \diamond 1, a), (1 \diamond 2, \varepsilon), (2 \diamond 1, b)\}$. This tree is shown in Figure 6.2. We will use this regular expression in examples throughout this chapter. □

Some of the following definitions are given with respect to a particular $E \in RE$.

We will be considering regular expressions with a dot placed somewhere in the regular expression. Such a regular expression, with the dot, is sometimes called an *item* or a *dotted regular expression*. We now give some formal definitions relating to dottings.

Definition 6.3 (Item): An *item* (or *dotted regular expression*) is a triple (E, n, p) where:

- E is the regular expression.
- $n \in dom(E)$, that is n is a node in the tree interpretation of regular expression E.
- $p \in \{BEF, AFT\}$ is the position of the dot, either *before* or *after* the node n.

We use *DRE* to denote the set of all dotted regular expressions. We can also write an item in a linear form, in which the dot (depicted as $\bullet$, not to be confused with the bullet used for typesetting lists) is embedded in the regular expression itself. This is shown in the example which follows. □

Example 6.4 (Item): Given our regular expression, $(a \cup \varepsilon) \cdot (b^*)$, the following is an item:

$$((a \cup \varepsilon) \cdot (b^*), 1, AFT)$$

We could also write this item as $((a \cup \varepsilon)\bullet) \cdot (b^*)$. The tree form of this item is shown in Figure 6.3. □

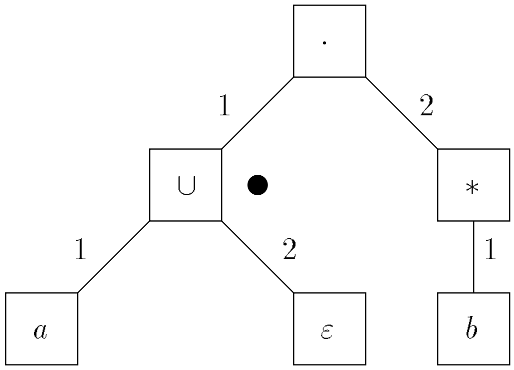


Figure 6.3: Tree form of an item.

Sometimes we will need access to the regular expression underlying a particular item. The following function gives such access.

Definition 6.5 (Function *undot*): Function *undot* defined as $undot(E, n, p) = E$ maps an item to its underlying regular expression. □

We will frequently be considering sets of dottings of a particular regular expression. In this case, we can drop the first component of the dottings (the regular expression) in the set, since they will all be over the same regular expression. This is simply a notational convenience. We will freely alternate between the pair and triple form of items, choosing the form that best suits the application. The following definition is the set of all dottings of a particular regular expression.

Definition 6.6 (Set $Dots_E$): Define the set $Dots_E$ as the set of all items over E. That is,

$$Dots_E = dom(E) \times \{BEF, AFT\}$$

Intuitively, $Dots_E$ is the set of all items D such that $undot(D) = E$. □

Example 6.7 ($Dots_E$): Given our example regular expression, $(a \cup \varepsilon) \cdot (b^*)$, we give the set $Dots_{(a\cup\varepsilon)\cdot(b^*)}$ as (where we have fully parenthesized the regular expression)

$$
\begin{array}{l}
(\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)) \\
((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)) \\
(((\bullet a) \cup (\varepsilon)) \cdot ((b)^*)) \\
(((a\bullet) \cup (\varepsilon)) \cdot ((b)^*)) \\
(((a) \cup (\bullet\varepsilon)) \cdot ((b)^*)) \\
(((a) \cup (\varepsilon\bullet)) \cdot ((b)^*)) \\
(((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)) \\
(((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)) \\
(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)) \\
(((a) \cup (\varepsilon)) \cdot ((b\bullet)^*)) \\
(((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)) \\
(((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)
\end{array}
$$

□

For every item, we can give a pair of regular expressions denoting (respectively) the language to the left and to the right of the dot. The regular expressions are given by the functions defined as follows.

Definition 6.8 (Functions $\overleftarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}}$): We define functions $\overleftarrow{\mathcal{E}}, \overrightarrow{\mathcal{E}} \in DRE \longrightarrow RE$ giving the regular expressions denoting languages to the left and to the right of the dot respectively. The definition of $\overrightarrow{\mathcal{E}}$ is inductive on the structure of items (we take some notational shortcuts in the definition), assuming that F, F_0, and F_1 are regular expressions:

$$
\begin{array}{lcl}
\overrightarrow{\mathcal{E}}(F, 0, BEF) & = & F \\
\overrightarrow{\mathcal{E}}(F, 0, AFT) & = & \varepsilon \\
\overrightarrow{\mathcal{E}}(F_0 \cup F_1, 1 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F_0, w, p) \\
\overrightarrow{\mathcal{E}}(F_0 \cup F_1, 2 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F_1, w, p) \\
\overrightarrow{\mathcal{E}}(F_0 \cdot F_1, 1 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F_0, w, p) \cdot F_1 \\
\overrightarrow{\mathcal{E}}(F_0 \cdot F_1, 2 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F_1, w, p) \\
\overrightarrow{\mathcal{E}}(F^*, 1 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F, w, p) \cdot F^* \\
\overrightarrow{\mathcal{E}}(F^+, 1 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F, w, p) \cdot F^* \\
\overrightarrow{\mathcal{E}}(F^?, 1 \diamond w, p) & = & \overrightarrow{\mathcal{E}}(F, w, p)
\end{array}
$$

(It is important to note that the right sides in the lines above are regular expressions.) Function $\overleftarrow{\mathcal{E}}$ can be defined similarly — however, the definition is not needed explicitly in this dissertation. □

Example 6.9 ($\overrightarrow{\mathcal{E}}$): We present an example of only $\overrightarrow{\mathcal{E}}$ (using the informal notation for items):

$$
\begin{array}{ll}
 & \overrightarrow{\mathcal{E}}((a \cup \varepsilon) \bullet \cdot (b^*)) \\
= & \quad \{ \overrightarrow{\mathcal{E}}(D \cdot E) \text{ rule} \} \\
 & \overrightarrow{\mathcal{E}}((a \cup \varepsilon)\bullet) \cdot (b^*) \\
= & \quad \{ \overrightarrow{\mathcal{E}}(E\bullet) \text{ rule} \} \\
 & \varepsilon \cdot (b^*)
\end{array}
$$

Note that the result is a regular expression. □

Definition 6.10 (Relation $\mathcal{D}$): We define a binary relation $\mathcal{D}$ on $Dots_E$. Before defining $\mathcal{D}$, we define a larger binary relation (called $\bar{\mathcal{D}}$) on items. $\bar{\mathcal{D}}$ is the smallest relation such that:

1. If $F_0, F_1 \in RE$, then (here we use infix notation for relation $\bar{\mathcal{D}}$):

$$
\begin{array}{lll}
\bullet\varepsilon & \bar{\mathcal{D}} & \varepsilon\bullet \\
\bullet(F_0 \cdot F_1) & \bar{\mathcal{D}} & (\bullet F_0) \cdot F_1 \\
(F_0\bullet) \cdot F_1 & \bar{\mathcal{D}} & F_0 \cdot (\bullet F_1) \\
F_0 \cdot (F_1\bullet) & \bar{\mathcal{D}} & (F_0 \cdot F_1)\bullet \\
\bullet(F_0 \cup F_1) & \bar{\mathcal{D}} & (\bullet F_0) \cup F_1 \\
\bullet(F_0 \cup F_1) & \bar{\mathcal{D}} & F_0 \cup (\bullet F_1) \\
(F_0\bullet) \cup F_1 & \bar{\mathcal{D}} & (F_0 \cup F_1)\bullet \\
F_0 \cup (F_1\bullet) & \bar{\mathcal{D}} & (F_0 \cup F_1)\bullet \\
\bullet(F_0^*) & \bar{\mathcal{D}} & (\bullet F_0)^* \\
\bullet(F_0^*) & \bar{\mathcal{D}} & (F_0^*)\bullet \\
(F_0\bullet)^* & \bar{\mathcal{D}} & (\bullet F_0)^* \\
(F_0\bullet)^* & \bar{\mathcal{D}} & (F_0^*)\bullet \\
\bullet(F_0^+) & \bar{\mathcal{D}} & (\bullet F_0)^+ \\
(F_0\bullet)^+ & \bar{\mathcal{D}} & (\bullet F_0)^+ \\
(F_0\bullet)^+ & \bar{\mathcal{D}} & (F_0^+)\bullet \\
\bullet(F_0^?) & \bar{\mathcal{D}} & (\bullet F_0)^? \\
\bullet(F_0^?) & \bar{\mathcal{D}} & (F_0^?)\bullet \\
(F_0\bullet)^? & \bar{\mathcal{D}} & (F_0^?)\bullet
\end{array}
$$

   Note that the pair $(\bullet\emptyset, \emptyset\bullet)$ does not appear in relation $\bar{\mathcal{D}}$.

2. If $F \in RE$ and $D_0, D_1 \in DRE$ such that $(D_0, D_1) \in \bar{\mathcal{D}}$, then:

   (a) $(F \cup D_0, F \cup D_1) \in \bar{\mathcal{D}}$, $(D_0 \cup F, D_1 \cup F) \in \bar{\mathcal{D}}$, $(F \cdot D_0, F \cdot D_1) \in \bar{\mathcal{D}}$, and $(D_0 \cdot F, D_1 \cdot F) \in \bar{\mathcal{D}}$.

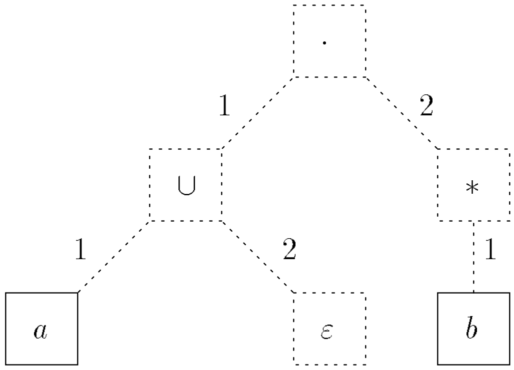


Figure 6.4: The *Symnodes* appear as solid nodes in the tree.

(b) $(D_0^*, D_1^*) \in \bar{\mathcal{D}}$, $(D_0^+, D_1^+) \in \bar{\mathcal{D}}$, and $(D_0^?, D_1^?) \in \bar{\mathcal{D}}$.

These are known as *closure* rules.

Note that $(D_0, D_1) \in \bar{\mathcal{D}} \Rightarrow undot(D_0) = undot(D_1)$. (That is, all dottings related by $\bar{\mathcal{D}}$ have the same underlying regular expression.)

For $E \in RE$, we define $\mathcal{D}$ to be the subset of $\bar{\mathcal{D}}$ on dottings of E. More formally, define

$$\mathcal{D} = \bar{\mathcal{D}} \cap (Dots_E \times Dots_E)$$

Since $\mathcal{D}$ depends upon E, we sometimes write $\mathcal{D}_E$. Note that, although $\bar{\mathcal{D}}$ is an infinite relation, $\mathcal{D}_E$ is a finite one since it is a binary relation on $Dots_E$ (which is a finite set). The relation $\mathcal{D}$ is called the *dot movement relation.* □

We do not present an example of relation $\mathcal{D}$ since one is implicitly included in Example 6.17.

Definition 6.11 (Set *Symnodes*): We define the set

$$Symnodes_E = \{\, e \mid e \in dom(E) \wedge E(e) \in V \,\}$$

That is, $Symnodes_E$ is the set of all nodes in E having labels in V. □

Example 6.12 (*Symnodes*): Given our example regular expression:

$$Symnodes_{(a \cup \varepsilon)\cdot(b^*)} = \{1 \diamond 1, 2 \diamond 1\}$$

The set of symbol nodes are also shown as solid nodes in Figure 6.4. □

Definition 6.13 (Relation $\mathcal{T}$): For each $a \in V$, we define a binary relation on $Dots_E$, called $\mathcal{T}_a$ defined as:

$$\mathcal{T}_a = \{\, ((e, BEF), (e, AFT)) \mid e \in Symnodes_E \wedge E(e) = a \,\}$$

The $\mathcal{T}$ relations allow the dot to hop over symbol nodes in the tree E. All of the $\mathcal{T}_a$ can also be combined into a ternary relation $\mathcal{T} \subseteq Dots_E \times V \times Dots_E$ (note that we have inserted the third component, V, as the middle component). Sometimes we write $\mathcal{T}_E$ to indicate that the relation $\mathcal{T}$ depends upon E. □

Example 6.14 ($\mathcal{T}$): Given our example regular expression, $(a \cup \varepsilon) \cdot (b^*)$, the relation $\mathcal{T}$ can be expressed as the two relations $\mathcal{T}_a$ containing the single pair

$$(((\bullet a) \cup \varepsilon) \cdot ((b)^*), ((a\bullet) \cup \varepsilon) \cdot ((b)^*))$$

and $\mathcal{T}_b$ containing the single pair

$$(((a) \cup \varepsilon) \cdot ((\bullet b)^*), ((a) \cup \varepsilon) \cdot ((b\bullet)^*))$$

□

6.3 A canonical construction

In this section, we present a 'canonical' finite automata construction. The states of the constructed finite automaton will contain information — encoding the left and right languages of the states. In order to encode the information, each state will be an item — the dot in the item denotes a language to the left and to the right of the dot. The dot movement relation $\mathcal{D}$ will be used as the ε-transition relation, while the relation $\mathcal{T}$ will be used as the symbol transition relation.

Using relations $\mathcal{D}$ and $\mathcal{T}$, we can give our canonical construction as follows.

Construction 6.15 (): We define a *canonical* FA construction $CA \in RE \longrightarrow FA$ as

$$CA(E) = (Dots_E, V, \mathcal{T}_E, \mathcal{D}_E, \{\bullet E\}, \{E\bullet\})$$

This construction is symmetrical. □

This construction is also called Construction () (the empty sequence of details) since it will be used as the root of our taxonomy graph.

Remark 6.16: Notice that all states either have an in-transition on ε or on a symbol in V, but not both (and similarly with the out-transitions). □

Example 6.17 (Construction CA): Given our example regular expression $(a \cup \varepsilon) \cdot (b^*)$, we refer back to Example 6.7 for set of states of $CA((a \cup \varepsilon) \cdot (b^*))$. The resulting FA is shown in Figure 6.5. The states in the figure are numbered as follows:

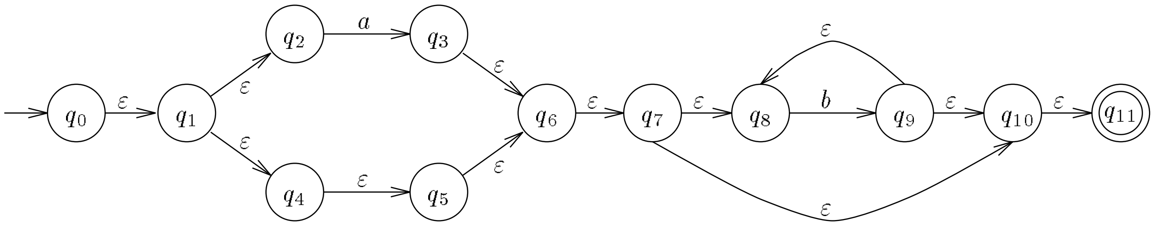


Figure 6.5: Automaton $CA((a \cup \varepsilon) \cdot (b^*))$.

$$
\begin{array}{ll}
q_0 & (\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)) \\
q_1 & ((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)) \\
q_2 & (((\bullet a) \cup (\varepsilon)) \cdot ((b)^*)) \\
q_3 & (((a\bullet) \cup (\varepsilon)) \cdot ((b)^*)) \\
q_4 & (((a) \cup (\bullet\varepsilon)) \cdot ((b)^*)) \\
q_5 & (((a) \cup (\varepsilon\bullet)) \cdot ((b)^*)) \\
q_6 & (((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)) \\
q_7 & (((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)) \\
q_8 & (((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)) \\
q_9 & (((a) \cup (\varepsilon)) \cdot ((b\bullet)^*)) \\
q_{10} & (((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)) \\
q_{11} & (((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)
\end{array}
$$

□

In Example 6.17, we can see that states q_0 and q_{11} are redundant — they can be merged with states q_1 and q_{10} respectively. Some other variants of Thompson's construction (see, for example, [AU92, HU79, Wood87]) take this (or other) redundancy into account and are able to produce smaller automata for some regular expressions.

The correctness of the canonical construction follows from the fact that $\overleftarrow{\mathcal{L}}_{CA(E)}(e) = \mathcal{L}_{RE}(\overleftarrow{\mathcal{E}}(e))$ and $\overrightarrow{\mathcal{L}}_{CA(E)}(e) = \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}(e))$ (i.e. the left language of a particular state — a dotting of E — is equal to the language left of the dot, and likewise for the right languages). For example, consider the right language of state q_9 of Example 6.17. By inspecting the automaton, we see that $\overrightarrow{\mathcal{L}}_{CA(E)}(q_9) = \{b\}^*$. Considering the dottings (the fact that $q_9 = ((a \cup \varepsilon) \cdot (b\bullet)^*))$, we obtain:

$$
\begin{array}{ll}
 & \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}((a \cup \varepsilon) \cdot (b\bullet)^*)) \\
= & \quad \{ \overrightarrow{\mathcal{E}} \text{ rule for } \cdot \} \\
 & \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}((b\bullet)^*)) \\
= & \quad \{ \overrightarrow{\mathcal{E}} \text{ rule for } * \}
\end{array}
$$

$$
\begin{array}{ll}
& \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}(b\bullet)\cdot b^*) \\
= & \quad \{ \overrightarrow{\mathcal{E}} \text{ rule for } E\bullet \} \\
& \mathcal{L}_{RE}(\varepsilon \cdot b^*) \\
= & \quad \{ \text{definition of } \mathcal{L}_{RE} \} \\
& \{b\}^*
\end{array}
$$

For the start state, we have $\overrightarrow{\mathcal{L}}_{CA(E)}(\bullet E) = \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}(\bullet E)) = \mathcal{L}_{RE}(E)$, as desired. The left and right language information is encoded in the states of the finite automaton as the language to the left and the language to the right of the dot.

6.4 ε-*free* constructions

In this section, we present the first of our ε-*free* constructions. The part of the taxonomy considered here is shown as the solid part in Figure 6.6. We can present the composition $rem_\varepsilon \circ CA$ (see Transformation 2.119 for the definition of rem_ε) as a first ε-*free* construction (one which produces ε-*free* automata). The following algorithm detail makes explicit the fact that we will be using function rem_ε to produce such automata.

Algorithm detail 6.18 (REM-ε): Composing function rem_ε onto a construction is algorithm detail REM-ε. □

Before presenting the composition, we note the following properties (which we will use to give the new set of states) of $(Q, V, T, G, S, F) = CA(E)$:

- For state $q \in Q$, $G^*(q) = \mathcal{D}^*(q)$ and so $G^*(S) = \mathcal{D}^*(\bullet E)$.
- $\{ G^*(q) \mid Q \times V \times \{q\} \cap T \neq \emptyset \} = \{ \mathcal{D}^*(e, AFT) \mid e \in dom(E) \wedge E(e) \in V \} = \{ \mathcal{D}^*(e, AFT) \mid e \in Symnodes_E \}$.

The intuition behind the second property is: the only states in $CA(E)$ with a non-ε in-transition are dottings of E of the form (e, AFT) where the label of node e is a symbol in V (that is, e is an element of $Symnodes_E$). (This follows from the definition of relation $\mathcal{T}$.) Assuming the definition of $CA(E)$ and the context of the **let** clause of the definition of rem_ε (page 29), we calculate the transition set ($T' \subseteq \mathcal{P}(Q) \times V \times \mathcal{P}(Q)$) of $(rem_\varepsilon \circ CA)(E)$ as follows:

$$
\begin{array}{ll}
& T' \\
= & \quad \{ \text{definitions of } CA(E) \text{ and } rem_\varepsilon \} \\
& \{ (q, a, \mathcal{D}^*(r)) \mid (\exists\ p : p \in q : (p, a, r) \in \mathcal{T}) \} \\
= & \quad \{ \text{change of bound variable: } p = (e, BEF) \wedge r = (e, AFT) \wedge E(e) = a \} \\
& \{ (q, a, \mathcal{D}^*(e, AFT)) \mid (e, BEF) \in q \wedge E(e) = a \wedge a \in V \} \\
= & \quad \{ \text{definition of } Symnodes_E\text{; eliminate bound variable } a \} \\
& \{ (q, E(e), \mathcal{D}^*(e, AFT)) \mid (e, BEF) \in q \wedge e \in Symnodes_E \}
\end{array}
$$

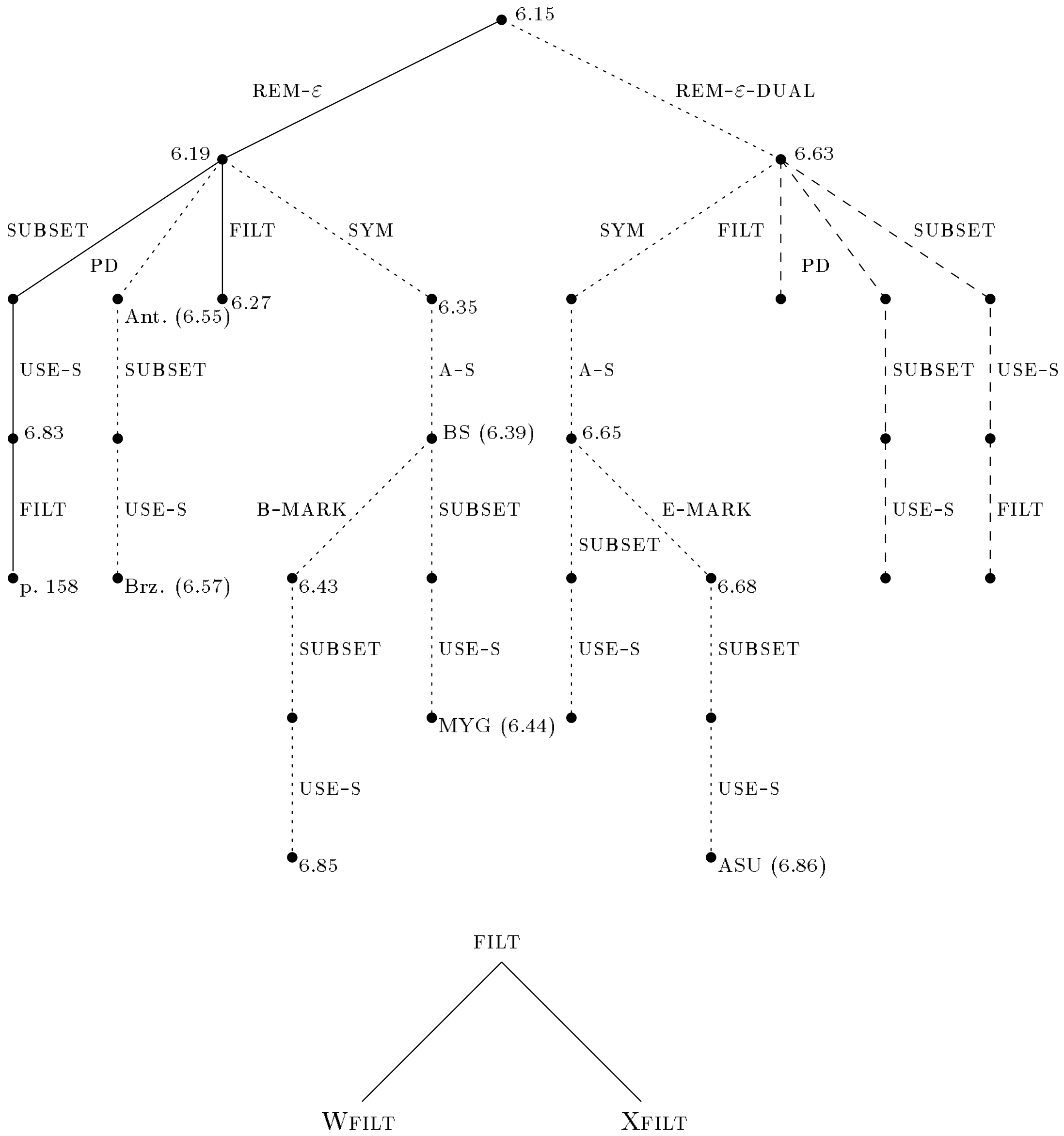


Figure 6.6: The constructions considered in Section 6.4 are shown as solid circles connected by solid lines. The smaller of the two graphs represents the two instantiations of the FILT detail which are discussed in this section.

We can now present the composite construction.

Construction 6.19 (REM-ε): The composition is $(rem_\varepsilon \circ CA)(E) =$

$$\begin{array}{ll}
\textbf{let} & Q' = \{\mathcal{D}^*(\bullet E)\} \cup \{\, \mathcal{D}^*(e, AFT) \mid e \in Symnodes_E \,\} \\
 & T' = \{\, (q, E(e), \mathcal{D}^*(e, AFT)) \mid (e, BEF) \in q \wedge e \in Symnodes_E \,\} \\
 & F' = \{\, f \mid f \in Q' \wedge E\bullet \in f \,\} \\
\textbf{in} & \\
 & (Q', V, T', \emptyset, \{\mathcal{D}^*(\bullet E)\}, F') \\
\textbf{end} &
\end{array}$$

An automaton constructed using this (composite) function has the following properties:

- It has a single start state.
- The single start state has no in-transitions.
- All in-transitions to a state are on the same symbol (in V).

$\square$

This construction is sometimes known as the (nondeterministic) item set construction.

Example 6.20 (Construction (REM-ε)): Recalling $CA((a\cup\varepsilon)\cdot(b^*))$ from Example 6.17, we obtain the following states for automaton $(rem_\varepsilon \circ CA)((a \cup \varepsilon) \cdot (b^*))$.

$$q_0' \quad \begin{array}{l}
(\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)), \\
((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)), \\
(((\bullet a) \cup (\varepsilon)) \cdot ((b)^*)), \\
(((a) \cup (\bullet\varepsilon)) \cdot ((b)^*)), \\
(((a) \cup (\varepsilon\bullet)) \cdot ((b)^*)), \\
(((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)), \\
(((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)
\end{array}$$

$$q_1' \quad \begin{array}{l}
(((a\bullet) \cup (\varepsilon)) \cdot ((b)^*)), \\
(((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)), \\
(((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)
\end{array}$$

$$q_2' \quad \begin{array}{l}
(((a) \cup (\varepsilon)) \cdot ((b\bullet)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)), \\
(((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)
\end{array}$$

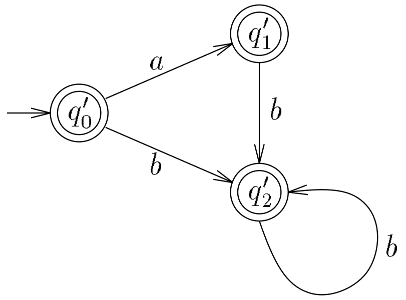


Figure 6.7: Automaton $(rem_\varepsilon \circ CA)((a \cup \varepsilon) \cdot (b^*))$

The structure of each state is more easily understood if we consider the *FA* given in Example 6.17. Note that (our new state) q'_0 is the set of all states (in Example 6.17) which are reachable (by ε-transitions) from q_0 — namely the set $\{q_0, q_1, q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{11}\}$. Similarly, q'_1 is the set of states reachable from q_3 (that is, $q'_1 = \{q_3, q_6, q_7, q_8, q_{10}, q_{11}\}$) and q'_2 is the set of states reachable from q_9 ($q'_2 = \{q_9, q_8, q_{10}, q_{11}\}$).

The resulting automaton is shown in Figure 6.7. □

We could also present a start-reachability version of this construction. Such a construction would make use of the following algorithm detail.

Algorithm detail 6.21 (USE-S)**:** Compose function $useful_s$ (Transformation 2.116) onto a construction, to produce automata with only start-reachable states. □

Using this detail would yield function $useful_s \circ rem_\varepsilon \circ CA$ to give Construction (REM-ε, USE-S). The example *FA* produced by this construction would be isomorphic to the one produced in Example 6.20, since all states of the *FA* in that example are start-reachable. (Note that it is possible for an automaton to have start-unreachable states. As an example, consider the automaton corresponding to the regular expression $\emptyset \cdot a$. This is left to the reader.)

Alternatively, we could apply the subset construction as well as start-reachability. The use of the subset construction is given by the following detail.

Algorithm detail 6.22 (SUBSET)**:** Compose the subset construction (function *subset* — Transformation 2.121) onto a construction, to produce a *DFA*. □

This would yield composite function $useful_s \circ subset \circ rem_\varepsilon \circ CA$ and give Construction (REM-ε, SUBSET, USE-S). This last construction is known as the '(deterministic) item set construction'.

Example 6.23 (Construction (REM-ε, SUBSET, USE-S)**):** Recall the *FA* produced in Example 6.20. That *FA* also happens to be a *DFA*. The composition of $useful_s \circ subset \circ rem_\varepsilon \circ CA$ produces a similar *DFA*, with a sink state. We do not give the state set here in detail. The resulting *DFA* is shown in Figure 6.8. □

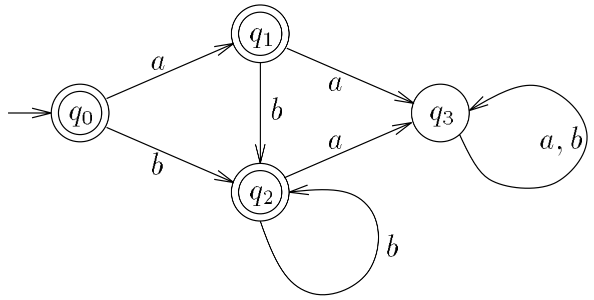


Figure 6.8: Automaton $(useful_s \circ subset \circ rem_\varepsilon \circ CA)((a \cup \varepsilon) \cdot (b^*))$.

6.4.1 Filters

In Construction 6.19, states are sets of items (of the regular expression E). Given state q, when the successor states to q are being constructed, the only information in q that is used are those items $(e, BEF) \in q$ where $e \in Symnodes_E$. This follows from the definition of transition function T' in the **let** clause of that construction. In other words, for any two states p and q such that $p \cap (Symnodes_E \times \{BEF\}) = q \cap (Symnodes_E \times \{BEF\})$, p and q will have the same out-transitions.

The information that is used to determine if q is a final state is the predicate $E\bullet \in q$. So, if the above condition holds on p and q, and $E\bullet \in p \equiv E\bullet \in q$, then the two states will be indistinguishable — that is, their right languages will be the same, and the we can use this information to identify the two states. We can therefore use a filter to remove redundant information from the item sets p and q, allowing them to be identified. We can now define such a filter.

Definition 6.24 (Item set filter $\mathcal{W}$): Given the above discussion, we define the filter function $\mathcal{W}$ on sets of items

$$\mathcal{W}(u) = u \cap ((Symnodes_E \times \{BEF\}) \cup \{E\bullet\})$$

This filter was called $\mathcal{Y}$ when it was first introduced in [Wat93a]. □

We will show later that there are other possible filters. The use of a filter is known as algorithm detail FILT.

Algorithm detail 6.25 (FILT): Usage of an item set filter, such as $\mathcal{W}$. □

This detail is used to indicate that a filter is used. We will also define an algorithm detail for each of the actual filters; the sequence of details describing a construction will contain the name of the actual filter used in place of detail FILT. The use of our first filter is given by the following algorithm detail.

Algorithm detail 6.26 (WFILT): Usage of filter $\mathcal{W}$ is detail WFILT. □

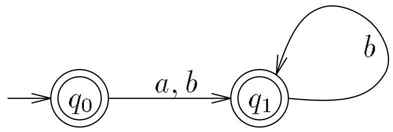


Figure 6.9: *FA* produced by Construction (REM-ε, WFILT)

This results in the following construction:

Construction 6.27 (REM-ε, WFILT): The construction using the filter is:

$$\begin{array}{ll} \textbf{let} & Q' = \{\mathcal{W}(\mathcal{D}^*(\bullet E))\} \cup \{\, \mathcal{W}(\mathcal{D}^*(e, AFT)) \mid e \in Symnodes_E \,\} \\ & T' = \{\, (q, E(e), \mathcal{W}(\mathcal{D}^*(e, AFT))) \mid (e, BEF) \in q \wedge e \in Symnodes_E \,\} \\ & F' = \{\, f \mid f \in Q' \wedge E\bullet \in f \,\} \\ \textbf{in} & \\ & (Q', V, T', \emptyset, \{\mathcal{W}(\mathcal{D}^*(\bullet E))\}, F') \\ \textbf{end} & \end{array}$$

□

The following example shows the type of improvement that filter $\mathcal{W}$ can make.

Example 6.28 (Construction (REM-ε, WFILT)): To show the operation of filter $\mathcal{W}$, we recall the set of states from Example 6.20. We obtain the following filtered states (where the state numbers are taken from Example 6.20):

$$\begin{array}{lcl} \mathcal{W}(q'_0) & = & \{(((\bullet a) \cup (\varepsilon)) \cdot ((b)^*)), (((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), (((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)\} \\ \mathcal{W}(q'_1) & = & \{(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), (((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)\} \\ \mathcal{W}(q'_2) & = & \{(((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), (((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet)\} \end{array}$$

From this, we can see that states q'_1 and q'_2 are identified under Construction (REM-ε, WFILT), becoming state q_1 in the two state *FA* (which is also a *DFA*) shown in Figure 6.9. □

Filters are also interesting (as the following example shows) in the case where the subset construction is used. An example of this will be given in Section 6.9, where we present an imperative program implementing Construction (REM-ε, SUBSET, USE-S, WFILT). That construction was presented as [Wat93a, Constr. 5.82].

Example 6.29 (Construction (REM-ε, SUBSET, USE-S, WFILT)): Construction (REM-ε, SUBSET, USE-S, WFILT) produces a *DFA* which is identical to the one given in Example 6.28, with the addition of a sink state. The resulting *DFA* is shown in Figure 6.10.
□

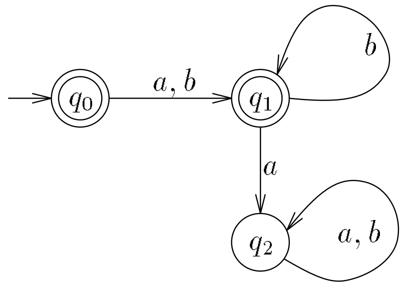


Figure 6.10: *DFA* produced by Construction (REM-ε, SUBSET, USE-S, WFILT).

Discarding more information than $\mathcal{W}$ already discards can be dangerous, since states could then be identified which should not be identified. Of course, we could also devise filters which discard *less* information than $\mathcal{W}$. Such a filter would be called a *safe* filter. One particular safe filter is of historical interest.

Definition 6.30 (DeRemer's filter): Filter $\mathcal{X}$ removes (filters out) the following types of items: any item containing a subexpression of the form $\bullet(E \cup F)$, $\bullet(E^*)$, or $(E\bullet)^*$. Clearly, this filter discards less than the ideal filter $\mathcal{W}$. □

The use of this filter is given by the following algorithm detail.

Algorithm detail 6.31 (XFILT): Usage of filter $\mathcal{X}$. □

Use of this filter would yield Construction (REM-ε, SUBSET, USE-S, XFILT). This construction was originally given by DeRemer in [DeRe74], where an LR parsing algorithm was modified for (compiler) lexical analysis. DeRemer attributes the definition of $\mathcal{X}$ to Earley [Earl70]. Although this construction has been largely ignored in the literature, it was the motivation for deriving the $\mathcal{W}$ filter and eventually the entire taxonomy presented in this chapter. An example of the use of this construction follows.

Example 6.32 (Construction (REM-ε, SUBSET, USE-S, XFILT)): The *DFA* resulting from this construction is isomorphic to the one given in Example 6.23. This example shows that filter $\mathcal{W}$ is frequently more effective (and never less effective) than filter $\mathcal{X}$. See Chapter 14 for data on the effectiveness of the two filters in practice. □

6.5 Encoding Construction (REM-ε)

While the use of filters can make the constructions more efficient in practice, they still produce automata whose states are sets of items. In this section, we explore even more compact representations of the states of the automata. The solid part of the graph in Figure 6.11 indicates the subpart of the taxonomy which is discussed in this section.

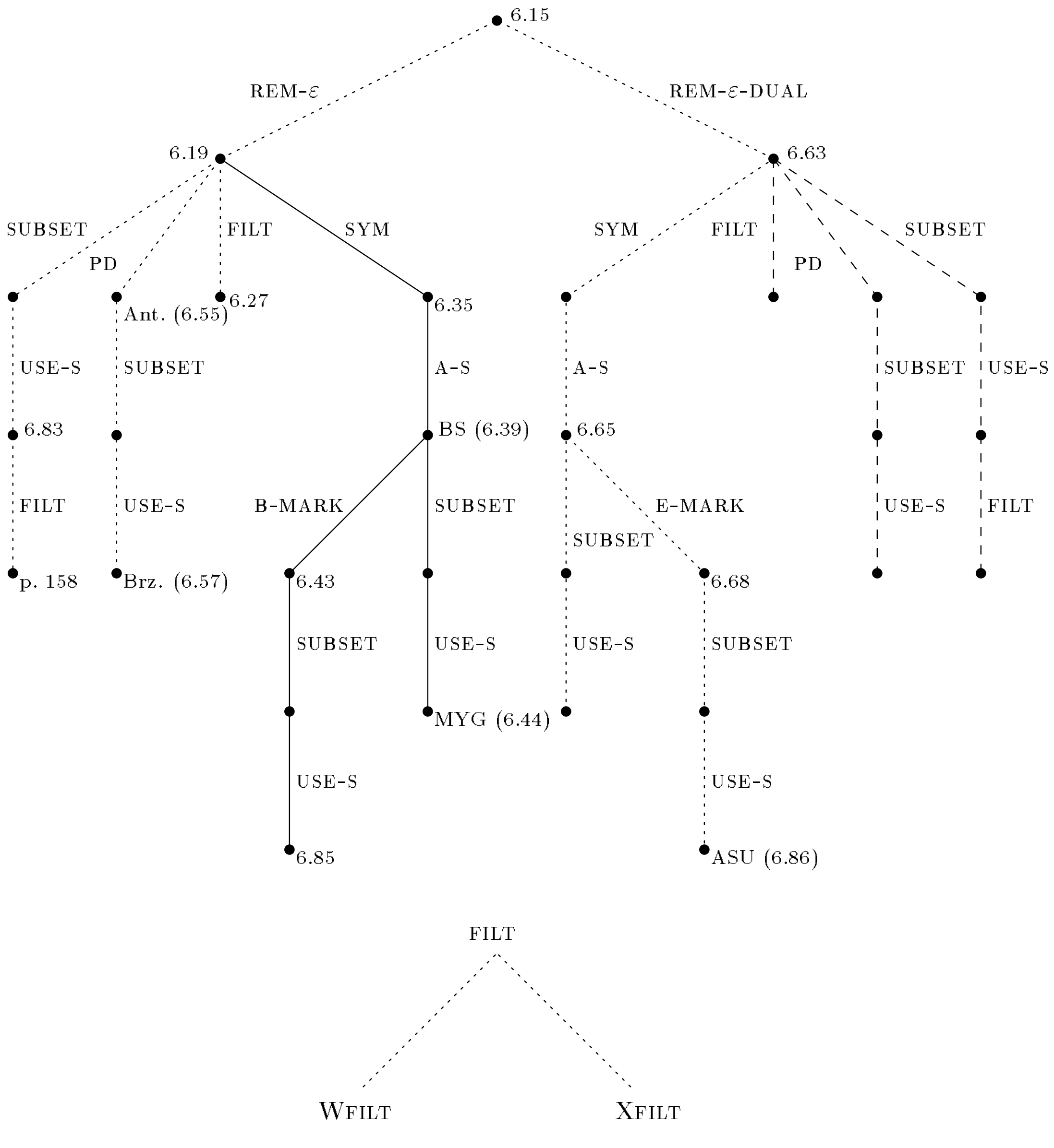


Figure 6.11: The constructions considered in Section 6.5 are shown as solid circles connected by solid lines.

We proceed by trying to characterize the set of states in Construction 6.19. Obviously, we can characterize each element of the set

$$\{\, \mathcal{D}^*(e, AFT) \mid e \in Symnodes_E \,\}$$

by an element in $Symnodes_E$.

To characterize the remaining state $\mathcal{D}^*(\bullet E)$ in Construction 6.19, we introduce a new state s in the **let** clause. We now turn to encoding the set of final states. In the unencoded construction, a state (item set) f is final if and only if $E\bullet \in f$. For an encoded state $e \in Symnodes_E$, this is equivalent to $E\bullet \in \mathcal{D}^*(e, AFT)$. The remaining state ($\mathcal{D}^*(\bullet E)$ in the unencoded construction) is final if and only if $E\bullet \in \mathcal{D}^*(\bullet E)$.

In providing an encoded version of the transition function, we take advantage of the fact that the single start state in Construction 6.19 never has an in-transition. We divide the definition of the transition function into two pieces. The out-transitions from the start state (a new state s, encoding the original start state $\mathcal{D}^*(\bullet E)$) are

$$\{\, (s, E(e), e) \mid (e, BEF) \in \mathcal{D}^*(\bullet E) \wedge e \in Symnodes_E \,\}$$

while the remaining transitions are

$$\{\, (f, E(e), e) \mid (e, BEF) \in \mathcal{D}^*(f, AFT) \wedge e \in Symnodes_E \wedge f \in Symnodes_E \,\}$$

The encoding of some of the states by elements of $Symnodes_E$ constitutes the following algorithm detail.

Algorithm detail 6.33 (SYM)**:** States are encoded by elements of $Symnodes_E$. □

Remark 6.34: This encoding is similar to the LR parsing technique of encoding sets of items by the kernel items [Knut65]. The closure operation can then be used to recover the set from the kernel items. □

Construction 6.35 (REM-ε, SYM)**:** Assuming $E \in RE$, the encoded automaton construction is:

$$
\begin{array}{lll}
\textbf{let} & s \text{ be a new state} & \\
\textbf{in} & & \\
& \textbf{let} & Q = \{s\} \cup Symnodes_E \\
& & T = \{\, (s, E(e), e) \mid (e, BEF) \in \mathcal{D}^*(\bullet E) \wedge e \in Symnodes_E \,\} \\
& & \quad \cup \{\, (f, E(e), e) \mid (e, BEF) \in \mathcal{D}^*(f, AFT) \wedge e, f \in Symnodes_E \,\} \\
& & F = \{\, e \mid E\bullet \in \mathcal{D}^*(e, AFT) \wedge e \in Symnodes_E \,\} \\
& & \quad \cup\ \textbf{if } E\bullet \in \mathcal{D}^*(\bullet E) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi} \\
& \textbf{in} & \\
& & (Q, V, T, \emptyset, \{s\}, F) \\
& \textbf{end} & \\
\textbf{end} & &
\end{array}
$$

For any given $E \in RE$, the FA produced by the above construction is isomorphic to the one produced by Construction 6.19. □

When elements of $Symnodes_E$ are used as states, they are frequently called 'positions' in the literature [ASU86, BS86, B-K93a, Chan92, CP92, tEvG93, MY60]. They can be encoded as integers, with a single traversal of the tree E being used to assign the integers to each node labeled by an element of V. Such an encoding is discussed in Chapter 10.

The above construction can still be made more efficient through the introduction of some auxiliary sets.

Definition 6.36 (Sets *Follow*, *First*, *Last*): Given $E \in RE$, relation $Follow_E$ is a binary relation on $Symnodes_E$:

$$Follow_E = \{ (e, f) \mid ((e, AFT), (f, BEF)) \in \mathcal{D}^* \wedge e, f \in Symnodes_E \}$$

Sets $First_E, Last_E \subseteq Symnodes_E$ are defined as:

$$First_E = \{ e \mid (e, BEF) \in \mathcal{D}^*(\bullet E) \wedge e \in Symnodes_E \}$$

and

$$Last_E = \{ e \mid E\bullet \in \mathcal{D}^*(e, AFT) \wedge e \in Symnodes_E \}$$

Note that *Follow*, *First*, and *Last* can also be viewed as functions with domain *RE* since they depend upon the regular expression E. □

In Section 6.8, we will present examples of these sets and methods for computing them.

In addition to making use of the auxiliary sets, we will also use the fact that $E\bullet \in \mathcal{D}^*(\bullet E) \equiv \varepsilon \in \mathcal{L}_{RE}(E)$. For brevity, we define a predicate on regular expressions.

Definition 6.37 (Predicate *Null*): For $E \in RE$, we define $Null(E) \equiv \varepsilon \in \mathcal{L}_{RE}(E)$. □

Algorithm detail 6.38 (A-S)**:** The use of auxiliary sets $Follow_E, First_E, Last_E$, and predicate *Null* constitute algorithm detail A-S. □

Construction 6.39 (REM-ε, SYM, A-S)**:** Assuming $E \in RE$, we use the auxiliary sets to produce the *FA*:

$$
\begin{array}{lll}
\textbf{let} & s \text{ be a new state} & \\
\textbf{in} & & \\
 & \textbf{let} & Q = \{s\} \cup Symnodes_E \\
 & & T = \{ (s, E(e), e) \mid e \in First_E \} \\
 & & \quad \cup \{ (q, E(e), e) \mid e \in Follow_E(q) \wedge q \in Symnodes_E \} \\
 & & F = Last_E \cup \textbf{if } Null(E) \textbf{ then } \{s\} \textbf{ else } \emptyset \textbf{ fi} \\
 & \textbf{in} & \\
 & & (Q, V, T, \emptyset, \{s\}, F) \\
 & \textbf{end} & \\
\textbf{end} & &
\end{array}
$$

This construction is the Berry-Sethi construction [BS86]. An alternative derivation of this construction is given in the original taxonomy as [Wat93a, Constr. 4.32]. □

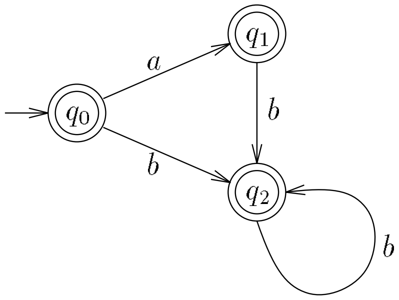


Figure 6.12: *FA* produced by Construction (REM-ε, SYM, A-S).

The following is a short history of the algorithm.

Remark 6.40: The history of this algorithm is somewhat complicated. The following account is given by Brüggemann-Klein [B-K93b]. Glushkov and McNaughton and Yamada simultaneously (and independently) discovered the same *DFA* construction [Glus61, MY60]. Those papers used the same underlying *FA* construction to which they apply the subset construction[1]. Unfortunately, neither of them present the construction without the subset construction explicitly. The underlying *FA* construction was presented in some depth (with correctness arguments) by Berry and Sethi in [BS86, Alg. 4.4]. In their paper, Berry and Sethi also relate the construction to the Brzozowski construction. In this chapter, we adopt the convention that the *FA* construction (without subset construction) is named after Berry and Sethi, while the construction with the subset construction is named after McNaughton, Yamada, and Glushkov. □

Example 6.41 (Construction (REM-ε, SYM, A-S)): The computation of the auxiliary sets is not discussed here — see Section 6.8. The resulting *FA* is shown in Figure 6.12. We briefly mention the state set: start state s is represented in the figure by q_0, $1 \diamond 1$ is represented by q_1, and $2 \diamond 1$ by q_2. Note that the *FA* is always isomorphic to the one given in Figure 6.7 from Example 6.20 since we have only given an encoding of Construction 6.19. □

We could also have presented an algorithm implementing function $useful_s$ composed with the above construction. The algorithm would be Construction (REM-ε, SYM, A-S, USE-S).

In Section 6.8, we will discuss how to compute the auxiliary sets given in Definition 6.36. In the following section, we consider another coding trick that proves to be particularly useful in the preceding construction.

6.5.1 Using begin-markers

In this section, we examine a method of making Construction 6.39 more efficient. One place to improve the efficiency of an implementation of the construction, is to treat state

[1]The underlying construction may actually produce a nondeterministic finite automata.

s (the start state) the same as the other states. As the construction is now written, s is treated in special cases of the definitions of the transition function and the set of final states.

Fortunately, we can achieve this goal by concatenating a single symbol to the left of regular expression E. For example, we concatenate the symbol \$ to give a new regular expression $\$ \cdot E$. In this case, node 1 is the \$ node, while any node q in E is now refered to as $2 \diamond q$ (this is known as *pushing down* the node q). Indeed, this gives us an encoding of all of the states in Construction 6.39. Any state q (where $q \in \{s\} \cup \mathit{Symnodes}_E$, the state set of that construction) is encoded as an element of $\mathit{Symnodes}_{\$ \cdot E}$ by

$$\textbf{if } q = s \textbf{ then } 1 \textbf{ else } 2 \diamond q \textbf{ fi}$$

In our encoded construction, we can now simply use the state set $\mathit{Symnodes}_{\$ \cdot E}$. We also need to provide encoded versions of the transition relation, and the final state set. In some of the following paragraphs, we refer to the context of the inner **let** clause of Construction 6.39.

First, we consider the part of the transition relation that relates to state s:

$$\{ (s, E(e), e) \mid e \in \mathit{First}_E \}$$

Under the new encoding, we note that s (which will be encoded as 1) will have transitions to the encoded states $2 \diamond \mathit{First}_E$. Furthermore, we note that $2 \diamond \mathit{First}_E = \mathit{Follow}_{\$ \cdot E}(1)$. (This is easily seen by the inductive definitions presented in Section 6.8.) We obtain the following to the encoded part of the transition relation for start state s:

$$\{ (1, (\$ \cdot E)(e), e) \mid e \in \mathit{Follow}_{\$ \cdot E}(1) \}$$

The other part of the transition relation remains almost the same as before:

$$\{ (f, (\$ \cdot E)(e), e) \mid e \in \mathit{Follow}_{\$ \cdot E}(f) \}$$

which is identical for the part of the relation involving the start state. We can therefore use the single expression above for the entire transition relation. We can also give an encoded version of the set of final states:

$$(2 \diamond \mathit{Last}_E) \cup \textbf{if } \mathit{Null}(E) \textbf{ then } \{1\} \textbf{ else } \emptyset \textbf{ fi}$$

Since $2 \diamond \mathit{Last}_E = \mathit{Last}_{\$ \cdot E} \setminus \{1\}$ (from the definition of *Last*) and $\mathit{Null}(E) \equiv 1 \in \mathit{Last}_{\$ \cdot E}$, we can rewrite the above expression as

$$\mathit{Last}_{\$ \cdot E} \setminus \{1\} \cup \textbf{if } 1 \in \mathit{Last}_{\$ \cdot E} \textbf{ then } \{1\} \textbf{ else } \emptyset \textbf{ fi}$$

Some simplification yields $\mathit{Last}_{\$ \cdot E}$.

The use of this encoding is given in the following algorithm detail:

Algorithm detail 6.42 (B-MARK)**:** By prepending a symbol (\$ in our case) to the regular expression E, and using the above encoding, we improve Construction 6.39. □

Note that (contrary to popular belief) it does not matter which particular symbol is chosen for the begin-marker. It is common to choose a special symbol so that it is obviously (to the reader) a begin-marker.

This gives us the following construction:

Construction 6.43 (REM-ε, SYM, A-S, B-MARK): Assuming $E \in RE$, construct an *FA* as follows:

let $Q = Symnodes_{\$\cdot E}$
$T = \{ (f, (\$ \cdot E)(e), e) \mid e \in Follow_{\$\cdot E}(f) \}$
$F = Last_{\$\cdot E}$
in
$(Q, V, T, \emptyset, \{1\}, F)$
end

The use of a begin-marker is a rather well-known encoding trick. This particular algorithm appears in the literature as [Wat93a, Constr. 4.38]. □

Since this construction produces *FA*s which are isomorphic to those produced by Construction (REM-ε, SYM, A-S), we do not give an example here. We will see in Section 6.9 how the use of a begin-marker can greatly simplify a construction implementation.

6.5.2 Composing the subset construction

We can also compose function $useful_s \circ subset$ with the two main constructions of this section, Constructions (REM-ε, SYM, A-S) and (REM-ε, SYM, A-S, B-MARK). One of the major constructions is the McNaughton-Yamada-Glushkov construction.

Construction 6.44 (McNaughton-Yamada-Glushkov): Construction (REM-ε, SYM, A-S, SUBSET, USE-S) is the McNaughton-Yamada-Glushkov construction. McNaughton and Yamada originally presented the algorithm in [MY60] while Glushkov independently derives the algorithm in [Glus61]. See Remark 6.40 for a brief history of this construction. □

An imperative algorithm implementing this construction is given in Algorithm 6.84. We present a brief example of an automaton produced by the construction.

Example 6.45 (McNaughton-Yamada-Glushkov): The *DFA* (with sink state) produced for regular expression $(a \cup \varepsilon) \cdot (b^*)$ is shown in Figure 6.13. Note that it is isomorphic to the one shown in Figure 6.8 from Example 6.23. □

The other resulting construction is (REM-ε, SYM, A-S, B-MARK, SUBSET, USE-S), which does not appear in the literature. It is, however, the dual of the Aho-Sethi-Ullman construction (which appears as Construction 6.69 in this chapter). An algorithm implementing it appears as Algorithm 6.85, where it is shown to be more elegant and concise than the McNaughton-Yamada-Glushkov construction. An example *DFA* produced by this construction would be isomorphic to the one given in the preceding example.

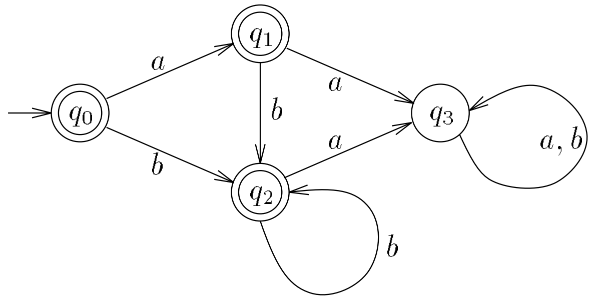


Figure 6.13: *DFA* produced by the McNaughton-Yamada-Glushkov construction.

6.6 An encoding using derivatives

In this section, we consider an alternative encoding of Construction (REM-ε) in which the states are regular expressions. The solid part of the graph in Figure 6.14 indicates the subpart of the taxonomy which is discussed in this section. For brevity, the new constructions presented in this section will be derived in a relatively informal manner.

Recall that we used the following state set in Construction (REM-ε)

$$Q' = \{\mathcal{D}^*(\bullet E)\} \cup \{\, \mathcal{D}^*(e, AFT) \mid e \in Symnodes_E \,\}$$

In Section 6.5, we elected to use the set $Symnodes_E$ to encode the set appearing as the second operand of the $\cup$ operator above. That left us with the problem of encoding the remaining state. In this section, we encode each state q by the (unique) item d such that $q = \mathcal{D}^*(d)$. The above state set would be encoded as:

$$\{\bullet E\} \cup \{\, (e, AFT) \mid e \in Symnodes_E \,\}$$

(Note that the item $\bullet E$ could also have been written as $(E, 0, BEF)$. We will freely mix the two item notations, choosing the most appropriate one for a given application.) Given this, we can provide the following (encoded) version of Construction (REM-ε).

Construction 6.46 (Encoding (REM-ε)): Assuming regular expression E, the following automaton accepts $\mathcal{L}_{RE}(E)$:

let $Q = \{\bullet E\} \cup \{\, (e, AFT) \mid e \in Symnodes_E \,\}$
$\quad T = \{\, (q, E(e), (e, AFT)) \mid (e, BEF) \in \mathcal{D}^*(q) \wedge e \in Symnodes_E \,\}$
$\quad F = \{\, f \mid f \in Q \wedge E\bullet \in \mathcal{D}^*(f) \,\}$
in
$\quad (Q, V, T, \emptyset, \{\bullet E\}, F)$
end

□

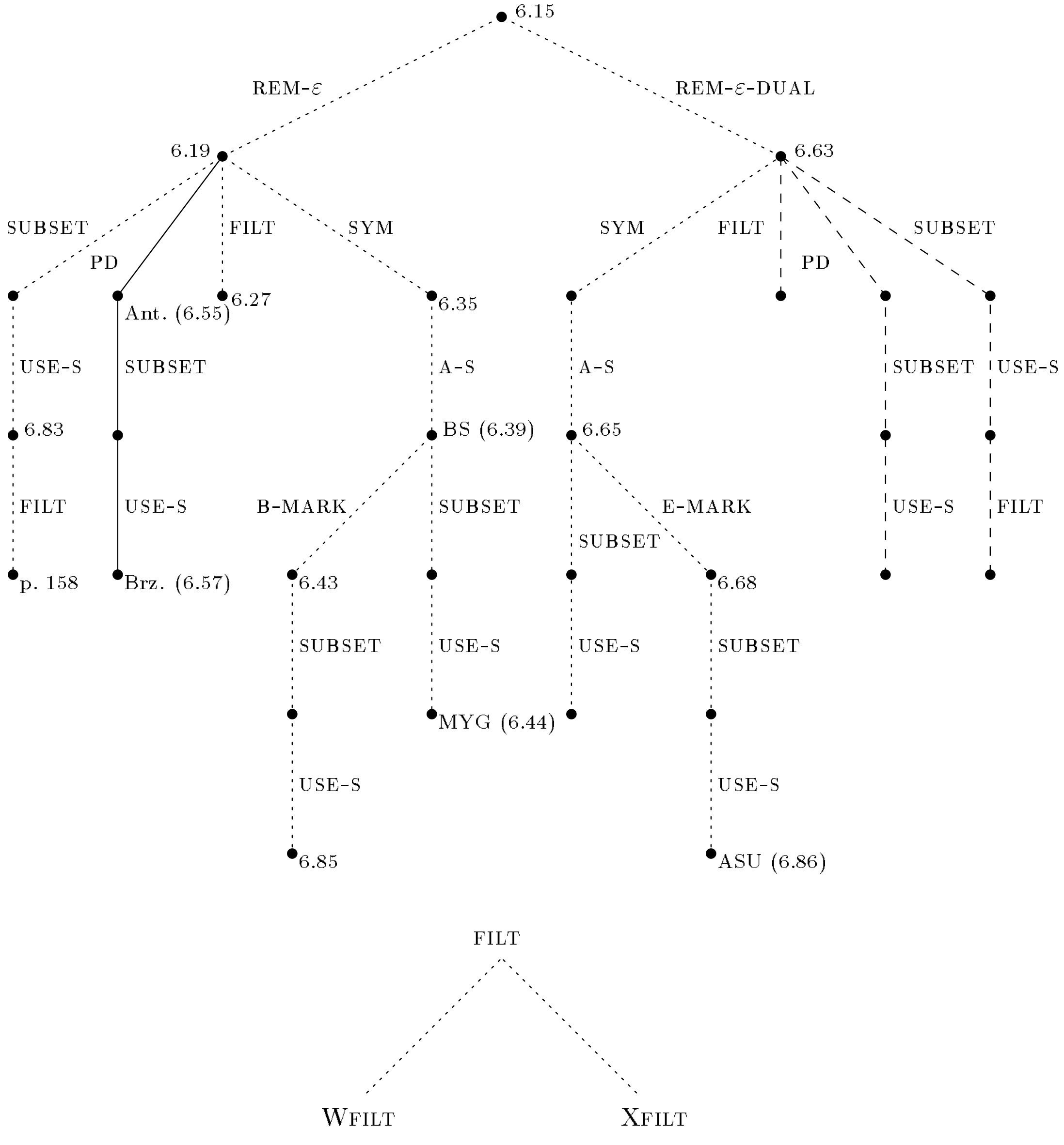


Figure 6.14: The constructions considered in Section 6.6 are shown as solid circles connected by solid lines.

Example 6.47 (Alternative encoding of states): Given our example regular expression, $(a \cup \varepsilon) \cdot b^*$, the three states are $Q = \{\bullet((a \cup \varepsilon) \cdot b^*), (a \bullet \cup \varepsilon) \cdot b^*, (a \cup \varepsilon) \cdot (b \bullet)^*\}$. The automaton is isomorphic to the one given in Example 6.20. □

We will now consider a further encoding of the set of states: using function $\overrightarrow{\mathcal{E}}$ to map each state (item) to a regular expression. In order to facilitate this, we write the transitions using the signature $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ instead of T as a relation over $Q \times V \times Q$. T can be rewritten, using this new signature, as:

$$T(q, a) = \{ (e, AFT) \mid (e, BEF) \in \mathcal{D}^*(q) \wedge E(e) = a \}$$

Note that we also have the property $T(q, a) = \mathcal{T}_a(\mathcal{D}^*(q))$. The change of representation (to regular expressions for states) will be easier with a definition of the transition function which is inductive on the structure of items. The following definition provides such an inductive transition function.

Definition 6.48 (Transition function t): Define transition function $t \in DRE \times V \longrightarrow \mathcal{P}(DRE)$ such that $t(d, a) = \mathcal{T}_a(\mathcal{D}^*(d))$. The definition (which is given without proof) is by induction on the structure of DREs (assuming $E, E_0, E_1 \in RE$ and $a \in V$):

$$t((E, 0, AFT), a) = \emptyset$$

$$t((\varepsilon, 0, BEF), a) = \emptyset$$

$$t((\emptyset, 0, BEF), a) = \emptyset$$

$$t((b, 0, BEF), a) = \textbf{if } a = b \textbf{ then } \{(b, 0, AFT)\} \textbf{ else } \emptyset \textbf{ fi}$$

$$\begin{aligned} t((E_0 \cup E_1, 0, BEF), a) = & \{ (E_0 \cup E_1, 1 \diamond v, p) \mid (E_0, v, p) \in t((E_0, 0, BEF), a) \} \\ & \cup \{ (E_0 \cup E_1, 2 \diamond v, p) \mid (E_1, v, p) \in t((E_1, 0, BEF), a) \} \end{aligned}$$

$$\begin{aligned} t((E_0 \cdot E_1, 0, BEF), a) = & \{ (E_0 \cdot E_1, 1 \diamond v, p) \mid (E_0, v, p) \in t((E_0, 0, BEF), a) \} \\ \cup & \begin{array}{ll} \textbf{if} & (E_0, 0, AFT) \in \mathcal{D}^*(E_0, 0, BEF) \\ \textbf{then} & \{ (E_0 \cdot E_1, 2 \diamond v, p) \mid (E_1, v, p) \in t((E_1, 0, BEF), a) \} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array} \end{aligned}$$

$$t((E^*, 0, BEF), a) = \{ (E^*, 1 \diamond v, p) \mid (E, v, p) \in t(E, 0, BEF) \}$$

$$t((E^+, 0, BEF), a) = \{ (E^+, 1 \diamond v, p) \mid (E, v, p) \in t(E, 0, BEF) \}$$

$$t((E^?, 0, BEF), a) = \{ (E^?, 1 \diamond v, p) \mid (E, v, p) \in t(E, 0, BEF) \}$$

$$t((E_0 \cup E_1, 1 \diamond w, p), a) = \{ (E_0 \cup E_1, 1 \diamond v, p') \mid (E_0, v, p') \in t((E_0, w, p), a) \}$$

$$t((E_0 \cup E_1, 2 \diamond w, p), a) = \{ (E_0 \cup E_1, 2 \diamond v, p') \mid (E_1, v, p') \in t((E_1, w, p), a) \}$$

$$
\begin{array}{rcl}
t((E_0 \cdot E_1, 1 \diamond w, p), a) & = & \{\, (E_0 \cdot E_1, 1 \diamond v, p') \mid (E_0, v, p') \in t((E_0, w, p), a) \,\} \\
& \cup & \begin{array}{ll} \textbf{if} & (E_0, 0, AFT) \in \mathcal{D}^*(E_0, w, p) \\ \textbf{then} & \{\, (E_0 \cdot E_1, 2 \diamond v, p') \mid (E_1, v, p') \in t((E_1, 0, BEF), a) \,\} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array}
\end{array}
$$

$$
t((E_0 \cdot E_1, 2 \diamond w, p), a) \;=\; \{\, (E_0 \cdot E_1, 2 \diamond v, p') \mid (E_1, v, p') \in t((E_1, w, p), a) \,\}
$$

$$
\begin{array}{rcl}
t((E^*, 1 \diamond w, p), a) & = & \{\, (E^*, 1 \diamond v, p') \mid (E, v, p') \in t((E, w, p), a) \,\} \\
& \cup & \begin{array}{ll} \textbf{if} & (E, 0, AFT) \in \mathcal{D}^*(E, w, p) \\ \textbf{then} & \{\, (E^*, 1 \diamond v, p') \mid (E, v, p') \in t((E, 0, BEF), a) \,\} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
t((E^+, 1 \diamond w, p), a) & = & \{\, (E^+, 1 \diamond v, p') \mid (E, v, p') \in t((E, w, p), a) \,\} \\
& \cup & \begin{array}{ll} \textbf{if} & (E, 0, AFT) \in \mathcal{D}^*(E, w, p) \\ \textbf{then} & \{\, (E^+, 1 \diamond v, p') \mid (E, v, p') \in t((E, 0, BEF), a) \,\} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array}
\end{array}
$$

$$
t((E^?, 1 \diamond w, p), a) \;=\; \{\, (E^?, 1 \diamond v, p') \mid (E, v, p') \in t((E, w, p), a) \,\}
$$

Note that the transition relation induced by t is infinite. In our construction, we would only use that portion of t which applies to $Dots_E \times V$ (for our regular expression E). □

Using the function t, we can define the following construction:

$$
\begin{array}{ll}
\textbf{let} & Q = \{\bullet E\} \cup \{\, (e, AFT) \mid e \in Symnodes_E \,\} \\
& T(q, a) = t(q, a) \\
& F = \{\, f \mid f \in Q \wedge E\bullet \in \mathcal{D}^*(f) \,\} \\
\textbf{in} & \\
& (Q, V, T, \emptyset, \{\bullet E\}, F) \\
\textbf{end} &
\end{array}
$$

Example 6.49 (Function t): As an example of the use of function t, consider the out-transitions from (encoded) state $\bullet((a \cup \varepsilon) \cdot b^*)$. Some lengthy calculations show that

$$
\begin{array}{rcl}
t(\bullet((a \cup \varepsilon) \cdot b^*), a) & = & \{((a\bullet) \cup \varepsilon) \cdot b^*\} \\
t(\bullet((a \cup \varepsilon) \cdot b^*), b) & = & \{(a \cup \varepsilon) \cdot (b\bullet)^*\} \\
t(((a\bullet) \cup \varepsilon) \cdot b^*, a) & = & t((a \cup \varepsilon) \cdot (b\bullet)^*, a) \\
& = & \emptyset \\
t(((a\bullet) \cup \varepsilon) \cdot b^*, b) & = & t((a \cup \varepsilon) \cdot (b\bullet)^*, b) \\
& = & \{(a \cup \varepsilon) \cdot (b\bullet)^*\}
\end{array}
$$

□

As our next encoding step, we represent each of the states (in the previous construction) q by the regular expression $\overrightarrow{\mathcal{E}}(q)$. This may identify two states since it is possible that $q_0 \neq q_1$ while $\overrightarrow{\mathcal{E}}(q_0) = \overrightarrow{\mathcal{E}}(q_1)$. Fortunately, since the regular expression $\overrightarrow{\mathcal{E}}(q)$ denotes the right language of state q, only states with the same right language will be identified. (This is one of the cases in which identifying states is safe — it does not alter the language accepted by the automaton.) This encoding can therefore be considered an optimization to the previous construction.

Encoding the transitions of Construction 6.46 yields

$$\{ (\overrightarrow{\mathcal{E}}(q_0), a, \overrightarrow{\mathcal{E}}(q_1)) \mid (q_0, a, q_1) \in T \}$$

In this new automaton, the states will be elements of RE. In a manner analogous to our use of the general (infinite) transition function t, we define a general transition function (which also induces an infinite relation) on RE. The new transition function, $t' \in \overrightarrow{\mathcal{E}}(DRE) \times V \longrightarrow \mathcal{P}(\overrightarrow{\mathcal{E}}(DRE))$ (where $\overrightarrow{\mathcal{E}}(DRE)$ is the image of DRE under $\overrightarrow{\mathcal{E}}$), is defined as follows (in terms of function t):

$$t'(\overrightarrow{\mathcal{E}}(d), a) = \overrightarrow{\mathcal{E}}(t(d, a))$$

We do this for all of the lines in the inductive definition of t (Definition 6.48). For example, consider the line for items of the form $(E_0 \cdot E_1, 0, BEF)$. We obtain the following line in the definition of $t'(\overrightarrow{\mathcal{E}}(E_0 \cdot E_1, 0, BEF), a) =$

$$\begin{array}{l} \overrightarrow{\mathcal{E}}(\{ (E_0 \cdot E_1, 1 \diamond v, p) \mid (E_0, v, p) \in t((E_0, 0, BEF), a) \}) \\ \cup \quad \begin{array}{ll} \textbf{if} & (E_0, 0, AFT) \in \mathcal{D}^*(E_0, 0, BEF) \\ \textbf{then} & \overrightarrow{\mathcal{E}}(\{ (E_0 \cdot E_1, 2 \diamond v, p) \mid (E_1, v, p) \in t((E_1, 0, BEF), a) \}) \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array} \end{array}$$

Recall (from page 162) that $(E_0, 0, AFT) \in \mathcal{D}^*(E_0, 0, BEF) \equiv \varepsilon \in \mathcal{L}_{RE}(E_0) \equiv Null(E_0)$. Using this equivalence, and moving $\overrightarrow{\mathcal{E}}$ into the sets, yields

$$\begin{array}{l} \{ \overrightarrow{\mathcal{E}}(E_0 \cdot E_1, 1 \diamond v, p) \mid (E_0, v, p) \in t((E_0, 0, BEF), a) \} \\ \cup \quad \begin{array}{ll} \textbf{if} & Null(E_0) \\ \textbf{then} & \{ \overrightarrow{\mathcal{E}}(E_0 \cdot E_1, 2 \diamond v, p) \mid (E_1, v, p) \in t((E_1, 0, BEF), a) \} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array} \end{array}$$

Rewriting the sets (using the definition of $\overrightarrow{\mathcal{E}}$) gives

$$\begin{array}{l} \{\ \overrightarrow{\mathcal{E}}(E_0, v, p) \cdot E_1 \mid (E_0, v, p) \in t((E_0, 0, BEF), a)\ \} \\ \cup \begin{array}{ll} \textbf{if} & Null(E_0) \\ \textbf{then} & \{\ \overrightarrow{\mathcal{E}}(E_1, v, p) \mid (E_1, v, p) \in t((E_1, 0, BEF), a)\ \} \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array} \end{array}$$

Using the definition of t gives

$$\begin{array}{l} \overrightarrow{\mathcal{E}}(t((E_0, 0, BEF), a)) \cdot E_1 \\ \cup \begin{array}{ll} \textbf{if} & Null(E_0) \\ \textbf{then} & \overrightarrow{\mathcal{E}}(t((E_1, 0, BEF), a)) \\ \textbf{else} & \emptyset \\ \textbf{fi} & \end{array} \end{array}$$

The above derivation can then be simplified (using the definition of t') to

$$t'(E_0 \cdot E_1, a) = t'(E_0, a) \cdot E_1 \cup \textbf{if } Null(E_0) \textbf{ then } t'(E_1, a) \textbf{ else } \emptyset \textbf{ fi}$$

Had we considered all of the other lines in the definition of t, we would see that a number of them are redundant when rewritten for the definition of t' (they are subsumed by the some of the other lines). This rewriting process is not given in full here; instead, we immediately present the definition of a function which satisfies the requirement on t' (though it has a slightly larger signature).

Definition 6.50 (Function ∂): We define function $\partial \in RE \times V \longrightarrow \mathcal{P}(RE)$. (For historical reasons, we write $\partial_a(E)$ instead of $\partial(E, a)$; this is intended to signify that $\partial_a(E)$ is the *partial derivative* of E with respect to a.) The definition is by structural induction on regular expressions:

$$\begin{array}{lcll} \partial_a(\emptyset) & = & \emptyset & \\ \partial_a(\varepsilon) & = & \emptyset & \\ \partial_a(b) & = & \textbf{if } a = b \textbf{ then } \{\varepsilon\} \textbf{ else } \emptyset \textbf{ fi} & (b \in V) \\ \partial_a(E_0 \cup E_1) & = & \partial_a(E_0) \cup \partial_a(E_1) & \\ \partial_a(E_0 \cdot E_1) & = & \partial_a(E_0) \cdot E_1 \cup \textbf{if } Null(E_0) \textbf{ then } \partial_a(E_1) \textbf{ else } \emptyset \textbf{ fi} & \\ \partial_a(E^*) & = & \partial_a(E) \cdot E^* & \\ \partial_a(E^+) & = & \partial_a(E) \cdot E^* & \\ \partial_a(E^?) & = & \partial_a(E) & \end{array}$$

This function contains t' (when viewed as a transition relation), though the signature is slightly larger since $\overrightarrow{\mathcal{E}}(DRE) \subseteq RE$. □

The definition given here corresponds exactly to the one given by Antimirov in [Anti94, Anti95], though his definitions are derived in a more language-theoretic manner.

Definition 6.51 (Partial derivatives): For regular expression E, we define PD_E to be the image of

$$\{\bullet E\} \cup \{ (e, AFT) \mid e \in Symnodes_E \}$$

under function $\overrightarrow{\mathcal{E}}$. That is (following the definition of $\overrightarrow{\mathcal{E}}$)

$$PD_E = \{E\} \cup \overrightarrow{\mathcal{E}}(\{ (e, AFT) \mid e \in Symnodes_E \})$$

Elements of the set PD_E are the *partial derivatives* of E. □

Example 6.52 (Partial derivatives): For our example regular expression, $(a \cup \varepsilon) \cdot b^*$, we have

$$PD_{(a\cup\varepsilon)\cdot b^*} = \{(a \cup \varepsilon) \cdot b^*, \varepsilon \cdot b^*\}$$

□

Remark 6.53: Our definition of partial derivatives corresponds very closely to the one given by Antimirov in [Anti94, Anti95]. There is, however, one difference: start-reachability is built into Antimirov's definition. For example, given regular expression $\emptyset \cdot a$, Antimirov's definition would yield the singleton set $\{\emptyset \cdot a\}$ for the partial derivatives, while our definition is $PD_{\emptyset\cdot a} = \{\emptyset \cdot a, \varepsilon\}$. □

The use of partial derivatives is given by the following algorithm detail:

Algorithm detail 6.54 (PD): Partial derivatives (and transition function ∂) are used to encode the automaton in Construction 6.46. □

Encoding the final states of Construction 6.46 is easily done by noticing that $E\bullet \in \mathcal{D}^*(f) \equiv \varepsilon \in \mathcal{L}_{RE}(\overrightarrow{\mathcal{E}}(f)) \equiv Null(\overrightarrow{\mathcal{E}}(f))$.

Armed with this, we are finally in a position to give Antimirov's construction [Anti94, Anti95].

Construction 6.55 (REM-ε, PD): Given $E \in RE$, we construct automaton

$$\begin{array}{ll} \textbf{let} & T(E', a) = \partial_a(E') \\ & F = \{ f \mid f \in PD_E \wedge Null(f) \} \\ \textbf{in} & \\ & (PD_E, V, T, \emptyset, \{E\}, F) \\ \textbf{end} & \end{array}$$

□

Antimirov's papers contain much more information on the correctness of this construction as well as some examples in which the construction produces automata that are much smaller than those produced by Construction (REM-ε). Antimirov derives this construction in a much more language theoretic way.

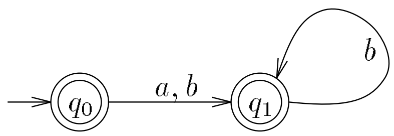


Figure 6.15: *FA* produced by Antimirov's construction.

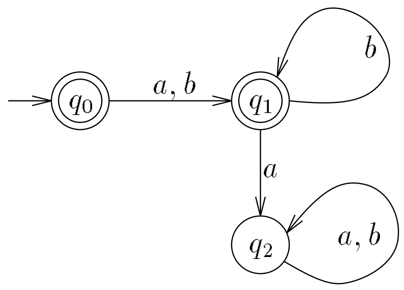


Figure 6.16: *FA* produced by Brzozowski's construction.

Example 6.56 (Antimirov's construction): Using our example expression, $(a \cup \varepsilon) \cdot b^*$, yields the *FA* shown in Figure 6.15. The state set is $PD_{(a\cup\varepsilon)\cdot b^*}$, which we encode as $q_0 = (a \cup \varepsilon) \cdot b^*$ and $q_1 = \varepsilon \cdot b^*$ in the figure. Interestingly, this *FA* is isomorphic to the one in Example 6.28. $\square$

We can compose the subset construction (with start-unreachable state removal) to Antimirov's construction. This yields Construction (REM-ε, PD, SUBSET, USE-S) — a variant of Brzozowski's construction.

Construction 6.57 (REM-ε, PD, SUBSET, USE-S): The composition of $useful_s \circ subset$ onto Antimirov's construction is a variant of Brzozowski's construction. It is not presented explicitly here. $\square$

This corresponds almost exactly to Brzozowski's construction, as presented in [Brzo64]. In the original version, no sink state is constructed. Although we have derived it from Antimirov's relatively new construction, it was in fact one of the first *DFA* constructions to be developed, in the 1960s. In [Anti94], Antimirov also derives Brzozowski's construction in this manner.

Example 6.58 (Brzozowski's construction): Using our example expression, $(a\cup\varepsilon)\cdot b^*$, yields the *FA* shown in Figure 6.16. In the original version, no sink state would have been constructed. This *DFA* is isomorphic to the one in Example 6.29. $\square$

In our version of Brzozowski's construction, we have states which are elements of $\mathcal{P}(PD_E)$ and a transition function $T' \in \mathcal{P}(PD_E) \times V \longrightarrow \mathcal{P}(PD_E)$ defined as

$$T'(p, a) = (\cup\ p' : p' \in p : \partial_a(p'))$$

A possible further encoding of the resulting *DFA* is to represent each state[2] $\{E_0, \ldots, E_k\}$ (where each E_k is a partial derivative of E) by a single regular expression $E_0 \cup \cdots \cup E_k$. Unfortunately, this representation is not unique — we could have chosen a different order to arrange the elements of the set of partial derivatives. In our variant of Brzozowski's construction, the associativity, commutativity and idempotence laws of set union would have allowed us to recognize that $\{E_0, E_1\}$, $\{E_1, E_0\}$ and $\{E_0, E_1, E_0\}$ are the same state. In the new representation, we would have the states (each of which is a regular expression) $E_0 \cup E_1$, $E_1 \cup E_0$, and $E_0 \cup E_1 \cup E_0$, which would are all syntactically different.

In order to recognize these regular expressions as denoting the same state, we define the following equivalence relation on states.

Definition 6.59 (Similarity): Two regular expressions, E and F, are *similar* (written $E \sim F$) if one can be transformed into the other using the rules $E_0 \cup (E_1 \cup E_2) \sim (E_0 \cup E_1) \cup E_2$ (associativity), $E_0 \cup E_1 \sim E_1 \cup E_0$ (commutativity), and $E_0 \cup E_0 \sim E_0$. Note that $\sim$ is an equivalence relation. □

The states are now elements of $[RE]_\sim$. Using similarity, the transition function would be defined as

$$T([F]_\sim, a) = [d_a(F)]_\sim$$

where $d_a(F)$ is the (full, as opposed to partial) derivative of F with respect to a, as defined below.

Definition 6.60 (Full derivatives): We define function $d \in RE \times V \longrightarrow RE$. (For historical reasons, we write $d_a(E)$ instead of $d(E, a)$.) The definition is by structural induction on regular expressions:

$$\begin{array}{lcll}
d_a(\emptyset) & = & \emptyset & \\
d_a(\varepsilon) & = & \emptyset & \\
d_a(b) & = & \mathbf{if}\ a = b\ \mathbf{then}\ \varepsilon\ \mathbf{else}\ \emptyset\ \mathbf{fi} & (b \in V) \\
d_a(E_0 \cup E_1) & = & d_a(E_0) \cup d_a(E_1) & \\
d_a(E_0 \cdot E_1) & = & d_a(E_0) \cdot E_1 \cup \mathbf{if}\ \mathit{Null}(E_0)\ \mathbf{then}\ d_a(E_1)\ \mathbf{else}\ \emptyset\ \mathbf{fi} & \\
d_a(E^*) & = & d_a(E) \cdot E^* & \\
d_a(E^+) & = & d_a(E) \cdot E^* & \\
d_a(E^?) & = & d_a(E) &
\end{array}$$

Note that the right sides in the above definition are regular expressions. This definition corresponds closely to the one given by Brzozowski in [Brzo64]. □

[2] Recall that each of the *DFA* states is a set of states in Antimirov's construction.

The use of derivatives and similarity yields the classically presented Brzozowski construction.

Remark 6.61: In the FIRE Engine, similarity is not used. Instead, a total ordering on regular expressions is defined. The $\cup$ nodes in the full derivatives are then rotated so that, for full derivative $(E_0 \cup E_1) \cup E_2$, we have $E_0 < E_1 < E_2$ (where $<$ is the total ordering). This yields a *similarity normal form*, which encodes the rules of $\sim$. □

6.7 The dual constructions

In some cases, the dual of a construction may be more efficient in practice. The correctness of the dual of a construction can be seen as follows: assume construction $f \in RE \longrightarrow FA$ such that $\mathcal{L}_{FA}(f(E)) = \mathcal{L}_{RE}(E)$; consider the dual of f, $R \circ f \circ R$; we have $\mathcal{L}_{FA}((R \circ f \circ R)(E)) = \mathcal{L}_{FA}(f(E^R))^R = \mathcal{L}_{RE}(E^R)^R = \mathcal{L}_{RE}(E)$.

In this section, we derive some of the more interesting dual constructions. In Figure 6.17, the solid part of the graph indicates the subpart of the taxonomy which is discussed in this section. Note that some of the dual constructions are not considered in this taxonomy at all (the dashed-line subpart of the taxonomy). We omit them since it appears difficult make them more efficient than their duals (which have already been considered).

Given the definition of the dual of function rem_ε, we can present the composition $R \circ rem_\varepsilon \circ R \circ CA$.

Algorithm detail 6.62 (REM-ε-DUAL)**:** The use of composite function $R \circ rem_\varepsilon \circ R$ (Transformation 2.120) is detail REM-ε-DUAL. □

The use of this detail gives the following construction.

Construction 6.63 (REM-ε-DUAL)**:** The composition is $(R \circ rem_\varepsilon \circ R \circ CA)(E) =$

$$\begin{array}{ll}
\textbf{let} & Q' = \{(\mathcal{D}^R)^*(E\bullet)\} \cup \{\, (\mathcal{D}^R)^*(e, BEF) \mid e \in Symnodes_E \,\} \\
& T' = \{\, ((\mathcal{D}^R)^*(e, BEF), E(e), q) \mid (e, AFT) \in q \wedge e \in Symnodes_E \,\} \\
& S' = \{\, s \mid s \in Q' \wedge \bullet E \in s \,\} \\
\textbf{in} & \\
& (Q', V, T', \emptyset, S', \{(\mathcal{D}^R)^*(E\bullet)\}) \\
\textbf{end} &
\end{array}$$

An automaton constructed using this (composite) function has the following properties:

- It has a single final state.
- The single final state has no out-transitions.
- All out-transitions from a given state are on the same symbol (in V).

This follows from the duality between this construction and Construction (REM-ε). □

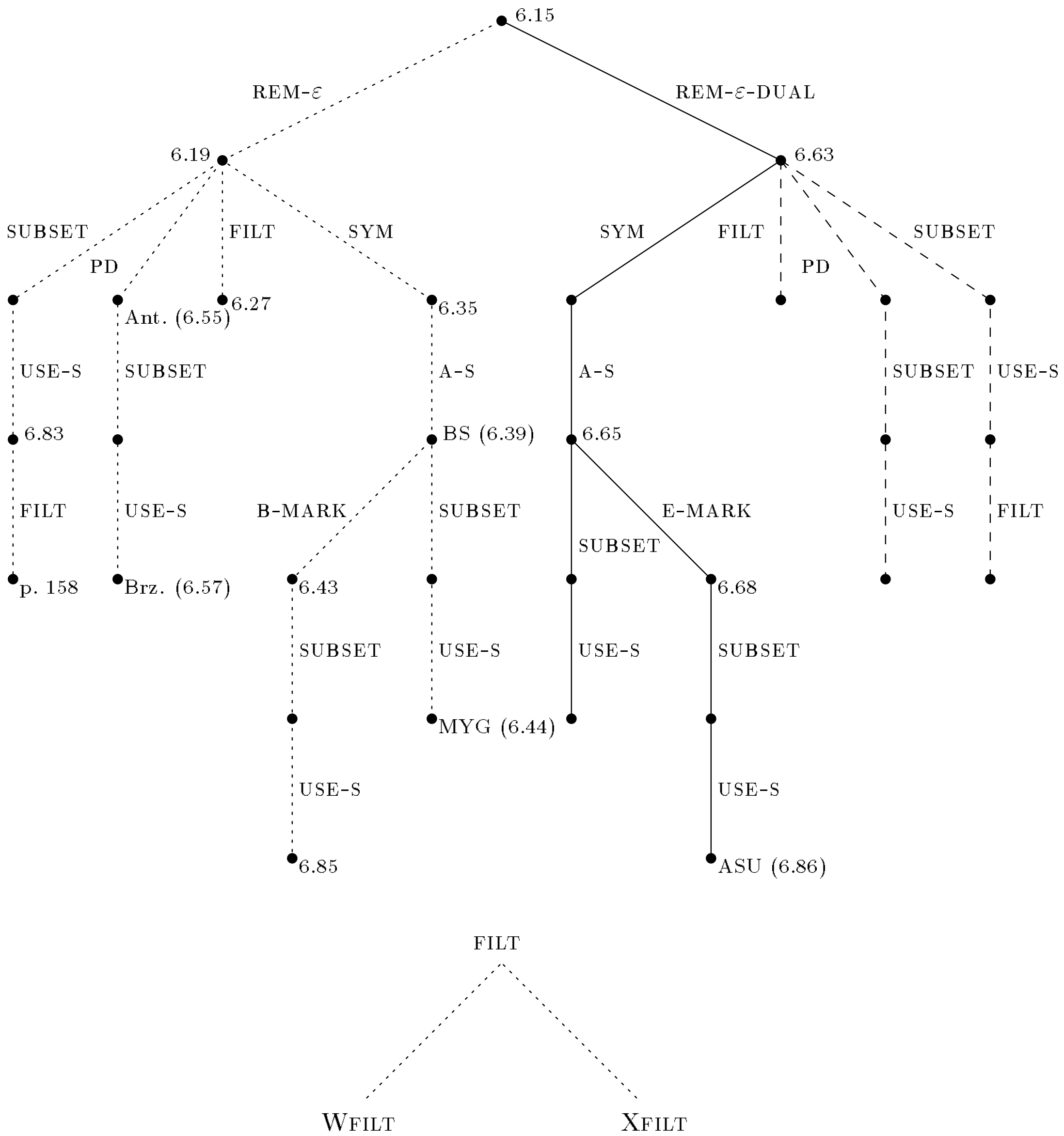


Figure 6.17: The constructions considered in Section 6.7 are shown as solid circles connected by solid lines.

Example 6.64 (Construction (REM-ε-DUAL)): Recalling our example regular expression, $(a \cup \varepsilon) \cdot (b^*)$, we obtain the following item sets for states:

$$q_0'' \quad \begin{array}{l} (\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)), \\ ((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)), \\ (((a\bullet) \cup (\varepsilon)) \cdot ((b)^*)), \\ (((a) \cup (\bullet\varepsilon)) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon\bullet)) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)), \\ (((a) \cup (\varepsilon)) \cdot ((b\bullet)^*)), \\ (((a) \cup (\varepsilon)) \cdot ((b)^*\bullet)), \\ (((a) \cup (\varepsilon)) \cdot ((b)^*)\bullet) \end{array}$$

$$q_1'' \quad \begin{array}{l} (\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)), \\ ((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)), \\ (((a\bullet) \cup (\varepsilon)) \cdot ((b)^*)), \\ (((a) \cup (\bullet\varepsilon)) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon\bullet)) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon)\bullet) \cdot ((b)^*)), \\ (((a) \cup (\varepsilon)) \cdot (\bullet(b)^*)), \\ (((a) \cup (\varepsilon)) \cdot ((\bullet b)^*)), \\ (((a) \cup (\varepsilon)) \cdot ((b\bullet)^*)) \end{array}$$

$$q_2'' \quad \begin{array}{l} (\bullet((a) \cup (\varepsilon)) \cdot ((b)^*)), \\ ((\bullet(a) \cup (\varepsilon)) \cdot ((b)^*)), \\ (((\bullet a) \cup (\varepsilon)) \cdot ((b)^*)) \end{array}$$

As in Example 6.20, the structure of each state is more easily understood if we consider the *FA* given in Example 6.17. The single final state q_0'' is the set of states (in Example 6.17) which are reverse reachable (by ε-transitions) from q_{11} — namely the set $\{q_0, q_1, q_3, q_4, q_5, q_6, q_7, q_9, q_{10}, q_{11}\}$. Similarly, q_1'' is the set of states reverse reachable from q_8 (that is, $q_1'' = \{q_0, q_1, q_3, q_4, q_5, q_6, q_7, q_8, q_9\}$) and q_2'' is the set of states reverse reachable from q_2 ($q_2'' = \{q_0, q_1, q_2\}$).

Note that all three states are start states and one of them is final. The resulting *FA* is shown in Figure 6.18. □

Just as we introduced Algorithm detail (SYM) into Construction (REM-ε) to encode most of the set of states by an element of $Symnodes_E$, we can do the same with the above construction, to give Construction (REM-ε-DUAL, SYM) — which is not given here. (Since the introduction of detail (SYM) is only an encoding, Construction (REM-ε-DUAL, SYM) is the dual of Construction (REM-ε, SYM).) It turns out that the same auxiliary sets ($Follow_E$, $First_E$, and $Last_E$) can also be used to improve this construction. This results

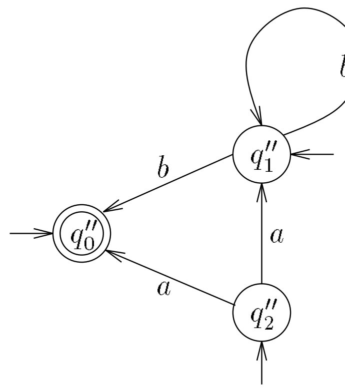


Figure 6.18: *FA* produced by Construction (REM-ε-DUAL)

in the following construction (using Algorithm detail A-S).

Construction 6.65 (REM-ε-DUAL, SYM, A-S): Assuming $E \in RE$, we use the auxiliary sets in the following construction:

$$
\begin{array}{l}
\textbf{let} \quad f \text{ be a new state} \\
\textbf{in} \\
\quad \textbf{let} \quad Q = \{f\} \cup Symnodes_E \\
\quad \qquad T = \{ (e, E(e), f) \mid e \in Last_E \} \\
\quad \qquad \quad \cup \{ (e, E(e), q) \mid q \in Follow_E(e) \wedge e \in Symnodes_E \} \\
\quad \qquad S = First_E \cup \textbf{if } Null(E) \textbf{ then } \{f\} \textbf{ else } \emptyset \textbf{ fi} \\
\quad \textbf{in} \\
\quad \qquad (Q, V, T, \emptyset, S, \{f\}) \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

This construction is the dual of the Berry-Sethi construction. It appears in the literature as [Wat93a, Constr. 4.45]. Note the duality with Construction 6.39. □

Example 6.66 (Construction (REM-ε-DUAL, SYM, A-S)): Using our regular expression, $(a \cup \varepsilon) \cdot b^*$, the auxiliary sets are the same as those used in Example 6.41 and the *FA* is the same as in Figure 6.18 from Example 6.64. In this example, the final state f is represented by q_0, state $2 \diamond 1$ (the b node in the regular expression) is represented by q_1, and state $1 \diamond 1$ (the a node) is represented by q_2. □

In Section 6.5.1, we made use of a begin-marker to make Construction (REM-ε, SYM, A-S) more concise. We can introduce an end-marker (as the dual concept of a begin-marker) to make the above construction more concise.

Algorithm detail 6.67 (E-MARK)**:** By appending a symbol (\$ in our case) to the regular expression E, and using an encoding similar to the one in Section 6.5.1, we improve Construction 6.65. □

Again, it does not matter which symbol is used as our end-marker symbol. The resulting construction is as follows.

Construction 6.68 (REM-ε-DUAL, SYM, A-S, E-MARK)**:** Assuming $E \in RE$, we construct an FA as follows:

$$
\begin{array}{ll}
\textbf{let} & Q = Symnodes_{E\cdot\$} \\
 & T = \{\, (e, (E \cdot \$)(e), f) \mid f \in Follow_{E\cdot\$}(e) \,\} \\
 & S = First_{E\cdot\$} \\
\textbf{in} & \\
 & (Q, V, T, \emptyset, S, \{2\}) \\
\textbf{end} &
\end{array}
$$

The use of an end-marker is also a well-known encoding trick. This particular algorithm appears as [ASU86, Example 3.22, p. 140] as well as [Wat93a, Constr. 4.48]. □

We could compose function $useful_s \circ subset$ with Construction (REM-ε-DUAL, SYM, A-S) to get a DFA construction known as Construction (REM-ε-DUAL, SYM, A-S, SUBSET, USE-S). Alternatively, we could use the end-marker construction, yielding the following construction.

Construction 6.69 (Aho-Sethi-Ullman): Construction (REM-ε-DUAL, SYM, A-S, E-MARK, SUBSET, USE-S) is known as the Aho-Sethi-Ullman construction ([ASU86, Alg. 3.5, Fig. 3.44] and [Wat93a, Constr. 4.50 and Alg. 4.52]). □

This construction is known to be one of the most efficient constructions in practice. For a comparison of the performance of some of the constructions, see Chapter 14. An imperative algorithm implementing it is given in Algorithm 6.86. The following is an example of the Aho-Sethi-Ullman construction.

Example 6.70 (Aho-Sethi-Ullman): Using our running example regular expression $(a \cup \varepsilon) \cdot (b^*)$, we append the end-marker to obtain $((a \cup \varepsilon) \cdot (b^*)) \cdot \$$. We compute the following auxiliary sets:

- $Symnodes_{((a\cup\varepsilon)\cdot(b^*))\cdot\$} = \{1 \diamond 1 \diamond 1, 1 \diamond 2 \diamond 1, 2\}$.
- $First_{((a\cup\varepsilon)\cdot(b^*))\cdot\$} = \{1 \diamond 1 \diamond 1, 1 \diamond 2 \diamond 1, 2\}$.
- $Follow_{((a\cup\varepsilon)\cdot(b^*))\cdot\$} = \{(1 \diamond 1 \diamond 1, 1 \diamond 2 \diamond 1), (1 \diamond 2 \diamond 1, 1 \diamond 2 \diamond 1), (1 \diamond 1 \diamond 1, 2), (1 \diamond 2 \diamond 1, 2)\}$.

Unlike our presentation, Aho, Sethi, and Ullman's presentation also removes the sink state from the DFA. For this example, we also remove the sink state. We have the following two states $q_0 = \{1 \diamond 1 \diamond 1, 1 \diamond 2 \diamond 1, 2\}$ and $q_1 = \{1 \diamond 2 \diamond 1, 2\}$. The resulting DFA is shown in Figure 6.19. □

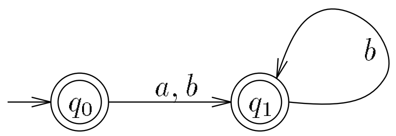


Figure 6.19: *DFA* produced by the Aho-Sethi-Ullman construction.

6.8 Precomputing the auxiliary sets and *Null*

In this section, we consider how to precompute the sets *Symnodes*, *First*, *Last*, *Follow*, and predicate *Null*. They can all be computed using a single recursive traversal of the tree $E \in RE$. This is especially efficient when the set $Symnodes_E$ is encoded as a set of integers, as is done in FIRE Lite and the FIRE Engine. In the following properties, we give inductive definitions which are based upon the structure of regular expressions.

Property 6.71 (Inductive definition of *Null*): The following definitions follow from the definition of *Null*:

$$\begin{array}{lcl}
Null(\emptyset) & = & false \\
Null(\varepsilon) & = & true \\
Null(a) & = & false \\
Null(E_0 \cup E_1) & = & Null(E_0) \vee Null(E_1) \\
Null(E_0 \cdot E_1) & = & Null(E_0) \wedge Null(E_1) \\
Null(E_0^*) & = & true \\
Null(E_0^+) & = & Null(E_0) \\
Null(E_0^?) & = & true
\end{array}$$

□

Example 6.72 ($Null(E)$): Recalling our example $(a \cup \varepsilon) \cdot (b^*) \in RE$:

$$\begin{array}{ll}
& Null((a \cup \varepsilon) \cdot (b^*)) \\
= & \quad \{ \textit{Null} \text{ on a} \cdot \text{node} \} \\
& Null(a \cup \varepsilon) \wedge Null(b^*) \\
= & \quad \{ \textit{Null} \text{ on } \cup \text{ and } * \text{ nodes} \} \\
& (Null(a) \vee Null(\varepsilon)) \wedge true \\
= & \quad \{ \textit{Null} \text{ on } a \text{ and } \varepsilon \text{ nodes} \} \\
& (false \vee true) \wedge true \\
= & \quad \{ \text{definitions of } \vee \text{ and } \wedge \} \\
& true
\end{array}$$

□

Property 6.73 (Inductive definition of *Symnodes*): The following definitions follow from the definition of *Symnodes*:

$$\begin{array}{lcll}
Symnodes_{\emptyset} & = & \emptyset & \\
Symnodes_{\varepsilon} & = & \emptyset & \\
Symnodes_{a} & = & \{0\} & \text{(the root)} \\
Symnodes_{E_0 \cup E_1} & = & 1 \diamond Symnodes_{E_0} \cup 2 \diamond Symnodes_{E_1} & \\
Symnodes_{E_0 \cdot E_1} & = & 1 \diamond Symnodes_{E_0} \cup 2 \diamond Symnodes_{E_1} & \\
Symnodes_{E_0^*} & = & 1 \diamond Symnodes_{E_0} & \\
Symnodes_{E_0^+} & = & 1 \diamond Symnodes_{E_0} & \\
Symnodes_{E_0^?} & = & 1 \diamond Symnodes_{E_0} &
\end{array}$$

□

Example 6.74 ($Symnodes_E$): Recalling our example $(a \cup \varepsilon) \cdot (b^*) \in RE$:

$$\begin{array}{ll}
 & Symnodes_{(a \cup \varepsilon)\cdot(b^*)} \\
= & \quad \{ \text{ } Symnodes \text{ on a} \cdot \text{node } \} \\
 & 1 \diamond Symnodes_{a \cup \varepsilon} \cup 2 \diamond Symnodes_{b^*} \\
= & \quad \{ \text{ } Symnodes \text{ on } \cup \text{ and } * \text{ nodes } \} \\
 & 1 \diamond (1 \diamond Symnodes_a \cup 2 \diamond Symnodes_{\varepsilon}) \cup 2 \diamond (1 \diamond Symnodes_b) \\
= & \quad \{ \text{ } Symnodes \text{ on } a, b, \text{ and } \varepsilon \text{ nodes } \} \\
 & 1 \diamond (1 \diamond \{0\} \cup 2 \diamond \emptyset) \cup 2 \diamond (1 \diamond \{0\}) \\
= & \quad \{ \diamond \text{ is associative } \} \\
 & \{1 \diamond 1 \diamond 0, 2 \diamond 1 \diamond 0\} \\
= & \quad \{ \text{ 0 is the unit of } \diamond \text{ } \} \\
 & \{1 \diamond 1, 2 \diamond 1\}
\end{array}$$

□

Property 6.75 (Inductive definition of *First*): We present an inductive definition for $First_E$ based upon the structure of E:

$$\begin{array}{lcll}
First_{\emptyset} & = & \emptyset & \\
First_{\varepsilon} & = & \emptyset & \\
First_{a} & = & \{0\} & \text{(the root)} \\
First_{E_0 \cup E_1} & = & 1 \diamond First_{E_0} \cup 2 \diamond First_{E_1} & \\
First_{E_0 \cdot E_1} & = & 1 \diamond First_{E_0} \cup \textbf{if } Null(E_0) \textbf{ then } 2 \diamond First_{E_1} \textbf{ else } \emptyset \textbf{ fi} & \\
First_{E_0^*} & = & 1 \diamond First_{E_0} & \\
First_{E_0^+} & = & 1 \diamond First_{E_0} & \\
First_{E_0^?} & = & 1 \diamond First_{E_0} &
\end{array}$$

□

Example 6.76 (*First*)**:** Again, using $(a \cup \varepsilon) \cdot (b^*) \in RE$:

$$\begin{array}{ll} & First_{(a\cup\varepsilon)\cdot(b^*)} \\ = & \quad \{\, First \text{ on a} \cdot \text{node} \,\} \\ & 1 \diamond First_{a\cup\varepsilon} \cup \textbf{if } Null(a \cup \varepsilon) \textbf{ then } 2 \diamond First_{b^*} \textbf{ else } \emptyset \textbf{ fi} \\ = & \quad \{\, \text{definition of } Null \,\} \\ & 1 \diamond First_{a\cup\varepsilon} \cup 2 \diamond First_{b^*} \\ = & \quad \{\, First \text{ on } \cup \text{ and } * \text{ nodes} \,\} \\ & 1 \diamond (1 \diamond First_a \cup 2 \diamond First_\varepsilon) \cup 2 \diamond (1 \diamond First_b) \\ = & \quad \{\, First \text{ on } a,\, b \text{ and } \varepsilon \text{ nodes} \,\} \\ & \{1 \diamond 1, 2 \diamond 1\} \end{array}$$

□

Property 6.77 (Inductive definition of ***Last*****):** The following are derived in a similar manner to *First*:

$$\begin{array}{lll} Last_\emptyset & = & \emptyset \\ Last_\varepsilon & = & \emptyset \\ Last_a & = & \{0\} \qquad\qquad \text{(the root)} \\ Last_{E_0\cup E_1} & = & 1 \diamond Last_{E_0} \cup 2 \diamond Last_{E_1} \\ Last_{E_0\cdot E_1} & = & 2 \diamond Last_{E_1} \cup \textbf{if } Null(E_1) \textbf{ then } 1 \diamond Last_{E_0} \textbf{ else } \emptyset \textbf{ fi} \\ Last_{E_0^*} & = & 1 \diamond Last_{E_0} \\ Last_{E_0^+} & = & 1 \diamond Last_{E_0} \\ Last_{E_0^?} & = & 1 \diamond Last_{E_0} \end{array}$$

□

Example 6.78 (*Last*)**:** We use the regular expression $(a \cup \varepsilon) \cdot (b^*)$:

$$\begin{array}{ll} & Last_{(a\cup\varepsilon)\cdot(b^*)} \\ = & \quad \{\, Last \text{ on a} \cdot \text{node} \,\} \\ & 2 \diamond Last_{b^*} \cup \textbf{if } Null(b^*) \textbf{ then } 1 \diamond Last_{a\cup\varepsilon} \textbf{ else } \emptyset \textbf{ fi} \\ = & \quad \{\, \text{definition of } Null \,\} \\ & 2 \diamond Last_{b^*} \cup 1 \diamond Last_{a\cup\varepsilon} \\ = & \quad \{\, Last \text{ on } * \text{ and } \cup \text{ nodes} \,\} \\ & 2 \diamond (1 \diamond Last_b) \cup 1 \diamond (1 \diamond Last_a \cup 2 \diamond Last_\varepsilon) \\ = & \quad \{\, Last \text{ on } a,\, b \text{ and } \varepsilon \text{ nodes} \,\} \\ & \{1 \diamond 1, 2 \diamond 1\} \end{array}$$

□

Before giving an inductive definition of *Follow*, we extend operator $\diamond$.

Notation 6.79 (Extension of $\diamond$): Given $i \in \mathbb{N}$ and $A \subseteq dom(E) \times dom(E)$, we extend $\diamond$ as follows:

$$i \diamond A = \{ (i \diamond a, i \diamond b) \mid (a, b) \in A \}$$

This is purely a notational convenience. □

Property 6.80 (Inductive definition of *Follow*): We present an inductive definition of *Follow*:

$$\begin{array}{lll}
Follow_{\emptyset} & = & \emptyset \\
Follow_{\varepsilon} & = & \emptyset \\
Follow_{a} & = & \{0\} \\
Follow_{E_0 \cup E_1} & = & 1 \diamond Follow_{E_0} \cup 2 \diamond Follow_{E_1} \\
Follow_{E_0 \cdot E_1} & = & 1 \diamond Follow_{E_0} \cup 2 \diamond Follow_{E_1} \cup (1 \diamond Last_{E_0}) \times (2 \diamond First_{E_1}) \\
Follow_{E_0^*} & = & 1 \diamond Follow_{E_0} \cup (1 \diamond Last_{E_0}) \times (1 \diamond First_{E_0}) \\
Follow_{E_0^+} & = & 1 \diamond Follow_{E_0} \cup (1 \diamond Last_{E_0}) \times (1 \diamond First_{E_0}) \\
Follow_{E_0^?} & = & 1 \diamond Follow_{E_0}
\end{array}$$

□

Example 6.81 (*Follow*): Taking our usual regular expression $(a \cup \varepsilon) \cdot (b^*)$:

$$\begin{array}{ll}
 & Follow_{(a \cup \varepsilon) \cdot (b^*)} \\
= & \quad \{ \textit{Follow} \text{ on a } \cdot \text{ node} \} \\
 & 1 \diamond Follow_{a \cup \varepsilon} \cup 2 \diamond Follow_{b^*} \cup (1 \diamond Last_{a \cup \varepsilon}) \times (2 \diamond First_{b^*}) \\
= & \quad \{ \text{definitions of } Last_{a \cup \varepsilon} \text{ and } First_{b^*} \} \\
 & 1 \diamond Follow_{a \cup \varepsilon} \cup 2 \diamond Follow_{b^*} \cup \{(1 \diamond 1, 2 \diamond 1)\} \\
= & \quad \{ \text{definitions of } Follow_{a \cup \varepsilon} \text{ and } Follow_{b^*} \} \\
 & 2 \diamond (1 \diamond Follow_b \cup (1 \diamond Last_b \times 1 \diamond First_b)) \cup \{(1 \diamond 1, 2 \diamond 1)\} \\
= & \quad \{ \text{definition of } First_b, Last_b, Follow_b \} \\
 & 2 \diamond (1 \diamond \emptyset \cup (1 \diamond \{0\} \times 1 \diamond \{0\})) \cup \{(1 \diamond 1, 2 \diamond 1)\} \\
= & \quad \{ \text{calculus} \} \\
 & 2 \diamond \{(1, 1)\} \cup \{(1 \diamond 1, 2 \diamond 1)\} \\
= & \quad \{ \text{calculus} \} \\
 & \{(2 \diamond 1, 2 \diamond 1)\} \cup \{(1 \diamond 1, 2 \diamond 1)\}
\end{array}$$

Intuitively, this shows that (in the language denoted by regular expression $(a \cup \varepsilon) \cdot (b^*)$) an a can be followed by a b and a b can be followed by a b. □

Remark 6.82: When the integer encoding of $Symnodes_E$ is used, the inductive definitions above turn out to be the same ones that are given in, for example, [ASU86]. There are a number of techniques to speed up the computation of these sets in practice. Of particular importance are the methods described by Brüggemann-Klein in [B-K93a] and by Chang and Paige in [Chan92, CP92]; a short summary of these methods appears in [Wat93a, Section 4.5.2]. Brüggemann-Klein's method works by manipulating the regular expression into a form which is more suited to the computation of the auxiliary sets. The Chang-Paige method works by computing the subparts of the auxiliary sets on an 'as-needed' (lazy) basis. □

6.9 Constructions as imperative programs

In this section, we give some algorithms implementing constructions from the past few sections. These algorithms are in the form most often seen in text-book and journal presentations of constructions. We make use of the algorithms presented in Section 2.6.2.1. We will not concern ourselves with the computation of any of the auxiliary sets or relations.

6.9.1 The item set constructions

We can present an imperative program which computes composite function $useful_s \circ subset \circ rem_\varepsilon \circ CA$ — Construction (REM-ε, SUBSET, USE-S).

Algorithm 6.83 (Implementing (REM-ε, SUBSET, USE-S)):

$\{\ E \in RE\ \}$
$S, T := \{\mathcal{D}^*(\bullet E)\}, \emptyset;$
$D, U := \emptyset, S;$
do $U \neq \emptyset \rightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \rightarrow$
    $d := (\cup\ e : (e, BEF) \in u \wedge E(e) = a : \mathcal{D}^*(e, AFT));$
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]\ \ d \in D \rightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{\ f \mid f \in D \wedge E\bullet \in f\ \}$
$\{\ \mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)$
  $\wedge\ (D, V, T, \emptyset, S, F) \in DFA$
  $\wedge\ Complete(D, V, T, \emptyset, S, F)\ \}$

□

This algorithm is essentially the 'deterministic item set construction', given in [Wat93a, Constr. 5.69].

We can also present the 'filtered' version, using filter $\mathcal{W}$. In order to do this, we simply rewrite some of the statements in the above algorithm:

- Assignment $S := \{\mathcal{D}^*(\bullet E)\}$ becomes $S := \{\mathcal{W}(\mathcal{D}^*(\bullet E))\}$.
- Statement

  $$d := (\cup\ e : (e, BEF) \in u \wedge E(e) = a : \mathcal{D}^*(e, AFT))$$

  is replaced by

  $$d := \mathcal{W}(\cup\ e : (e, BEF) \in u \wedge E(e) = a : \mathcal{D}^*(e, AFT))$$

The resulting algorithm implements Construction (REM-ε, SUBSET, USE-S, WFILT). This algorithm does not appear in the literature. In Chapter 14, we see that it displays good performance in practice. It produces *DFA*s which are isomorphic to those produced by Construction (REM-ε-DUAL, SYM, A-S, SUBSET, USE-S) and the Aho-Sethi-Ullman construction.

6.9.2 The *Symnodes* constructions

We present an algorithm implementing Construction (REM-ε, SYM, A-S, SUBSET, USE-S) which produces a *DFA*. Here, the first iteration is unrolled to accommodate the special treatment of the start state in construction (REM-ε, SYM, A-S), and some obvious improvements have not yet been made.

$\{\ E \in RE\ \}$
let $S = \{\{s\}\} : s$ is a new state;
$T := \emptyset$;
$D, U := \emptyset, S$;
let $u : u \in U$;
$\{\ u = \{s\}\ \}$
$D, U := D \cup \{u\}, U \setminus \{u\}$;
for $a : a \in V \rightarrow$
  $d := (\cup\ p : p \in u : \{\, e \mid e \in First_E \wedge E(e) = a\,\})$;
  $\{\ p \in u \equiv p = \{s\}\ \}$
  if $d \notin D \rightarrow U := U \cup \{d\}$
  $[\!]\ \ d \in D \rightarrow$ **skip**
  fi;
  $T := T \cup \{(u, a, d)\}$
rof;

do $U \neq \emptyset \rightarrow$
  let $u : u \in U$;
  $D, U := D \cup \{u\}, U \setminus \{u\}$;
  for $a : a \in V \rightarrow$
    $d := (\cup\ p : p \in u : \{\, e \mid e \in \mathit{Follow}_E(p) \land E(e) = a \,\})$;
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]$ $d \in D \rightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{\, d \mid d \in D \land d \cap \mathit{Last}_E \neq \emptyset \,\} \cup$ **if** $\mathit{Null}(E)$ **then** S **else** $\emptyset$ **fi**
$\{\ \mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)$
  $\land\ (D, V, T, \emptyset, S, F) \in \mathit{DFA}$
  $\land\ \mathit{Complete}(D, V, T, \emptyset, S, F)\ \}$

Some simplification gives the following algorithm — the McNaughton-Yamada-Glushkov construction [MY60, Glus61]; it is also given in [Wat93a, Algorithm 4.42].

Algorithm 6.84 (Implementing McNaughton-Yamada-Glushkov):

$\{\ E \in \mathit{RE}\ \}$
let $S = \{\{s\}\} : s$ is a new state;
$T := \emptyset$;
$D, U := S, \emptyset$;
for $a : a \in V \rightarrow$
  $d := \{\, e \mid e \in \mathit{First}_E \land E(e) = a \,\}$;
  $U := U \cup \{d\}$;
  $T := T \cup \{(\{s\}, a, d)\}$
rof;
do $U \neq \emptyset \rightarrow$
  let $u : u \in U$;
  $D, U := D \cup \{u\}, U \setminus \{u\}$;
  for $a : a \in V \rightarrow$
    $d := (\cup\ p : p \in u : \{\, e \mid e \in \mathit{Follow}_E(p) \land E(e) = a \,\})$;
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]$ $d \in D \rightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{\, d \mid d \in D \land d \cap \mathit{Last}_E \neq \emptyset \,\} \cup$ **if** $\mathit{Null}(E)$ **then** S **else** $\emptyset$ **fi**
$\{\ \mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)$

$\wedge\ (D, V, T, \emptyset, S, F) \in DFA$
$\wedge\ Complete(D, V, T, \emptyset, S, F)\ \}$

□

To show the simplicity gained by the use of a begin-marker, we give an imperative algorithm implementing Construction (REM-ε, SYM, A-S, B-MARK, SUBSET, USE-S):

Algorithm 6.85 (Implementing (REM-ε, SYM, A-S, B-MARK, SUBSET, USE-S)):

$\{\ E \in RE\ \}$
$S, T := \{\{1\}\}, \emptyset;$
$D, U := \emptyset, S;$
do $U \neq \emptyset \rightarrow$
  let $u : u \in U;$
  $D, U := D \cup \{u\}, U \setminus \{u\};$
  for $a : a \in V \rightarrow$
    $d := (\cup\ p : p \in u : \{\ e \mid e \in Follow_E(p) \wedge E(e) = a\ \});$
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]\ \ d \in D \rightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{\ f \mid f \in D \wedge f \cap Last_{\$.E} \neq \emptyset\ \}$
$\{\ \mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)$
$\wedge\ (D, V, T, \emptyset, S, F) \in DFA$
$\wedge\ Complete(D, V, T, \emptyset, S, F)\ \}$

□

This algorithm is somewhat more elegant and more practical than the one given as Algorithm 6.84, thanks to the fact that the start state does not require special treatment.

6.9.3 A dual construction

The following algorithm implements the Aho-Sethi-Ullman construction — Construction (REM-ε-DUAL, SYM, A-S, E-MARK, SUBSET, USE-S).

Algorithm 6.86 (Implementing Aho-Sethi-Ullman):

$\{\ E \in RE\ \}$
$S, T := \{First_{E \cdot \$}\}, \emptyset;$
$D, U := \emptyset, S;$

do $U \neq \emptyset \rightarrow$
  let $u : u \in U$;
  $D, U := D \cup \{u\}, U \setminus \{u\}$;
  for $a : a \in V \rightarrow$
    $d := (\cup\ p : p \in u \wedge E(p) = a : \mathit{Follow}_{E \cdot \$}(p))$;
    if $d \notin D \rightarrow U := U \cup \{d\}$
    $[\!]$ $d \in D \rightarrow$ **skip**
    fi;
    $T := T \cup \{(u, a, d)\}$
  rof
od;
$F := \{ f \mid f \in D \wedge 2 \in f \}$
$\{ \mathcal{L}_{FA}(D, V, T, \emptyset, S, F) = \mathcal{L}_{RE}(E)$
  $\wedge\ (D, V, T, \emptyset, S, F) \in \mathit{DFA}$
  $\wedge\ \mathit{Complete}(D, V, T, \emptyset, S, F)\ \}$

$\square$

This algorithm appears as [ASU86, Alg. 3.5, Fig. 3.44] and as [Wat93a, Constr. 4.50 and Alg. 4.52].

Comparing this algorithm with the variant of the McNaughton-Yamada-Glushkov algorithm (Algorithm 6.85) shows that the two are very similar, with two notable exceptions: the assignment to d in the inner repetition, and assignment to F are both considerably more efficient to implement in the above construction than in the McNaughton-Yamada-Glushkov algorithm. The differences in efficiency can be seen clearly from the data presented in Chapter 14.

6.10 Conclusions

A number of conclusions can be drawn about the taxonomy presented in this chapter:

- The original algorithms were presented in widely differing styles (some where aimed at compiler applications, while others were aimed at digital circuitry applications) over a period of many years. Interestingly, they can all be related in rather direct ways. This can be attributed to the following facts:
  - We used a ‘canonical’ construction (directly related to the input regular expression), in which each state contains the ‘maximal’ amount of information.
  - We introduced various efficient encodings of the canonical states. Some the encodings removed information, allowing states to be safely identified (and therefore creating smaller automata).

  - Some simple algorithmic building blocks (for example ε-transition removal, the subset construction, and start-unreachable state removal) were identified and factored out. This lead to concise descriptions of the algorithms (as compositions of functions), and subsequent transformation into imperative programs was particularly easy. The direct transformation of imperative programs would have been difficult.

- The earlier taxonomy presented in [Wat93a] contained two taxonomy trees. The development of that taxonomy seemed to indicate that the two subfamilies of algorithms were related, but could not be derived from one another. The taxonomy presented in this chapter shows otherwise: all of the constructions can be derived from a single canonical construction.

- Many of the constructions produce automata whose states contain information. For example, the states of the canonical construction are 'items', encoding the left and right languages of the states. This approach had two advantages:

  - The additional information made it easier to argue the correctness of each of the constructions, especially the canonical construction.
  - The information could be encoded in various ways, leading to more efficient constructions.
  - In some cases, the encodings may lead to states being identified — reducing the size of the produced automata.

- One of the most recently developed constructions (Antimirov's) was also successfully integrated into the taxonomy.

- All of the constructions were successfully presented as compositions of mathematical functions (as opposed to only being presented as imperative programs). The corresponding imperative programs were also presented.

- We can also draw some conclusions about the individual constructions and relationships between them:

  - The Berry-Sethi construction is an encoding of an ε-transition removal function composed with the canonical construction.
  - Antimirov's construction is an encoding of the Berry-Sethi construction. (Note that Antimirov's construction may produce smaller automata than the Berry-Sethi construction, meaning that Antimirov's construction is an optimization.)
  - The deterministic item set construction can be improved through the use of 'filters'. One such filter yields DeRemer's construction, while another filter yields a new construction.
  - The McNaughton-Yamada-Glushkov is the subset construction composed with the Berry-Sethi construction.

  - The subset construction composed with the Antimirov's construction is a variant of Brzozowski's construction.
  - The original version of Brzozowski's construction, with the similarity relation on regular expressions, is shown to be a further encoding of our variant.
  - From the above three observations, we can conclude that the deterministic item set construction, the McNaughton-Yamada-Glushkov construction, and Brzozowski's construction are encodings of one another. (It is possible, however, that Brzozowski's construction produces a smaller *DFA*.)
  - The Aho-Sethi-Ullman construction is the subset construction composed with the dual of the Berry-Sethi construction.
  - The new (filter-based) construction is an encoding of the Aho-Sethi-Ullman construction.

- The taxonomy presented here was crucial to the development of FIRE Lite, a C++ toolkit of construction algorithms. The structure of the taxonomy tree is reflected in the inheritance hierarchy of FIRE Lite.
- Lastly, we note that there is room for improvement in the taxonomy presented in this chapter. The constructions which are based upon derivatives were added after some of the other constructions were taxonomized. As a result, it may be possible to factor their development even more.

Chapter 7

DFA minimization algorithms

This chapter presents a taxonomy of finite automata minimization algorithms. Brzozowski's elegant minimization algorithm differs from all other known minimization algorithms, and is derived separately. All of the remaining algorithms depend upon computing an equivalence relation on states. We define the equivalence relation, the partition that it induces, and its complement. Additionally, some useful properties are derived. It is shown that the equivalence relation is the greatest fixed point of a function, providing a useful characterization of the required computation. We derive an upperbound on the number of approximation steps required to compute the fixed point. Algorithms computing the equivalence relation (or the partition, or its complement) are derived systematically in the same framework. The algorithms include Hopcroft's, several algorithms from text-books (including Hopcroft and Ullman's [HU79], Wood's [Wood87], and Aho, Sethi, and Ullman's [ASU86]), and several new algorithms or variants of existing algorithms.

An early version of this taxonomy appeared in [Wat93b].

7.1 Introduction

The minimization of deterministic finite automata is a problem that has been studied since the late 1950's. Simply stated, the problem is to find the unique (up to isomorphism) minimal deterministic finite automaton that accepts the same language as a given deterministic finite automaton. Algorithms solving this problem are used in applications ranging from compiler construction to hardware circuit minimization. With such a variety of applications, the number of differing presentations also grew: most text-books present their own variation, while the algorithm with the best running time (Hopcroft's) remains obscure and difficult to understand.

This chapter presents a taxonomy of finite automata minimization algorithms. The need for a taxonomy is illustrated by the following:

- Most text-book authors claim that their minimization algorithm is directly derived from those presented by Huffman [Huff54] and Moore [Moor56]. Unfortunately, most text-books present vastly differing algorithms (for example, compare [AU92], [ASU86], [HU79], and [Wood87]), and only the algorithms presented by Aho and Ullman and by Wood are directly derived from those originally presented by Huffman and Moore.

- While most of the algorithms rely on computing an equivalence relation on states, many of the explanations accompanying the algorithm presentations do not explicitly mention whether the algorithm computed the equivalence relation, the partition (of states) that it induces, or its complement.

- Comparison of the algorithms is further hindered by the vastly differing styles of presentation — sometimes as imperative programs, or as functional programs, but frequently only as a descriptive paragraph.

For notational convenience, we restrict ourselves to producing minimal *Complete DFA*s. This is strictly a notational convenience, as the minimization algorithms can be modified to work for in-*Complete DFA*s. A *Complete* minimized *DFA* will (in general) have one more state (a sink state) than a in-*Complete* minimized *DFA*.

All except one of the algorithms rely on determining the set of automaton states which are equivalent[1]. The algorithm that does not make use of equivalent states is discussed in Section 7.2. In Section 7.3 the definition and some properties of equivalence of states is given. Algorithms that compute equivalent states are presented in Section 7.4. The main results of the taxonomy are summarized in the conclusions — Section 7.5. The minimization algorithm relationships are shown in a 'family tree' in Figure 7.1. Unlike in Chapters 4 and 6, the algorithm and problem details remain implicit in the presentation of the algorithms. In the family tree, the details are shown as edges, depicting refinements of the solution.

The principal computation in most minimization algorithms is the determination of equivalent (or inequivalent) states — thus yielding an equivalence relation on states. In this chapter, we consider the following minimization algorithms:

- Brzozowski's finite automaton[2] minimization algorithm as presented in [Brzo62]. This elegant algorithm (Section 7.2) was originally invented by Brzozowski, and has since been re-invented by a number of others (in some cases without credit to Brzozowski). Given a (possibly nondeterministic) finite automaton without ε-transitions, this algorithm produces the minimal deterministic finite automaton accepting the same language.

- Layerwise computation of equivalence as presented in [Wood87, Brau88, Urba89]. This algorithm (Algorithm 7.18, also known as Wood's algorithm in the literature) is a straightforward implementation suggested by the approximation sequence arising from the fixed-point definition of equivalence of states.

[1] Equivalence of states is defined later.

[2] This algorithm also works on nondeterministic finite automata, in contrast with the other algorithms which only work on deterministic finite automata.

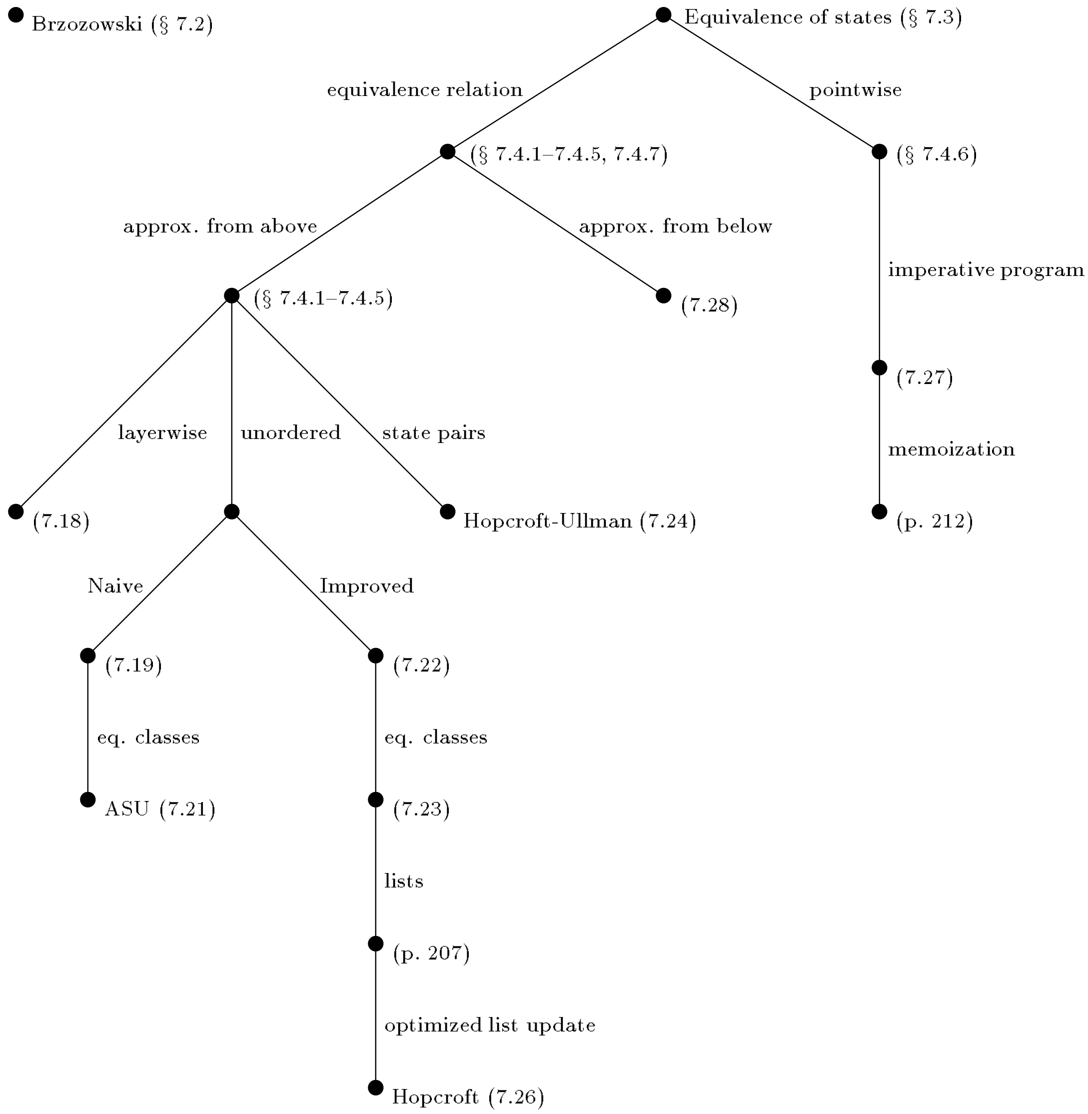


Figure 7.1: The family trees of finite automata minimization algorithms. Brzozowski's minimization algorithm is unrelated to the others, and appears as a separate (single vertex) tree. Each algorithm presented in this chapter appears as a vertex in this tree. For each algorithm that appears explicitly in this chapter, the construction number appears in parentheses (indicating where it appears in this chapter). For algorithms that do not appear explicitly, a reference to the section or page number is given. Edges denote a refinement of the solution (and therefore explicit relationships between algorithms). They are labeled with the name of the refinement.

- Unordered computation of equivalence. This algorithm (Algorithm 7.19, not appearing in the literature) computes the equivalence relation; pairs of states (for consideration of equivalence) are chosen in an arbitrary order.

- Unordered computation of equivalence classes as presented in [ASU86]. This algorithm (Algorithm 7.21) is a modification of the above algorithm computing equivalence of states.

- Improved unordered computation of equivalence. This algorithm (Algorithm 7.22, not appearing in the literature) also computes the equivalence relation in an arbitrary order. The algorithm is a minor improvement of the other unordered algorithm.

- Improved unordered computation of equivalence classes. This algorithm (appearing as Algorithm 7.23 in this dissertation, not appearing in the literature) is a modification of the above algorithm to compute the equivalence classes of states. This algorithm is used in the derivation of Hopcroft's minimization algorithm.

- Hopcroft and Ullman's algorithm as presented in [HU79]. This algorithm (Algorithm 7.24) computes the inequivalence (distinguishability) relation. Although it is based upon the algorithms of Huffman and Moore [Huff54, Moor56], this algorithm uses some interesting encoding techniques.

- Hopcroft's algorithm as presented in [Hopc71, Grie73]. This algorithm (appearing here as Algorithm 7.26) is the best known algorithm (in terms of running time complexity) for minimization. As the original presentation by Hopcroft is difficult to understand, the presentation in this chapter is based upon the one given by Gries.

- Pointwise computation of equivalence. This algorithm (Algorithm 7.27, appearing in the literature only in a form suitable for comparing types for structural equivalence) computes the equivalence of a given pair of states. It draws upon some non-automata related techniques, such as: structural equivalence of types and memoization of functional programs.

- Computation of equivalence from below (with respect to refinement). This algorithm (Algorithm 7.28, not appearing in the literature) computes the equivalence relation from below. Unlike any of the other known algorithms, the intermediate result of this algorithm can be used to construct a smaller (although not minimal) deterministic finite automaton.

7.2 An algorithm due to Brzozowski

Most minimization algorithms are applied to a *DFA*. In the case of a nondeterministic *FA*, the subset construction (function *subset* — Transformation 2.121) is applied first, followed by the minimization algorithm. In this section, we consider the possibility of applying the subset construction (with start-unreachable state removal) after an (as yet unknown) algorithm, thus yielding a minimal *DFA*. We now derive such an algorithm.

Let $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ be the ε-*free* finite automaton (not necessarily a *DFA*) to be minimized and $M_2 = (Q_2, V, T_2, \emptyset, S_2, F_2)$ be the minimized *Complete DFA* such that $\mathcal{L}_{FA}(M_0) = \mathcal{L}_{FA}(M_2)$ (and of course $Min_C(M_2)$ — see Definition 2.109). (For the remainder of this section we make use of $Minimal_C$ (Property 2.111) as opposed to Min_C.) Since we apply the subset construction last, we have some intermediate finite automaton $M_1 = (Q_1, V, T_1, \emptyset, S_1, F_1)$ such that $M_2 = (useful_s \circ subset)(M_1)$. (Note that $Useful_s(M_2) \wedge Complete(M_2)$ holds.) We require that M_1 is somehow obtained from M_0, and that $\mathcal{L}_{FA}(M_2) = \mathcal{L}_{FA}(M_1) = \mathcal{L}_{FA}(M_0)$.

From the definition of $Minimal_C(M_2)$ (which appears in Property 2.111), we require

$$(\forall\ p, q : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge Useful_s(M_2) \wedge Complete(M_2)$$

For all states $q \in Q_2$ we have $q \in \mathcal{P}(Q_1)$ since $M_2 = (useful_s \circ subset)(M_1)$. Property 2.123 of the subset construction gives

$$(\forall\ p : p \in Q_2 : \overrightarrow{\mathcal{L}}(p) = (\cup\ q : q \in Q_1 \wedge q \in p : \overrightarrow{\mathcal{L}}(q)))$$

We need a sufficient condition on M_1 to ensure $Minimal_C(M_2)$. The following derivation gives such a condition:

$$\begin{array}{ll}
& Minimal_C(M_2) \\
\equiv & \quad \{ \text{definition of } Minimal_C \text{ (Property 2.111)} \} \\
& (\forall\ p, q : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge Useful_s(M_2) \wedge Complete(M_2) \\
\Leftarrow & \quad \{ \text{Property 2.123; } M_2 = (useful_s \circ subset)(M_1) \} \\
& (\forall\ p, q : p \in Q_1 \wedge q \in Q_1 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \cap \overrightarrow{\mathcal{L}}(q) = \emptyset) \wedge Useful_f(M_1) \\
\equiv & \quad \{ \text{definition of } Det' \text{ (Property 2.107) and } Useful_s,\ Useful_f \text{ (Remark 2.100)} \} \\
& Det'(M_1^R) \wedge Useful_s(M_1^R) \\
\Leftarrow & \quad \{ \text{Property 2.107: } Det'(M) \Leftarrow Det(M) \} \\
& Det(M_1^R) \wedge Useful_s(M_1^R)
\end{array}$$

The required condition on M_1 can be established by (writing reversal as a prefix function) $M_1 = (R \circ useful_s \circ subset \circ R)(M_0)$.

The complete minimization algorithm (for any ε-*free* $M_0 \in FA$) is

$$M_2 = (useful_s \circ subset \circ R \circ useful_s \circ subset \circ R)(M_0)$$

This algorithm was originally given by Brzozowski in [Brzo62]. A generalization of the algorithm was independently derived by Kameda and Weiner [KW70] just after Brzozowski's presentation. The origin of this algorithm was obscured when Jan van de Snepscheut presented the algorithm in his Ph.D dissertation [vdSn85], where the algorithm is attributed to a private communication from Professor Peremans of the Eindhoven University of Technology. Peremans had originally found the algorithm in an article by Mirkin [Mirk65]. Although Mirkin does cite a paper by Brzozowski [Brzo64], it is not clear whether Mirkin's work was influenced by Brzozowski's work on minimization. Jan van de Snepscheut's recent book [vdSn93] describes the algorithm, but provides neither a history nor citations (other than his dissertation) for this algorithm.

7.3 Minimization by equivalence of states

In this section, we lay the foundations for those algorithms which determine the set of equivalent states. Let $M = (Q, V, T, \emptyset, S, F)$ be a *Complete DFA*; this particular *DFA* will be used throughout this section. We also assume that all of the states of M are start-reachable, that is $\mathit{Useful}_s(M)$. Since M is deterministic and *Complete*, we will also take the transition relation to be the total function $T \in Q \times V \longrightarrow Q$ instead of $T \in Q \times V \longrightarrow \mathcal{P}(Q)$.

In order to minimize the *DFA* M, we compute an equivalence relation $E \sqsubseteq Q \times Q$. In the following section, we will consider the definition of this equivalence relation.

7.3.1 The equivalence relation on states

Given $\mathit{Minimal}_C(M) \equiv \mathit{Min}_C(M)$, we will derive algorithms which ensure $\mathit{Minimal}_C(M)$. For the remainder of this chapter, we only consider ensuring

$$(\forall\ p, q : p \in Q \land q \in Q \land p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q))$$

The other two conjuncts of $\mathit{Minimal}_C(M)$, $\mathit{Useful}_s(M) \land \mathit{Complete}(M)$, are trivial to ensure and are left to the reader.

In order to minimize the *DFA* M, we compute equivalence relation E (on state set Q) such that:

$$(p, q) \in E \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q))$$

(This relation is not to be confused with the ε-transition relation of an *FA* which is also a binary relation on states. The name E has been used to signify 'equivalence'.) Since this is an equivalence relation, we are really interested in unordered pairs of states. It is notationally more convenient to use ordered pairs instead of unordered pairs.

When two states are found to be equivalent under E, the states (and their transitions) can be identified, thereby shrinking the automaton.

In order to compute relation E, we need a property of function $\overrightarrow{\mathcal{L}}$. (This is an intuitive recursive definition of $\overrightarrow{\mathcal{L}}$ which follows from a recursive definition of T^*.)

Property 7.1 (Function $\overrightarrow{\mathcal{L}}$): Function $\overrightarrow{\mathcal{L}}$ satisfies

$$\overrightarrow{\mathcal{L}}(p) = (\cup\ a : a \in V : \{a\} \cdot \overrightarrow{\mathcal{L}}(T(p, a))) \cup (\textbf{if } p \in F \textbf{ then } \{\varepsilon\} \textbf{ else } \emptyset \textbf{ fi})$$

□

This allows us to give an alternate (but equivalent) characterization of equivalence of states.

Property 7.2 (Equivalence of states): Given the above property, we can rewrite E into a recursive form. Relation E is the greatest equivalence relation (on Q) such that

$$(p,q) \in E \equiv (p \in F \equiv q \in F) \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in E)$$

□

We will shortly present E as the greatest fixed point of a continuous function.

Definition 7.3 (Equivalence relations on Q): We define ER_Q to be the set of all equivalence relations on state set Q. □

Property 7.4 (ER_Q): The set ER_Q is a lattice under the refinement ($\sqsubseteq$) ordering, with least element I_Q and greatest element $Q \times Q$. □

Definition 7.5 (Function g): Define function $g \in ER_Q \longrightarrow ER_Q$ as $g(H) =$

$$\{\, (p,q) \mid (p,q) \in H \wedge (p \in F \equiv q \in F) \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in H) \,\}$$

This function is continuous on the lattice of equivalence relations ER_Q. □

Property 7.6 (Fixed point characterization of E): Relation E is the greatest fixed point of function g on the lattice of equivalence relations E_Q. More formally

$$E = (\mathbf{MAX}_{\sqsubseteq}\ H : H \sqsubseteq Q \times Q \wedge H = g(H) : H)$$

Note that $g(H) \sqsubseteq H$. □

For more on this type of fixed point characterization, see [PTB85].

Remark 7.7: Any fixed point of the equivalence in Property 7.2 can be used. In order to *minimize* the automaton (instead of simply shrinking it), the *greatest* fixed point is desired.
□

Property 7.8 (Computing E): Since g is continuous and the lattice is finite, we can compute E by successive applications of function g, beginning with the 'top' equivalence relation $\top = Q \times Q$. We use the following notation to refer to the result of each step (in the computation of E) $E_k = g^{k+1}(\top)$ (for $k \geq 0$).

We can already make the first step, by noting that $E_0 = g(\top) = (Q \setminus F)^2 \cup F^2$. In this case, we can simplify the definition of g (since for all $(p,q) \in E_0$, $p \in F \equiv q \in F$) to give

$$g'(H) = \{\, (p,q) \mid (p,q) \in H \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in H) \,\}$$

Note that $E_k = g'^k(E_0)$. □

Remark 7.9: We can also give an intuitive explanation of equivalence relation E_k at each step of computing the fixed point. A pair of states p, q are said to be k-equivalent (written $(p, q) \in E_k$) if and only if there is no string $w : |w| \leq k$ such that $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$. As a consequence, p and q are k-equivalent ($k > 0$) if and only if

- they are both final or both non-final, and
- for all $a \in V$, $T(p, a)$ and $T(q, a)$ are $(k-1)$-equivalent (by the definitions of $\overrightarrow{\mathcal{L}}$ and T^*).

□

Property 7.10 (E as a greatest fixed point): We can also cast the recursive definition of E as the greatest solution of a set of (perhaps mutually recursive) equivalences. We wish to obtain a set of $|Q|^2$ equivalences with left sides $f_{p,q}$ (for states $p, q \in Q$). When a solution to the system of equivalences is found, we will have $f_{p,q} \equiv (p, q) \in E$. We define each of the $|Q|^2$ equations as follows (where $p, q \in Q$):

$$f_{p,q} \equiv (p \in F \equiv q \in F) \wedge (\forall\ a : a \in V : f_{T(p,a),T(q,a)})$$

The fixed point approximation begins with all of the $f_{p,q} \equiv \mathit{true}$. At each step in the approximation, any one of the equations can be updated, bringing the entire system closer to a solution. Unlike the fixed point approximation sequence outlined in Property 7.8, the relation given by the $f_{p,q}$ may not be an equivalence relation at every step in finding a solution to the system; the final solution is, however, the equivalence relation E. An interesting property (which we will not prove here) is that once an $f_{p,q}$ has become *false*, it will not become *true* at a later step in the approximation sequence (this property is similar to $g(H) \sqsubseteq H$ given in Property 7.6). Algorithms that make use of this method of computing E are given in Sections 7.4.2–7.4.5 and 7.4.7. This approach is equivalent to the approach taken in Property 7.8; in can, however, be implemented very efficiently in some cases. □

All previously known algorithms compute E by successive approximation from above (with respect to $\sqsubseteq$) — a standard approach for computing a greatest fixed point. A new algorithm in Section 7.4.7 computes E by successive approximation from below. In that section, the practical importance of this is explained.

7.3.2 Distinguishability

It is also possible to compute E by first computing its complement $D = \neg E$. Relation D (called the distinguishability relation on states) is required to satisfy

$$(p, q) \in D \equiv (\overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q))$$

Definition 7.11 (Distinguishability of states): D is the least (under set containment[3], $\subseteq$) relation such that

$$(p, q) \in D \equiv (p \in F \not\equiv q \in F) \vee (\exists\ a : a \in V : (T(p, a), T(q, a)) \in D)$$

$\square$

Property 7.12 (Approximating D): As with equivalence relation E, relation D can be computed by successive approximations (for $k \geq 0$)

$$(p, q) \in D_{k+1} \equiv (p, q) \in D_k \vee (\exists\ a : a \in V : (T(p, a), T(q, a)) \in D_k)$$

with $D_0 = \neg E_0 = ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$. For all $k \geq 0$ we have $D_k = \neg E_k$. We could have started with the complement $\emptyset$ of $\top$ (which we used as our starting point for computing fixed point E); for efficiency reasons we would start with D_0 in practice. We also have the property that $D_{k+1} \supseteq D_k$ for $k \geq 0$. $\square$

Remark 7.13: As with E_k, an intuitive explanation of D_k is useful. A pair of states p, q are said to be k-distinguished (written $(p, q) \in D_k$) if and only if there is a string $w : |w| \leq k$ such that $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$. As a consequence, p and q are k-distinguished ($k > 0$, some authors say k-distinguishable) if and only if

- one is final and the other is non-final, or
- there exists $a \in V$ such that $T(p, a)$ and $T(q, a)$ are $(k-1)$-distinguished.

$\square$

7.3.3 An upperbound on the number of approximation steps

We can easily place an upperbound on the number of steps in the computation of E. (This is not the same as the complexity of computing E; instead, we show the number of steps required in an approximating sequence while computing E.)

Let E_j be the greatest fixed point of the equation defining E. We have the sequence of approximations (where I_Q is the identity relation on states):

$$E_0 \supset E_1 \supset \cdots \supset E_j \supseteq I_Q$$

The indices of some of the equivalence relations in the approximation sequence are known: $\sharp I_Q = |Q|$ and $\sharp E_0 \leq 2$. We can deduce that:

$$\sharp E_0 < \sharp E_1 < \cdots < \sharp E_j \leq \sharp I_Q = |Q|$$

In the case that $\sharp E_0 = 0$ (when $Q = \emptyset$), we have that E_0 is the greatest fixed point. In the case that $\sharp E_0 = 1$, either all states are final states, or all states are non-final ones; in

[3]Here, $\subseteq$ denotes normal set containment; refinement does not apply since D is not necessarily an equivalence relation.

both cases E_0 is the greatest fixed point. In the case that $\sharp E_0 = 2$, we have $i + 2 \leq \sharp E_i$ (for all i). Since $j + 2 \leq \sharp E_j \leq \sharp I_Q = |Q|$ we get $j \leq |Q| - 2$. This gives an upperbound of $(|Q| - 2)\, \mathbf{max}\, 0$ steps for the computation (starting at E_0) of the greatest fixed point E_j (using the approximating sequence given in Property 7.8).

A consequence of this upperbound is that $E = E_{(|Q|-2)\,\mathbf{max}\,0}$. As we shall see later, this can lead to some efficiency improvements to algorithms computing E. This result is also noted by Wood in [Wood87, Lemma 2.4.1] and by Brzozowski and Seger in [BS95, Theorem 10.7]. This upperbound also holds for computing D and $[Q]_E$ by approximation.

7.3.4 Characterizing the equivalence classes of E

In this section, we give a characterization of the set $[Q]_E$, the set of equivalence classes of Q under E. The set $[Q]_E$ is the largest (under $\sqsubseteq$) paritition P of Q such that

$$(\forall\ Q_0 : Q_0 \in P : (\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p, q) \in E))$$

Our derivation proceeds as follows:

$$\begin{array}{ll}
 & (\forall\ Q_0 : Q_0 \in P : (\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p, q) \in E)) \\
\equiv & \quad \{\text{ Property 7.2 }\} \\
 & (\forall\ Q_0 : Q_0 \in P : \\
 & \quad (\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p \in F \equiv q \in F) \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in E))) \\
\equiv & \quad \{\text{ move } a \text{ to outer quantification }\} \\
 & (\forall\ Q_0, a : Q_0 \in P \wedge a \in V : \\
 & \quad (\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p \in F \equiv q \in F) \wedge (T(p,a), T(q,a)) \in E)) \\
\equiv & \quad \{\text{ introduced equivalence class } Q_1 \text{ explicitly }\} \\
 & (\forall\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \\
 & \quad (\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p \in F \equiv q \in F) \wedge (T(p,a) \in Q_1 \equiv T(q,a) \in Q_1)))
\end{array}$$

Given the last line above, $[Q]_E$ is the largest (under $\sqsubseteq$) partition P such that $P \sqsubseteq \{Q\}$ and for all $Q_0, Q_1 \in P, a \in V$:

$$(\forall\ p, q : p \in Q_0 \wedge q \in Q_0 : (p \in F \equiv q \in F) \wedge (T(p,a) \in Q_1 \equiv T(q,a) \in Q_1))$$

As with the sequence used to compute E, we can make the first approximation step, leading to a simpler characterization of $[Q]_E$. To make this more readable, we define an auxiliary predicate.

Definition 7.14 (Predicate *Splittable*): In order to make this quantification more concise, we define

$$\mathit{Splittable}(Q_0, Q_1, a) \equiv (\exists\ p, q : p \in Q_0 \wedge q \in Q_0 : (T(p,a) \in Q_1 \not\equiv T(q,a) \in Q_1))$$

□

Given the definition of *Splittable* (and the last line of the derivation above), we can now characterize $[Q]_E$.

Property 7.15 (Characterization of $[Q]_E$): $[Q]_E$ is the largest (under $\sqsubseteq$) partition P such that $P \sqsubseteq [Q]_{E_0}$ and

$$(\forall\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \neg Splittable(Q_0, Q_1, a))$$

This characterization will be used in the computation of $[Q]_E$. □

In [AHU74, p. 157–162], the above characterization of $[Q]_E$ is stated as the coarsest partition problem. That problem can be phrased in one of two ways:

- $[Q]_E$ is the coarsest partition of Q compatible with $\{Q\}$ and functions $T_a \in Q \longrightarrow Q$ (for all $a \in V$).
- $[Q]_E$ is the coarsest partition of Q compatible with $[Q]_{E_0}$ and functions $T_a \in Q \longrightarrow Q$ (for all $a \in V$).

The second formulation includes the first step. In that book, only the single function problem is considered, whereas the above phrasing includes a transition function T_a for each alphabet symbol a.

When $V = \{a\}$ ($|V| = 1$), we have the single transition function $T_a \in Q \longrightarrow Q$. This means that computing $[Q]_E$ is the single-function coarsest partition problem. In [PTB85], a linear time algorithm is given for this problem; the implication is that a *DFA* over a one letter alphabet can be minimized in linear time (instead of $\mathcal{O}(|Q| \log |Q|)$ for Hopcroft's algorithm — the best known general algorithm).

7.4 Algorithms computing E, D, or $[Q]_E$

In this section, we consider several algorithms that compute D, E, or $[Q]_E$. Some of the algorithms are presented in general terms: computing D and E. Since only one of D or E is needed (and not both), such a general algorithm would be modified for practical use to compute only one of the two.

7.4.1 Computing D and E by layerwise approximations

We now present an implementation of the method of computing E outlined in Property 7.8. The following algorithm computes D and E (where variable k is a ghost variable, used only for specifying the invariant)

Algorithm 7.16:

$G, H := D_0, E_0;$
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0;$
$\{$ invariant: $G = D_k \wedge H = E_k$ $\}$
do $H \neq H_{old} \rightarrow$
  $\{ G \neq G_{old} \wedge H \neq H_{old} \}$
  $G_{old}, H_{old} := G, H;$
  $H := g'(H_{old});$
  $G := \neg H;$
  $\{ G = \neg H \}$
  $k := k + 1$
od$\{ G = D \wedge H = E \}$

□

We can expand the definition of function g' and give a more detailed computation of G, yielding

Algorithm 7.17:

$G, H := D_0, E_0;$
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0;$
$\{$ invariant: $G = D_k \wedge H = E_k$ $\}$
do $H \neq H_{old} \rightarrow$
  $\{ G \neq G_{old} \wedge H \neq H_{old} \}$
  $G_{old}, H_{old} := G, H;$
  $G := (\cup\ p, q : (p, q) \in G_{old} \vee (\exists\ a : a \in V : (T(p, a), T(q, a)) \in G_{old}) : \{(p, q)\});$
  $H := (\cup\ p, q : (p, q) \in H_{old} \wedge (\forall\ a : a \in V : (T(p, a), T(q, a)) \in H_{old}) : \{(p, q)\});$
  $\{ G = \neg H \}$
  $k := k + 1$
od$\{ G = D \wedge H = E \}$

□

This algorithm is said to compute D and E layerwise, since it computes the sequences D_k and E_k. The update of G and H in the repetition can be made with another repetition as shown in the program now following.

Algorithm 7.18 (Wood's algorithm — Layerwise computation of D and E):

$G, H := D_0, E_0;$
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0;$
$\{$ invariant: $G = D_k \wedge H = E_k$ $\}$

do $H \neq H_{old} \rightarrow$
  $\{ G \neq G_{old} \wedge H \neq H_{old} \}$
  $G_{old}, H_{old} := G, H;$
  for $(p,q) : (p,q) \in H_{old} \rightarrow$
    if $(\exists\ a : a \in V : (T(p,a), T(q,a)) \in G_{old}) \rightarrow G, H := G \cup \{(p,q)\}, H \setminus \{(p,q)\}$
    $[\!]$ $(\forall\ a : a \in V : (T(p,a), T(q,a)) \in H_{old}) \rightarrow$ **skip**
    fi
  rof;
  $\{ G = \neg H \}$
  $k := k + 1$
od$\{ G = D \wedge H = E \}$

$\square$

The algorithm can be split into two: one computing only D, and the other computing only E. The algorithm computing only E is essentially the algorithm presented by Wood in [Wood87, p. 132]. According to Wood, it is based on the work of Moore [Moor56]. Its running time is $\mathcal{O}(|Q|^3)$. Brauer uses some encoding techniques to provide an $\mathcal{O}(|Q|^2)$ version of this algorithm in [Brau88], while Urbanek improves upon the space requirements of Brauer's version in [Urba89]. None of these variants is given here. The algorithm computing only D does not appear in the literature.

With a little effort this algorithm can be modified to compute $[Q]_E$.

7.4.2 Computing D, E, and $[Q]_E$ by unordered approximation

Instead of computing each E_k (computing E layerwise), we can compute E by considering pairs of states in an arbitrary order (as outlined in Property 7.10). In the following algorithm, H is the set of all pairs of states (p,q) such that $f_{p,q} \equiv \mathit{true}$ at each step; similarly, G is the set of all pairs of states (p,q) such that $f_{p,q} \equiv \mathit{false}$.

Algorithm 7.19:

$G, H := D_0, E_0;$
$\{$ invariant: $G = \neg H \wedge G \subseteq D \}$
do $(\exists\ p, q, a : a \in V \wedge (p,q) \in H : (T(p,a), T(q,a)) \in G) \rightarrow$
  let $p, q : (p,q) \in H \wedge (\exists\ a : a \in V : (T(p,a), T(q,a)) \in G);$
  $\{ (p,q) \in D \}$
  $G, H := G \cup \{(p,q)\}, H \setminus \{(p,q)\}$
od$\{ G = D \wedge H = E \}$

$\square$

At each step, the algorithm chooses a pair $(p,q) \in H$ such that $f_{p,q}$ should not be *true*. This algorithm can be split into one computing only D, and one computing only E.

Remark 7.20: At the end of each iteration step, it may be that H is not an equivalence relation — see Property 7.10. A slight modification to this algorithm can be made by adding the pair (q,p) to H whenever (p,q) is added, and also performing the following assignment before the **od**:

$$H := (\mathbf{MAX}_{\sqsubseteq}\ J : J \subseteq H \wedge J = J^* : J); G := \neg H$$

Addition of this assignment makes the algorithm compute the refinement sequence E_k (see Property 7.10). This assignment may improve the running time of the algorithm if a cheap method of computing the quantified **MAX** is used. The algorithm with this improvement does not appear in the literature. □

Converting the above algorithm to compute $[Q]_E$ yields the following one (which is also given by Aho, Sethi, and Ullman in [ASU86, Alg. 3.6]):

Algorithm 7.21:

$P := [Q]_{E_0};$
$\{\text{ invariant: } [Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \}$
do $(\exists\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \mathit{Splittable}(Q_0, Q_1, a)) \rightarrow$
  let $Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V \wedge \mathit{Splittable}(Q_0, Q_1, a);$
  $Q_0' := \{ p \mid p \in Q_0 \wedge T(p,a) \in Q_1 \};$
  $\{ \neg\mathit{Splittable}(Q_0 \setminus Q_0', Q_1, a) \wedge \neg\mathit{Splittable}(Q_0', Q_1, a) \}$
  $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$
od
$\{ (\forall\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \neg\mathit{Splittable}(Q_0, Q_1, a)) \}$
$\{ P = [Q]_E \}$

□

This algorithm has running time $\mathcal{O}(|Q|^2)$.

7.4.3 More efficiently computing D and E by unordered approximation

We present another algorithm that considers pairs of states in an arbitrary order. This algorithm (which also computes D) consists of two nested repetitions. It is essentially the same as Algorithm 7.19, with a slight change in loop structure.

Algorithm 7.22:

$G, H := D_0, E_0;$
$\{\text{ invariant: } G = \neg H \wedge G \subseteq D \}$
do $(\exists\ p, q, a : a \in V \wedge (p,q) \in H : (T(p,a), T(q,a)) \in G) \rightarrow$
  let $p, a : p \in Q \wedge a \in V \wedge (\exists\ q : (p,q) \in H : (T(p,a), T(q,a)) \in G);$

$\quad\quad$ **for** $q : (p,q) \in H \wedge (T(p,a), T(q,a)) \in G \rightarrow$
$\quad\quad\quad$ $G, H := G \cup \{(p,q)\}, H \setminus \{(p,q)\}$
$\quad\quad$ **rof**
$\quad$ **od**$\{\ G = D \wedge H = E\ \}$

□

This algorithm can also be modified to compute only D or only E.

At the end of each outer iteration step, it may be that H is not an equivalence relation. This can be solved with a symmetrical update of H and an assignment to H as can be done in Algorithm 7.19. This algorithm does not appear in the literature.

Modifying the above algorithm to compute $[Q]_E$ is particularly interesting; the modified algorithm will be used in Section 7.4.5 to derive an algorithm (by Hopcroft) which is the best known algorithm for *DFA* minimization. The modification yields:

Algorithm 7.23:

$P := [Q]_{E_0}$;
$\{$ invariant: $[Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0}$ $\}$
do $(\exists\ Q_1, a : Q_1 \in P \wedge a \in V : (\exists\ Q_0 : Q_0 \in P : \mathit{Splittable}(Q_0, Q_1, a))) \rightarrow$
$\quad$ **let** $Q_1, a : Q_1 \in P \wedge a \in V \wedge (\exists\ Q_0 : Q_0 \in P : \mathit{Splittable}(Q_0, Q_1, a))$;
$\quad$ $P_{old} := P$;
$\quad$ $\{$ invariant: $[Q]_E \sqsubseteq P \sqsubseteq P_{old}$ $\}$
$\quad$ **for** $Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a) \rightarrow$
$\quad\quad$ $Q_0' := \{\ p \mid p \in Q_0 \wedge T(p,a) \in Q_1\ \}$;
$\quad\quad$ $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$
$\quad$ **rof**
$\quad$ $\{\ (\forall\ Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\ \}$
od
$\{\ (\forall\ Q_1, a : Q_1 \in P \wedge a \in V : (\forall\ Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a)))\ \}$
$\{\ P = [Q]_E\ \}$

□

The inner repetition 'splits' each eligible equivalence class Q_0 with respect to pair (Q_1, a). (In actuality, some particular Q_0 will not be split by (Q_1, a) if $\neg \mathit{Splittable}(Q_0, Q_1, a)$.)

7.4.4 An algorithm due to Hopcroft and Ullman

From the definition of D, we see that a pair (p,q) is in D if and only if $p \in F \not\equiv q \in F$ or there is some $a \in V$ such that $(T(p,a), T(q,a)) \in D$. This forms the basis of the algorithm considered in this section. With each pair of states (p,q) we associate a set of pairs of states $L(p,q)$ such that

$$(r,s) \in L(p,q) \Rightarrow ((p,q) \in D \Rightarrow (r,s) \in D)$$

We start with D_0 as our approximation of D. For each pair (p,q) (such that $(p,q) \notin D_0$ — p and q are not already known to be distinguished) we do the following:

- If there is an $a \in V$ such that we know that $(T(p,a), T(q,a)) \in D$ then $(p,q) \in D$. We add (p,q) to our approximation of D, along with $L(p,q)$, and for each $(r,s) \in L(p,q)$ add $L(r,s)$, and for each $(t,u) \in L(r,s)$ add $L(t,u)$, etc.
- If there is no $a \in V$ such that $(T(p,a), T(q,a)) \in D$ is known to be true, then for all $b \in V$ we put (p,q) in the set $L(T(p,b), T(q,b))$ since $(T(p,b), T(q,b)) \in D \Rightarrow (p,q) \in D$. If later it turns out that for some $b \in V$, $(T(p,b), T(q,b)) \in D$, then we will also put $L(T(p,b), T(q,b))$ (including (p,q)) in D.

In our presentation of the algorithm, the invariants given are not sufficient to prove the correctness of the algorithm, but are used to illustrate the method in which the algorithm works.

Algorithm 7.24:

---

for $(p,q) : (p,q) \in (Q \times Q) \rightarrow$
  $L(p,q) := \emptyset$
rof;
$G := D_0$;
$\{$ invariant: $G \subseteq D$
  $\wedge\ (\forall\ p,q : (p,q) \notin D_0 : (\forall\ r,s : (r,s) \in L(p,q) : (p,q) \in D \Rightarrow (r,s) \in D))\ \}$
for $(p,q) : (p,q) \notin D_0 \rightarrow$
  if $(\exists\ a : a \in V : (T(p,a), T(q,a)) \in G) \rightarrow$
    $\mathit{toadd}, \mathit{added} := \{(p,q)\}, \emptyset$;
    $\{$ invariant: $\mathit{toadd} \subseteq D \wedge \mathit{added} \subseteq G \wedge \mathit{toadd} \cap \mathit{added} = \emptyset$
      $\wedge\ \mathit{toadd} \cup \mathit{added} = (\cup\ p,q : (p,q) \in \mathit{added} : L(p,q)) \cup \{(p,q)\}\ \}$
    do $\mathit{toadd} \neq \emptyset \rightarrow$
      let $(r,s) : (r,s) \in \mathit{toadd}$;
      $G := G \cup \{(r,s)\}$;
      $\mathit{toadd}, \mathit{added} := \mathit{toadd} \setminus \{(r,s)\}, \mathit{added} \cup \{(r,s)\}$;
      $\mathit{toadd} := \mathit{toadd} \cup (L(r,s) \setminus \mathit{added})$
    od
  $[\!]$ $(\forall\ a : a \in V : (T(p,a), T(q,a)) \notin G) \rightarrow$
    for $a \in V : T(p,a) \neq T(q,a) \rightarrow$
      $\{\ (T(p,a), T(q,a)) \in D \Rightarrow (p,q) \in D\ \}$
      $L(T(p,a), T(q,a)) := L(T(p,a), T(q,a)) \cup \{(p,q)\}$
    rof
  fi
rof $\{\ G = D\ \}$

□

This algorithm has running time $\mathcal{O}(|Q|^2)$ and is given by Hopcroft and Ullman [HU79, Fig. 3.8]. In [HU79] it is attributed to Huffman [Huff54] and Moore[Moor56]. In their description, Hopcroft and Ullman describe L as mapping each pair of states to a list of pairs of states. The list data-type is not required here, and a set is used here instead.

It is possible to modify the above algorithm to compute E. Such an algorithm does not appear in the literature.

7.4.5 Hopcroft's algorithm to compute $[Q]_E$ efficiently

We now derive an efficient algorithm due to Hopcroft [Hopc71]. This algorithm has also been derived by Gries [Grie73]. This algorithm presently has the best known running time analysis of all *DFA* minimization algorithms.

We begin with Algorithm 7.23. Recall that the inner repetition 'splits' each equivalence class Q_0 with respect to pair (Q_1, a). An observation (due to Hopcroft) is that once all equivalence classes have been split with respect to a particular (Q_1, a), no equivalence classes need to be split with respect to the same (Q_1, a) on any subsequent iteration step of the outer repetition [Hopc71, pp. 190–191], [Grie73, Lemma 5]. The observation is simple to prove: the equivalence classes never grow in size, and we need only prove that (for all equivalence classes Q_0):

$$\neg Splittable(Q_0, Q_1, a) \Rightarrow (\forall\, Q_0' : Q_0' \subseteq Q_0 : \neg Splittable(Q_0', Q_1, a))$$

We can use this fact to maintain a set L of pairs, where each pair consists of an equivalence class and an alphabet symbols. We will then split the equivalence classes with respect to elements of L. In the original presentations of this algorithm [Hopc71, Grie73], L is a list. As this is not necessary, we retain L's type as a set; in Chapter 11 (User function 11.7) we will see an efficient encoding of L as an array.

$P := [Q]_{E_0}$;
$L := P \times V$;
{ invariant: $[Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \wedge L \subseteq (P \times V)$
  $\wedge\ L \supseteq \{ (Q_1, a) \mid (Q_1, a) \in (P \times V) \wedge (\exists\, Q_0 : Q_0 \in P : Splittable(Q_0, Q_1, a)) \}$
  $\wedge\ L = \emptyset \Rightarrow P = [Q]_E$ }
do $L \neq \emptyset \rightarrow$
  let $Q_1, a : (Q_1, a) \in L$;
  $P_{old} := P$;
  $L := L \setminus \{(Q_1, a)\}$;
  { invariant: $[Q]_E \sqsubseteq P \sqsubseteq P_{old}$ }
  for $Q_0 : Q_0 \in P_{old} \wedge Splittable(Q_0, Q_1, a) \rightarrow$
    $Q_0' := \{ p \mid p \in Q_0 \wedge T(p, a) \in Q_1 \}$;
    $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$;
    for $b : b \in V \rightarrow$
      if $(Q_0, b) \in L \rightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q_0', b), (Q_0 \setminus Q_0', b)\}$

$$
\begin{array}{l}
\quad\quad\quad\quad [\!]\ (Q_0, b) \notin L \rightarrow L := L \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\} \\
\quad\quad\quad\quad \mathbf{fi} \\
\quad\quad\quad \mathbf{rof} \\
\quad\quad \mathbf{rof} \\
\quad\quad \{\ (\forall\ Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\ \} \\
\quad \mathbf{od}\{\ P = [Q]_E\ \}
\end{array}
$$

The innermost update of L is intentionally clumsy and will be used to arrive at the algorithm given by Hopcroft and Gries. In the update of set L, if $(Q_0, b) \in L$ (for some $b \in V$) and Q_0 has been split into $Q_0 \setminus Q'_0$ and Q'_0 then (Q_0, b) is replaced (in L) by $(Q_0 \setminus Q'_0, b)$ and (Q'_0, b).

Another observation due to Hopcroft is shown in the following lemma (which is taken from [Hopc71, pp. 190–191] and [Grie73, Lemma 6]).

Lemma 7.25 (Redundant splitting): Splitting an equivalence class with respect to any two of (Q_0, b), (Q'_0, b), and $(Q_0 \setminus Q'_0, b)$ (where $Q'_0 \subseteq Q_0$) is the same as splitting the equivalence class with respect to all three.

Proof:

We only prove that: if an equivalence class $\hat{Q}$ has been split with respect to (Q_0, b) and (Q'_0, b), then it need not be split with respect to $(Q_0 \setminus Q'_0, b)$. The two remaining cases can be proven analogously.

$$
\begin{array}{ll}
 & \neg \mathit{Splittable}(\hat{Q}, Q_0, b) \wedge \neg \mathit{Splittable}(\hat{Q}, Q'_0, b) \\
\equiv & \quad \{\ \text{De Morgan}\ \} \\
 & \neg(\mathit{Splittable}(\hat{Q}, Q_0, b) \vee \mathit{Splittable}(\hat{Q}, Q'_0, b)) \\
\equiv & \quad \{\ \text{definition of } \mathit{Splittable}\ \} \\
 & \neg((\exists\ p, q : p, q \in \hat{Q} : T(p, b) \in Q_0 \not\equiv T(q, b) \in Q_0) \\
 & \quad \vee (\exists\ p, q : p, q \in \hat{Q} : T(p, b) \in Q'_0 \not\equiv T(q, b) \in Q'_0)) \\
\equiv & \quad \{\ \text{combine existential quantifications}\ \} \\
 & \neg(\exists\ p, q : p, q \in \hat{Q} : (T(p, b) \in Q_0 \not\equiv T(q, b) \in Q_0) \vee (T(p, b) \in Q'_0 \not\equiv T(q, b) \in Q'_0)) \\
\Rightarrow & \quad \{\ Q'_0 \subseteq Q_0\ \} \\
 & \neg(\exists\ p, q : p, q \in \hat{Q} : T(p, b) \in Q_0 \setminus Q'_0 \not\equiv T(q, b) \in Q_0 \setminus Q'_0) \\
\equiv & \quad \{\ \text{definition of } \mathit{Splittable}\ \} \\
 & \neg \mathit{Splittable}(\hat{Q}, Q_0 \setminus Q'_0, b)
\end{array}
$$

□

Given the lemma above, for efficiency reasons we therefore choose the smallest two of the three (comparing $|Q_0|$, $|Q'_0|$, and $|Q_0 \setminus Q'_0|$) in the update of set L. If $(Q_0, b) \notin L$, then splitting has already been done with respect to (Q_0, b) and we add either (Q'_0, b) or $(Q_0 \setminus Q'_0, b)$ (whichever is smaller) to L. On the other hand, if $(Q_0, b) \in L$, then splitting has not yet been done and we remove (Q_0, b) from L and add (Q'_0, b) and $(Q_0 \setminus Q'_0, b)$ instead.

Lastly, we observe that by starting with $P = [Q]_{E_0} = \{Q \setminus F, F\}$ we have already split Q. As a result, we need only split with respect to either $(Q \setminus F, b)$ or (F, b) (for all $b \in V$) [Hopc71, pp. 190–191], [Grie73, Lemma 7]. This gives the algorithm:

Algorithm 7.26 (Hopcroft):

$P := [Q]_{E_0}$;
if $|F| \leq |Q \setminus F| \to L := \{F\} \times V$
$[\!]$ $|F| > |Q \setminus F| \to L := \{Q \setminus F\} \times V$
fi;
$\{$ invariant: $[Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \wedge L \subseteq (P \times V)$
$\quad\wedge\ L = \emptyset \Rightarrow P = [Q]_E\ \}$
do $L \neq \emptyset \to$
  let $Q_1, a : (Q_1, a) \in L$;
  $P_{old} := P$;
  $L := L \setminus \{(Q_1, a)\}$;
  $\{$ invariant: $[Q]_E \sqsubseteq P \sqsubseteq P_{old}\ \}$
  for $Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a) \to$
    $Q'_0 := \{\, p \mid p \in Q_0 \wedge T(p, a) \in Q_1 \,\}$;
    $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, Q'_0\}$;
    for $b : b \in V \to$
      if $(Q_0, b) \in L \to L := L \setminus \{(Q_0, b)\} \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\}$
      $[\!]$ $(Q_0, b) \notin L \to$
        if $|Q'_0| \leq |Q_0 \setminus Q'_0| \to L := L \cup \{(Q'_0, b)\}$
        $[\!]$ $|Q'_0| > |Q_0 \setminus Q'_0| \to L := L \cup \{(Q_0 \setminus Q'_0, b)\}$
        fi
      fi
    rof
  rof
  $\{\ (\forall\ Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\ \}$
od$\{\ P = [Q]_E\ \}$

□

The running time analysis of this algorithm is complicated and is not discussed here. It is shown by both Gries and Hopcroft that it is $\mathcal{O}(|Q| \log |Q|)$, [Grie73, Hopc71]. A simpler derivation of the running time of this algorithm is given by Keller and Paige in [KP95].

A very different derivation of this algorithm is given by Keller and Paige in [KP95]. In their paper, the algorithm is presented as an example of program derivation in their new framework. Interestingly, their derivation is not only clear, but they also manage to derive a new version which is more space efficient than the one presented here.

7.4.6 Computing $(p, q) \in E$

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences $f_{p,q}$ (from Property 7.10) into a functional program. If the definition were to be used directly as a functional program, there is the possibility of non-termination. In order for the functional program to work, it takes a third parameter along with the two states.

The following program, similar to the one presented in [t-Ei91], computes relation E pointwise; an invocation $equiv(p, q, \emptyset)$ determines whether states p and q are equivalent. It assumes that two states are equivalent (by placing the pair of states in S, the third parameter) until shown otherwise.

```
func equiv(p, q, S) →
    if {p, q} ∈ S → eq := true
    ▯ {p, q} ∉ S →
        eq := (p ∈ F ≡ q ∈ F);
        eq := eq ∧ (∀ a : a ∈ V : equiv(T(p, a), T(q, a), S ∪ {{p, q}}))
    fi;
    return eq
cnuf
```

The $\forall$ quantification can be implemented using a repetition

---

```
func equiv(p, q, S) →
    if {p, q} ∈ S → eq := true
    ▯ {p, q} ∉ S →
        eq := (p ∈ F ≡ q ∈ F);
        for a : a ∈ V →
            eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}})
        rof
    fi;
    return eq
cnuf
```

The correctness of this program is shown in [t-Ei91]. Naturally, the guard eq can be used in the repetition (to terminate the repetition when $eq \equiv \mathit{false}$) in a practical implementation. This optimization is omitted here for clarity.

There are a number of methods for making this program more efficient. From Section 7.3.3 recall that $E = E_{(|Q|-2)\,\mathbf{max}\,0}$. We add a parameter k to function $equiv$ such that an invocation $equiv(p, q, \emptyset, k)$ returns $(p, q) \in E_k$ as its result. It follows that an invocation $equiv(p, q, \emptyset, (|Q| - 2)\,\mathbf{max}\,0)$ returns $(p, q) \in E$ as its result. The recursion depth is bounded by $(|Q| - 2)\,\mathbf{max}\,0$. The new function is

---

func $equiv(p, q, S, k) \rightarrow$
  if $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F)$;
    for $a : a \in V \rightarrow$
      $eq := eq \wedge equiv(T(p, a), T(q, a), S \cup \{\{p, q\}\}, k - 1)$
    rof
  fi;
  return eq
cnuf

The third parameter S is made a global variable, improving the efficiency of this algorithm in practice. As a result, *equiv* is no longer a functional program in the sense that it now makes use of a global variable. The correctness of this transformation is shown in [t-Ei91]. We assume that S is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0)$ returns $(p, q) \in E$; after such an invocation $S = \emptyset$.

Algorithm 7.27 (Pointwise computation of E):

---

func $equiv(p, q, k) \rightarrow$
  if $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F)$;
    $S := S \cup \{\{p, q\}\}$;
    for $a : a \in V \rightarrow$
      $eq := eq \wedge equiv(T(p, a), T(q, a), k - 1)$
    rof;
    $S := S \setminus \{\{p, q\}\}$
  fi;
  return eq
cnuf

$\square$

The procedure *equiv* can be memoized[4] to further improve the running time in practice. This algorithm does not appear in the literature.

7.4.7 Computing E by approximation from below

This latest version of function *equiv* can be used to compute E and D (assuming I_Q is the identity relation on states, and S is the global variable used in Algorithm 7.27):

Algorithm 7.28 (Computing E from below):

$S, G, H := \emptyset, \emptyset, I_Q;$
$\{$ invariant: $G \subseteq D \wedge H \subseteq E \}$
do $(G \cup H) \neq Q \times Q \rightarrow$
  let $p, q : (p, q) \in ((Q \times Q) \setminus (G \cup H));$
  if $equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow H := H \cup \{(p, q)\}$
  $[\!]$ $\neg equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow G := G \cup \{(p, q)\}$
  fi
od$\{ G = D \wedge H = E \}$

$\square$

Further efficiency improvements can be made as follows:

- We change the initialization of G to $G := D_0$, equivalently $G := ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$.
- As in Remark 7.20, we make use of the fact that $E = E^*$; obviously E is symmetrical, halving the required amount of computation — we can update H with the pair (q, p) whenever we add (p, q). H can also be updated at each iteration step by $H := H^*$. In Chapter 11 we will describe an implementation of the above algorithm that uses data-structures particularly suited to the $*$-closure operation.
- Make use of the facts that

  $$\begin{array}{lll} (p,q) \notin E & \Rightarrow & (\forall\, r, s : r \in Q \wedge s \in Q \\ & & \quad \wedge\, (\exists\, w : w \in V^* : T^*(r,w) = p \wedge T^*(s,w) = q) : (r,s) \notin E) \\ (p,q) \in E & \Rightarrow & (\forall\, w : w \in V^* : (T^*(p,w), T^*(q,w)) \in E) \end{array}$$

  The first implication states that if p, q are two distinguished states, and r, s are two states such that there is $w \in V^*$ and $T(r, w) = p \wedge T(s, w) = q$, then r, s are also distinguished. The second implication states that if p, q are two equivalent states,

[4]Memoizing a functional program means that the parameters and the result of each invocation are tabulated in memory; if the function is invoked again with the same parameters, the tabulated return value is fetched and returned without recomputing the result.

and r, s are two states such that there is $w \in V^*$ and $T(p, w) = r \wedge T(q, w) = s$, then r, s are also equivalent.

The first of the two above facts is particularly difficult to implement in practice, since the transition functions would have to be traversed from right to left; this is backwards for most implementations, as is shown in Chapter 11.

Although this algorithm has worse running time than the $\mathcal{O}(|Q| \log |Q|)$ of Hopcroft's algorithm [Hopc71, Grie73], in practice the difference is often not significant (see Chapter 15 where a non-memoizing version of *equiv* was used). This algorithm has a significant advantage over all of the known algorithms: although function *equiv* computes E pointwise from above (with respect to $\sqsubseteq$, refinement), the main program computes E from below (with respect to $\subseteq$, normal set inclusion[5]). As such, any intermediate result H in the computation of E is usable in (at least partially) reducing the size of an automaton; all of the other algorithms presented have unusable intermediate results. This property has use in reducing the size of automata when the running time of the minimization algorithm is restricted for some reason (for example, in real-time applications).

7.5 Conclusions

The conclusions about minimization algorithms are:

- A derivation of Brzozowski's minimization algorithm was presented. This derivation proved to be easier to understand than either the original derivation (by Brzozowski), or the derivations given by Kameda and by van de Snepscheut. A brief history of the minimization algorithm was presented, hopefully resolving some misattributions of its discovery.

- The definition of equivalence (relation E) and distinguishability (relation D) as fixed points of certain functions proved easier to understand than many text-book presentations.

- The fixed point characterization of E made it particularly easy to calculate an upperbound on the number of approximation steps required to compute E (or D). This upperbound later proved useful in determining the running time of some of the algorithms, and also in making efficiency improvements to the pointwise algorithm.

- The definition of E as a greatest fixed point helped to identify the fact that all of the (previously) known algorithm computed E from above (with respect to refinement). As such, all of these algorithms have intermediate results that are not usable in minimizing the finite automaton.

[5]This is set inclusion, as opposed to refinement, since the intermediate result H may not be an equivalence relation during the computation.

- We successfully presented all of the well-known text-book algorithms in the same framework. Most of them were shown to be essentially the same, with minor differences in their loop structures. One exception was Hopcroft and Ullman's algorithm [HU79], which has an entirely different loop structure. The presentation of that algorithm (with invariants) in this chapter is arguably easier to understand than the original presentation. Our presentation highlights the fact that the main data-structure in the algorithm need not be a list — a set suffices.

- Hopcroft's minimization algorithm [Hopc71] was originally presented in a style that is not very understandable. As with Gries's paper [Grie73], we strive to derive this algorithm in a clear and precise manner. The presentation in this chapter highlights two important facts: the beginning point for the derivation of this algorithm is one of the easily understood straightforward algorithms; and, the use of a list data-structure in both Hopcroft's and Gries's presentation of this algorithm is not necessary — a set can be used instead.

- This chapter presented several new minimization algorithms, many of which were variations on the well-known algorithms. Two of the new algorithms (presented in Sections 7.4.6 and 7.4.7) are not derived from any of the well-known algorithms, and are significant in their own right.
  - An algorithm was presented that computes the relation E in a pointwise manner. This algorithm was refined from an algorithm used to determine the structural equivalence of types. Several techniques played important roles in the refinement:
    * The upperbound on the number of steps required to compute E was used to improve the algorithm by limiting the number of pairs of states that need to be considered in computing E pointwise.
    * Memoization of the functional-program portion of the algorithm can be used to reduce the amount of redundant computation.
  - A new algorithm was presented, that computes E from below. This algorithm makes use of the pointwise computation of E to construct and refine an approximation of E. Since the computation is from below, the intermediate results of this algorithm are usable in (at least partially) reducing the size of the *DFA*. This can be useful in applications where the amount of time available for minimization of the *DFA* is limited (as in real-time applications). In contrast, all of the (previously) known algorithms have unusable intermediate results.

Part III

The implementations

Chapter 8

Designing and implementing class libraries

In this part of the dissertation, we will consider the design and implementation of class libraries of the algorithms derived in Part II. In this chapter, we briefly discuss some of the issues involved in designing, implementing, and presenting class libraries (or *toolkits*). The following description of a toolkit is taken from [GHJV95, p. 26]:

> A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They are the object-oriented equivalent of subroutine libraries.

We will use the terms *class library, library,* and *toolkit* interchangeably. We will also use the term *client* to refer to a program that makes use of classes in the toolkit, or the author of such a program. The important aspects and design goals of a toolkit are:

- Toolkits do not provide a user interface. (Toolkits that do provide user interfaces should be placed in the category of 'application program'.)
- The classes in the toolkit must have a coherent design, meaning that they are designed and coded in the same style. They have a clear relationship and a logical class hierarchy.
- The client interface to the library must be easily understood, permitting clients to make use of the library with a minimum of reading.
- The efficiency of using the classes in the toolkit must be comparable to hand-coded special-purpose routines — the toolkit must be applicable to production quality software.
- To provide an educational use for the toolkits, and to allow clients to easily modify classes and member functions, the method of implementation must be clear and understandable.

The toolkits described in this part are implemented in the C++ programming language, which was chosen because of its widespread availability. Efforts were made to refrain from using obscure features of C++ (such as RTTI or name spaces), or language features not easily found in other object-oriented programming languages (such as multiple-inheritance).

Throughout this part of the dissertation, we assume that the reader is familiar with the C++ language and object-oriented terminology (especially the C++ variety). For the uninitiated, the C++ literature can be divided into three groups: introductions to C++ [Lipp91, Stro91], advanced C++ [Copl92, SE90], C++ tricks and techniques [HN92, MeyS92, Murr93], and (of course) the draft standard. Introductions to object-oriented design and programming can be found in [Booc94, Budd91, MeyB88, Tali94, Wein95].

The general process of library design will not be described here, as there is a large body of literature discussing this issue. The following books are of particular relevance:

- [GHJV95, Souk94] discuss 'design patterns' (not related to our pattern matching problem) which are used heavily in library design.
- [CE95], [Stro91, Chapter 13] and [Stro94, Chapter 8] provide a general discussion of C++ library design.
- [MeyB94] is an excellent treatment of the design of a number of loosely coupled libraries in the Eiffel programming language. Many of the concepts and techniques discussed in the book are broadly applicable to C++ as well.
- [Plau95, Teal93] discuss the design and implementation of specific C++ libraries — the standard C++ library[1] and the IOStreams (input and output) class libraries respectively.

The toolkit related terms that we will use in this part are defined as follows.

Definition 8.1 (Toolkit terminology): We define the following types of classes:

User: A class intended for use by a client program.

Client: A class defined in the client program.

Implementation: A class defined in the toolkit for exclusive use by the toolkit. The class is used to support the implementation of the client classes.

Foundation: Those implementation classes which are simple enough to be reused in other (perhaps unrelated) class libraries.

Interface: An abstract (pure virtual) class which is declared to force a particular public interface upon its inheritance descendants.

Base: An inheritance ancestor of a particular class.

Derived: An inheritance descendant of a particular class.

Note that the terms *base* and *derived* are relative. □

[1] Plauger's book considers the implementation of an early, and now defunct, draft of the standard library

8.1 Motivations for writing class libraries

There are a number of motivations for creating the class libraries (which will be described in Chapters 9, 10, and 11):

- Until now, few general purpose toolkits of pattern matchers or finite automata construction algorithms existed. The finite automata toolkits that do exist are not intended for general use in production quality software.
- The level of coherence normally required to implement a toolkit was not previously possible. The literature on pattern matching algorithms was scattered and in some places incomplete. With the construction of the taxonomies, all of the algorithms are described in a coherent fashion, allowing us to base the class library structures on the taxonomy structure.
- The uniformity of implementation that was possible (given the taxonomies) had two important effects:
  - Clients need not examine the source code in order to make a decision on which class to use; the quality of the implementations of each of the pattern matchers is roughly the same.
  - Uniformity gives greater confidence in the accuracy of relative performance comparing different algorithms (as is presented in Part IV of this dissertation).
- The toolkits and the taxonomies can serve as examples of implementation techniques for class libraries; in particular methods for organizing template classes[2] and class hierarchies.
- Implementing the abstract algorithm can be painless and fun, given the taxonomy presentation of the algorithms and their correctness arguments.

8.2 Code sharing

One of the main aims of object-oriented programming is that it permits, and even encourages, code sharing (or code reuse). The code reuse in object-oriented programming corresponds neatly with the factoring of common parts of algorithms in the taxonomies.

Although code sharing can be achieved in a number of ways, in this section we discuss four techniques which could have been used in the design of the toolkits. The first discussion centres around the use of base classes (with virtual member functions) versus templates. The second discussion concerns the use of composition versus protected inheritance.

[2] We use the term *template class*, as opposed to *class template* suggested by Carroll and Ellis in [CE95]. Our choice was made to correspond to the term *generic class* used in some other object-oriented programming languages.

8.2.1 Base classes versus templates

A number of the pattern matching objects have common functionality, and it seems wasteful to duplicate the code in each of the specific types of pattern matchers.

The obvious design involves creating a new base class and factoring the common code into the base class. Each of the pattern objects would then inherit from this base, and provide specific virtual member functions to obtain the desired functionality. For example, the Commentz-Walter algorithms all share a common algorithm skeleton; they each have specific shift functions. We could create a CW base class with the functionality of the skeleton, and provide a virtual 'shift distance' member function to obtain the Commentz-Walter variants.

The advantage of this approach is its elegance. It provides a relatively easy to understand class hierarchy, which largely reflects the structure of the taxonomy presented in Chapter 4. Furthermore, a member function which takes (as parameter) a pointer to a CW object need not know which particular variant (of a CW object) the pointer points to, only that the CW object satisfies the general CW functionality. This solution provides code reuse at both the source language and executable image levels. The disadvantage is that it would require a virtual function call for every shift. Indeed, if the same technique was used to factor the common code from the Aho-Corasick variants, it would require a virtual function call for every character of the input string.

The other approach is to create a template class CW, which takes a 'shifter class' as its template (type) parameter. We would then provide a number of such shifter classes, for use as template parameters — each giving rise to one of the Commentz-Walter variants. The primary advantage of this approach is that it is efficient: when used to implement the Aho-Corasick algorithms, each character in the input string will require a non-virtual function call (which may be inlined, unlike virtual function calls). The disadvantages are twofold: pointers to the variants of the CW algorithms are not interchangeable, and code will be generated for each of the CW variants. The code reuse is at the source level, and not at the executable image level.

It is expected that few clients of the toolkits will instantiate objects of different CW classes, for example. A programmer writing an application using pattern matching is more likely to choose a particular type of pattern matcher, as opposed to creating objects of various different types. For this reason, the advantages of the template approach are deemed to outweigh its disadvantages, and we prefer to use it over base classes in the toolkits.

8.2.2 Composition versus protected inheritance

Composition (sometimes called the *has-a* relationship) and protected inheritance (sometimes called the *is-a* relationship) are two additional solutions to code sharing. We illustrate the differences between these two solutions using an example. When implementing a *Set* class, we may wish to make use of an already-existing *Array* class. There are two ways to do this: protected inheritance and composition.

With protected inheritance, class *Set* inherits from *Array* in a protected way. Class *Set* still gets the required functionality from *Array*, but the protected inheritance prevents the *is-a* relation between *Set* and *Array* (that is, we cannot treat a *Set* as an *Array*). The advantage of this approach is that it is elegant, and it is usually the approach taken in languages such as Smalltalk and Objective-C [Budd91]. The disadvantage is that the syntax of C++ places the inheritance clause at the beginning of the class declaration of *Set*, making it plain to all clients of *Set* that it is implemented in terms of *Array*. Furthermore, protected inheritance (and indeed private inheritance) is one of the rarely-used corners of C++, and it is unlikely to be familiar to the average programmer [MeyS92, Murr93].

In a composition approach, an object of class *Set* has (in its private section) an object of class *Array*. The *Set* member functions invoke the appropriate member functions of *Array* to provide the desired functionality. The advantage of this approach is that it places all implementation details in the private section of the class definition. The disadvantage is that it deviates from the accepted practice of inheriting for implementation in some other languages. It is, however, the standard approach in C++. At first glance, it would appear that composition can lead to some inefficiency: in our example, an invocation of a *Set* member function would, in turn, call an *Array* member function. These extra function calls, usually called *pass-throughs*, are frequently eliminated through inlining.

There are no efficiency-based reasons to choose one approach over the other. For this reason, we arbitrarily choose composition because of the potential readability and understandability problems with protected inheritance.

8.3 Coding conventions and performance issues

At this time, coding in C++ presents at least two problems: the language is not yet stable (it is still being standardized) and, correspondingly, the standard class libraries are not yet stable.

In designing the libraries, every effort was made to use only those language features which are well-understood, implemented by most compilers and almost certain to remain in the final language. Likewise, the use of classes from the proposed standard library, or from the Standard Template Library [SL94], was greatly restricted. A number of relatively simple classes (such as those supporting strings, arrays, and sets) were defined from scratch, in order to be free of library changes made by the standardizing committee. A future version of the toolkits will make use of the standard libraries once the International Standards Organization has approved the C++ standard.

In the object-oriented design process, it is possible to go overboard in defining classes for even the smallest of objects — such as alphabet symbols, and the states of a finite automaton. In the interests of efficiency, we draw the line at this level and make use of integers for such basic objects.

Almost all of the classes in the toolkits have a corresponding class invariant member function, which returns *TRUE* if the class is structurally correct, and *FALSE* otherwise. Structural invariants have proven to be particularly useful in debugging and in understanding the code (the structural invariant is frequently a good first place to look when trying to understand the code of a class). For this reason, they have been left in the released code (they can be disabled as described in the next section).

We use a slightly non-traditional way of splitting the source code into files. The public portion of a class declaration is given in a `.hpp` file, while the private parts are included from a `.ppp` file. There is a corresponding `.cpp` file containing all of the out-of-line member function definitions. A `.ipp` file contains member functions which can be inlined for performance reasons. By default the member functions in the `.ipp` file are out-of-line. The inlining can be enabled by defining the macro `INLINING`. To implement such conditional inlining, the `.ipp` file is conditionally included into the `.hpp` or the `.cpp` file. The inlining should be disabled during debugging or for smaller executable images.

8.3.1 Performance tuning

The algorithms implemented in the taxonomy are already highly tuned from an algorithmic point of view. Clients that find the performance inadequate should take the following steps in order until the performance is sufficient:

1. Ensure that assertions are disabled (by defining the `NDEBUG` macro — see [ISO90, Section 7.2]).

2. Enable appropriate compiler optimizations.

3. Define macro `INLINING` to obtain inlining of member functions.

4. Profile the code to determine 'hot-spots'.

5. Inline any out-of-line functions whose call-overhead is contributing to the hot-spots.

6. Define and use special versions of the **new** and **delete** operators for the classes that make extensive use of heap memory (and are contributing to the hot-spots).

7. If a lot of time is spent in copy constructors, convert the offending class to make use of use-counting.

8. When using a class with one or more virtual member functions, and the virtual function calls are causing hot-spots: flatten the class hierarchy, eliminating the virtuality of the functions and convert calls to them into normal (non-virtual) function calls. (Note that this is a last resort, since every effort has been made to reduce the number of virtual function calls, and it involves the modification of the library source code.)

9. Contact me.

8.4 Presentation conventions

In the following chapters, some of the source code of the toolkits will be presented. In the presentation of a particular source file, we typeset the program variables in a roman shape. When the same variable is discussed in running text, we typeset it in an italic shape.

Since composition and templates are preferred over inheritance, very little inheritance is present in the toolkits. As a result, we will not present any inheritance graphs in this dissertation.

Each class is presented in a standard format. The following items appear in the description of a class:

1. The name of the class and whether it is a user-level class (one for use by clients), or an implementation class (for use by other classes within the toolkit).

2. The **Files** clause lists the files in which the class declaration and definition are stored. The names of the files associated with a class are usually the class name, followed by `.hpp` for interface (or header) files, `.ppp` for private section files, `.ipp` for inline member function definitions, and `.cpp` for out-of-line definition files. In this clause, we only mention the root file name (without the suffix). All file names are short enough for use under MS-DOS.

3. The **Description** clause gives a brief description of the purpose of the class.

4. The optional **Implementation** clause outlines the implementation of the class.

5. The optional **Performance** clause gives some suggestions on possible performance improvements to the implementation of the class.

6. The description ends with the □ symbol.

Chapter 9

SPARE Parts: String PAttern REcognition in C++

This chapter contains a description of a C++ pattern matching toolkit known as the SPARE Parts (String PAttern REcognition). Both the client interface and aspects of the design and implementation are considered.

9.1 Introduction and related work

The SPARE Parts is the second generation string pattern matching toolkit from the Eindhoven University of Technology. The first toolkit (called the Eindhoven Pattern Kit, written in C, and described in [Wat94a, Appendix A]) is a procedural library based upon the original taxonomy of pattern matching algorithms [WZ92]. Experience with the toolkit revealed a number of deficiencies, detailed as follows. The rudimentary and explicit memory management facilities in C caused a number of errors in the code, and made it difficult to perform pattern matching over more than one string simultaneously (in separate threads of the program) without completely duplicating the code. While the performance of the toolkit was excellent, some of the speed was due to sacrifices made in the understandability of the client interface.

There are other existing pattern matching toolkits, notably the toolkit of Hume and Sunday [HS91]. Their toolkit consists of a number of implementations of Boyer-Moore type algorithms — organized so as to form a taxonomy of the Boyer-Moore family of algorithms. Their toolkit was primarily designed to collect performance data on the algorithms. As a result, the algorithms are implemented (in C) for speed and they sacrifice some of the safety that would normally be expected of a general toolkit. Furthermore, the toolkit does not include any of the non-Boyer-Moore pattern matching algorithms (other than a brute-force pattern matcher) — most noticeably, there are no multiple keyword pattern matchers.

The SPARE Parts is a completely redesigned and object-oriented implementation of the algorithms appearing in Chapter 4. The SPARE Parts is designed to address the shortcomings of both of the toolkits described above. The following are the primary features of the library:

- The design of the SPARE Parts follows the structure of the taxonomy in Chapter 4 more closely. As a result, the code is easier to understand and debug. In addition, the SPARE Parts includes implementations of almost all of the algorithms described in Chapter 4.
- The use of C++ (instead of C) for the implementation has helped to avoid many of the memory management-related bugs that were present in the original toolkit.
- The client interface to the toolkit is particularly easy to understand and use. The flexibility introduced into the interface does not reduce the performance of the code in any significant way.
- The toolkit supports multi-threaded use of a single pattern matching object.

The toolkit is presented largely in a top-down fashion — from the client level classes to the foundation classes. The reader is assumed to have an overview of the taxonomy given in Chapter 4. Chapter 9 is structured as follows:

- Section 9.2 gives an introduction to the client interface of the toolkit. It includes some examples of programs which use the SPARE Parts.
- Section 9.3 describes the design decisions that lead to the client interface defined (with the use of abstract classes) in the toolkit.
- The design and implementation of concrete classes (implementing the client interface) is outlined in Section 9.4.
- Section 9.5 outlines the design and implementation of the foundation classes.
- Section 9.6 presents some experiences with the toolkit and the conclusions of this chapter.
- Some information on how to obtain and compile the toolkit is given in Section 9.7.

This chapter can also be read effectively with the source code of the toolkit.

9.2 Using the toolkit

In this section, we describe the client interface of the toolkit and present some examples of programs using the toolkit. The design issues that lead to the current client interface are not described here, but rather in Section 9.3.

The client interface defines two types of abstract pattern matchers: one for single keyword pattern matching, and one for multiple keyword pattern matching. (A future version of SPARE Parts can be expected to include classes for regular expression pattern matching — for example, an implementation of the algorithm described in Chapter 5.) All of the single keyword pattern matching classes have constructors which take a keyword.

Likewise, the multiple keyword pattern matchers have constructors which take a set of keywords. Both types of pattern matchers make use of *call-backs* (to be explained shortly) to register matched patterns. In order to match patterns using the call-back mechanism, the client takes the following steps (using single keyword pattern matching as an example):

1. A pattern matching object is constructed (using the pattern as the argument to the constructor).

2. The client calls the pattern matching member function *PMSingle::match*, passing the input string and a pointer *f* to a client defined function which takes an **int** and returns an **int**[1]. (This function is called the *call-back function*.)

3. As each match is discovered by the member function, the call-back function is called; the argument to the call is the index (into the input string) of the symbol immediately to the right of the match. (If there is no symbol immediately to the right, the length of the input string is used.)

4. If the client wishes to continue pattern matching, the call-back function returns the constant *TRUE*, otherwise *FALSE*.

5. When no more matches are found, or the call-back function returns *FALSE*, the member function *PMSingle::match* returns the index of the symbol immediately to the right of the last symbol processed.

We now consider an example of single keyword pattern matching.

Example 9.1 (Single keyword matching): The following program searches an input string for the keyword `hisher`, printing the locations of all matches along with the set of matched keywords:

```
#include "com-misc.hpp"
#include "pm-kmp.hpp"
#include <iostream.h>

static int report( int index ) {
    cout << index << '\n';
    return( TRUE );
}

int main( void ) {
    auto PMKMP Machine( "hisher" );
    Machine.match( "hishershey", &report );
```

[1]The integer return value is a Boolean value; recall that *TRUE* and *FALSE* have type **int** in C and C++. A recent draft of the C++ standard indicates that **bool** will be a new type (and a new keyword); the compilers used in the development of the SPARE Parts do not support this yet.

```
    return( 0 );
}
```

The header file `com-misc.hpp` provides a definition of constants *TRUE* and *FALSE*. Header file `pm-kmp.hpp` defines the Knuth-Morris-Pratt pattern matching class, while header file `iostream.h` defines the input and output streams, including the standard output *cout*. Function *report* is our call-back function, simply printing the index of the match (to the standard output), and returning *TRUE* to continue matching. The *main* function (the program mainline) creates a local KMP machine, with keyword `hisher`. The machine is then used to find all matches in string `hishershey`. (Recall that, in C and C++, a pointer to the beginning of the string is passed to member *match*, as opposed to the entire string.) □

In addition to the KMP algorithm defined in `pm-kmp.hpp`, other single keyword pattern matchers are defined in header file `bms.hpp`, which contains suggestions for instantiating some of the Boyer-Moore variants. Additionally, a brute-force single keyword pattern matcher is defined in `pm-bfsin.hpp`.

Multiple keyword pattern matching is performed in a similar manner, as the following example shows.

Example 9.2 (Multiple keyword matching): The following program searches an input string for the keywords `his`, `her`, and `she`, printing the locations of all matches:

```
#include "com-misc.hpp"
#include "string.hpp"
#include "set.hpp"
#include "acs.hpp"
#include <iostream.h>

static int report( int index, const Set<String>& M ) {
    cout << index << M << '\n';
    return( TRUE );
}

int main( void ) {
    auto Set<String> P( "his" );
    P.add( "her" ); P.add( "she" );
    auto PMACOpt Machine( P );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

Header file `string.hpp` defines a string class, while `set.hpp` defines a template class for sets of objects. Header file `acs.hpp` defines the Aho-Corasick pattern matching classes. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and the set of keywords matching, and returning *TRUE* to continue matching. Note that the call-back function has a different signature for multiple keyword pattern matching: it takes the index of the symbol to the right of the match, and the set of keywords matching with *index* as their right end-point.

The *main* function (the program mainline) creates a local AC machine from the keyword set. The machine is then used to find all matches in string `hishershey`. □

In the following two sections, we consider ways to use the SPARE Parts more efficiently in certain application domains.

9.2.1 Multi-threaded pattern matching

One important design feature (as a result of the call-back client interface) of the SPARE Parts is that it supports multi-threading. This can lead to high performance in applications hosted on multi-threading operating systems. For example, consider an implementation of a keyword `grep` application, in which 1000 files are to be searched for occurrences of a given keyword. The following are three potential solutions:

- In a sequential solution, a single pattern matching object is constructed and each of the 1000 files are scanned (in turn) for matches.
- In a naïve multi-threaded solution, 1000 threads are created (each corresponding to one of the input files). Each of the threads construct a pattern matching object, which is then used to search the file.
- An efficient solution is to create a single matching object, with 1000 threads sharing the single object. Each of the threads proceeds to search its file, using its own invocation of member function *PMSingle::match*.

The last (most efficient) solution would not have been possible without the call-back client interface. The technical reasons why this is possible are considered further in Section 9.3.

9.2.2 Alternative alphabets

The default structure in the SPARE Parts is to make use of the entire ASCII character set as the alphabet. This can be particularly inefficient and wasteful in cases where only a subset of these letters are used. For example, in genetic sequence searching, only the letters a, c, g, and t are used. The SPARE Parts facilitates the use of smaller alphabets through the use of *normalization*. Header file `alphabet.hpp` defines a constant *ALPHABETSIZE* (which, by default is *CHAR_MAX*). The alphabet which SPARE Parts uses is the range $[0, ALPHABETSIZE)$. An alternative alphabet can be used by redefining *ALPHABETSIZE*, and mapping the alternative alphabet in the required range. The mapping is performed by functions *alphabetNormalize* and *alphabetDenormalize*, both declared in `alphabet.hpp` (by default, these functions are the identity functions). The only requirement is that the functions map 0 to 0 (this is used to identify the end of strings).

Example 9.3 (Genetic sequence alphabet): In the genetic sequence example, we would make use of the following version of header `alphabet.hpp`:

```
#include <assert.h>
#define ALPHABETSIZE 5

inline char alphabetNormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 'a':   return( 1 );
    case 'c':   return( 2 );
    case 'g':   return( 3 );
    case 't':   return( 4 );
    default:    assert( !"Non-genetic character" );
    }
}

inline char alphabetDenormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 1:     return( 'a' );
    case 2:     return( 'c' );
    case 3:     return( 'g' );
    case 4:     return( 't' );
    default:    assert( !"Non-genetic image" );
    }
}
```

□

9.3 Abstract pattern matchers

In this section, we briefly consider the two abstract pattern matching classes which are used to define the client interfaces of single and multiple keyword pattern matchers.

User class 9.4 (*PMSingle*)

Files: `pm-singl`

Description: Class *PMSingle* defines the call-back client interface outlined in Example 9.1. The brute-force, Knuth-Morris-Pratt, and Boyer-Moore classes implement the defined interface.

Implementation: As an abstract class, there is no implementation.

□

User class 9.5 (*PMMultiple*)

Files: `pm-multi`

Description: This class defines the call-back client interface outlined in Example 9.2. The brute-force, Aho-Corasick, and Commentz-Walter classes implement the defined interface.

Implementation: As an abstract class, there is no implementation.

□

The use of call-backs in the client interface warrants some further explanation. As mentioned in Section 9.2, the use of call-backs allows multiple threads to make use of a single pattern matching object. Let us consider another possible (perhaps more obvious) client interface. In the alternative interface, member functions are provided to:

- Restart the matcher with a new input string
- Determine if there is a valid match
- Return the location of the current match
- Advance to the next match

The pattern matcher must contain state information such as: a pointer to the input string, the index of the current match, and a Boolean variable indicating if there is a valid match. Since all of this information is contained in a pattern matcher, multi-threaded use of a single object is not possible.

Although the call-back client interface must maintain the same state information, the information is stored in variables local to the *match* member function. Each thread making use of a pattern matcher has its own invocation of *match*, and therefore its own state information.

The call-back interface is not without its disadvantages. The most noticeable one is that call-backs require the client to write a function (in particular a free-standing function, as opposed to a member function of some other class) for use as the call-back function. This requirement may force the client to adopt a design approach that is not entirely object-oriented (due to the free-standing function). In practice, this disadvantage has proven to be relatively minor compared to the gains.

9.4 Concrete pattern matchers

In this section, we describe the classes which implement the interface defined by classes *PMSingle* and *PMMultiple*. The treatment of each of the families of classes includes any auxiliary (non-foundation) classes used. A summary of the classes and their template parameters (if any) is given in Section 9.4.6. We first consider the brute-force pattern matchers, followed by the KMP, AC, CW, and BM pattern matchers.

9.4.1 The brute-force pattern matchers

The brute-force pattern matchers are the most basic of the classes. While they are easy to understand, they are not intended for use in production quality software; they are used for benchmarking the other pattern matchers.

User class 9.6 (*PMBFSingle*, *PMBFMulti*)

Files: `pm-bfsin`, `pm-bfmul`

Description: The brute-force pattern matchers are naïve implementations of pattern matchers. They are only intended to form a baseline, against which the other (more efficient) classes can be measured. As a result, they are not intended for serious use.

Implementation: The implementations correspond (roughly) to Algorithm 4.10.

Performance: Instead of improving the performance of these classes, the client should make use of one of the other pattern matcher classes.

□

9.4.2 The KMP pattern matcher

User class 9.7 (*PMKMP*)

Files: `pm-kmp`

Description: This pattern matcher implements the Knuth-Morris-Pratt single keyword pattern matching algorithm. It inherits from *PMSingle* and implements the interface defined there.

Implementation: This class maintains the pattern keyword and a *FailIdx* representing the indexing failure function. The implementation of member function *match* corresponds to Algorithm 4.84.

□

Implementation class 9.8 (*FailIdx*)

Files: `failidx`

Description: Class *FailIdx* is the indexing failure function for use in *PMKMP*. The constructor takes a keyword. The only interesting member function is one to apply the failure function.

Implementation: The class contains an array (of size $|p| + 1$ for keyword p) of integers, representing the function. The constructor implements the classic KMP precomputation — see [Wat94a, Appendix A].

□

9.4.3 The AC pattern matchers

We now consider the Aho-Corasick family of pattern matchers. As the derivation in Section 4.3 shows, all of the Aho-Corasick variants share the same algorithm skeleton. The primary difference is in the mechanism used to compute the transition function on an input symbol. For speed, the AC algorithm skeleton is implemented via a template class as opposed to a base class (see Section 8.2.1). A number of variants (instantiations) of the Aho-Corasick objects are declared in the header `acs.hpp`, which is intended for client use.

Implementation class 9.9 (*PMAC*)

Files: `pm-ac`

Description: Template class *PMAC* implements the skeleton of the AC algorithms. The template argument must be one of the *ACMachine*... classes (called a *transition machine*) which is used to compute the next transition. The class inherits from *PMMultiple* and implements the corresponding interface. The header file is not intended to be used directly by clients; use `acs.hpp` instead.

Implementation: The implementation contains an *ACMachine*... object. The *PMAC* constructor passes the keyword set to the transition machine constructor. The implementation of member function *match* is taken from Algorithm 4.47.

Performance: The class is already highly tuned. The member functions of the transition machine (the template argument) should be inline for high performance.

□

9.4.3.1 AC transition machines and auxiliary classes

The transition machines are used in the AC skeleton (template class *PMAC*). The variety of transition machines corresponds to the different methods of computing the next transition discussed in Section 4.3.

Implementation class 9.10 (*ACMachineOpt*)

Files: `acmopt`

Description: This class provides an implementation of the optimized Aho-Corasick transition function, as described in Section 4.3.2. Member functions are provided to compute the next state (make a transition) and to compute the output (matched keywords) of a particular state (these member functions are the minimum interface required by template *PMAC*). It implements function γ_f and *Output* (see Definitions 4.49 and 4.44).

Implementation: The class contains a *Gamma* and an *ACOutput*. The member functions are pass-throughs to these classes.

□

Implementation class 9.11 (*ACMachineFail*)

Files: `acmfail`

Description: Class *ACMachineFail* implements the Aho-Corasick failure function method of computing the next transition, as described in Section 4.3.5.

Implementation: The class contains an *EFTrie* (an extended trie function τ_{ef}) and an *ACOutput*. The computation of the transition function is done with a linear search, as detailed in Section 4.3.5. The output member function is simply a pass-through to the *ACOutput*.

□

Implementation class 9.12 (*ACMachineKMPFail*)

Files: `acmkmpfl`

Description: Class *ACMachineKMPFail* is an implementation of the (multiple keyword) Knuth-Morris-Pratt method of computing the next transition, as described in Section 4.3.6.

Implementation: The class contains an *FTrie* and an *ACOutput*. The transition function is computed by linear search — see Section 4.3.6.

Performance: Most inefficiencies are due to the difference between this class and *ACMachineFail*, as mentioned in the comment after Algorithm 4.76 on page 75.

□

Implementation class 9.13 (*Gamma*)

Files: `acgamma`

Description: Class *Gamma* implements the 'optimized' Aho-Corasick transition function γ_f. The constructor takes an *FTrie* and an *FFail*. The main member function computes the image of the function, given a *State* and a character.

Implementation: Class *Gamma* is implemented via a *StateTo*< *SymbolTo*<*State*> >. The constructor performs a breadth-first traversal of the trie, using the failure function to compute function γ_f.

Performance: There are methods of computing function γ_f directly from the keyword set without the *FTrie*. Computing the trie and the failure function independently is more costly (in both space and time), but provides a modular separation of the functions and keeps the constructor for *Gamma* manageable.

□

Implementation class 9.14 (*EFTrie*)

Files: `aceftrie`

Description: Class *EFTrie* implements the extended forward trie — function τ_{ef} (see Definition 4.68). The constructor takes an *FTrie*.

Implementation: This class is implemented with a *StateTo*< *SymbolTo*<*State*> >. The constructor simply copies the *FTrie* and extends it.

Performance: The performance could be significantly increased by not copying the trie from scratch. Unfortunately, this is not possible because a single *FTrie* is used to construct the *EFTrie* and the *ACOutput* objects in *ACMachineFail* so the *State*s of the *EFTrie* and the *ACOutput* objects correspond.

□

Implementation class 9.15 (*ACOutput*)

Files: `acout`

Description: *ACOutput* implements the Aho-Corasick output function *Output* (see Definition 4.44). The constructor takes the set of keywords, the corresponding forward trie (*FTrie*), and the corresponding forward failure function (*FFail*).

Implementation: This class contains a *StateTo*< *Set*<*String*> >. The constructor performs a breadth-first traversal of the trie, using the failure function to compute function *Output* — implementing the algorithm given in [WZ92].

Performance: The high performance of this class depends quite heavily upon the use-counting of class *String* (see User class 9.34).

□

9.4.4 The CW pattern matchers

As outlined in Section 4.4, all of the Commentz-Walter variants share a common algorithm skeleton. The difference lies in how the safe shift distance is computed. For this reason, the skeleton is defined as a template class as follows. The variants of the Commentz-Walter algorithm are defined (via **typedef**) in header `cws.hpp`.

User class 9.16 (*PMCW*)

Files: `pm-cw`

Description: Class *PMCW* implements the Commentz-Walter skeleton. It inherits from *PMMultiple* and implements the public interface defined there. The template argument must be one of the *CWShift...* classes. The argument provides the safe shift distance during the scanning of the string.

Implementation: A *PMCW* contains a shifter object, an *RTrie* (a reverse trie for scanning the string), and a *CWOutput* (output function for detecting a match). The constructor passes the set of keywords through to the sub-objects. The implementation of member function *match* is taken directly from Algorithm 4.93.

Performance: Improvements in performance can most easily be gained by improving the shifter objects or the implementation of the reverse trie and output function. The member functions of the safe shift objects should be inline, since they are simple and they are called repeatedly in the inner repetition of *match*.

□

9.4.4.1 Safe shifters and auxiliary functions

The safe shifter classes form the basis of the Commentz-Walter algorithms. The derivation of the safe shifts is covered in Section 4.4. The choice of which one to use in a given application is dominated by a tradeoff between precomputation time and greater shift distances. For applications in which the input string is relatively short, class *CWShiftNLA* has the fastest precomputation but provides the smallest shift distances. (Actually, *CWShiftNaive* provides a shift distance of 1; it is intended only for use in benchmarking the algorithms.) For an application in which the time to scan the string outweighs the time for precomputation, *CWShiftRLA* is the best choice.

Implementation class 9.17 (*CWShiftNaive*)

Files: `cwshnaiv`

Description: *CWShiftNaive* implements a naïve safe shift distance of 1 in the Commentz-Walter algorithm. This class is intended for benchmarking use, as opposed to serious applications.

Implementation: The implementation is trivial since no data is stored. The shift distance member function simply returns 1.

□

Implementation class 9.18 (*CWShiftNLA*)

Files: `cwshnla`

Description: This class implements the 'no-lookahead' shift distance of Definition 4.101.

Implementation: The class contains a single shift function. In the constructor, local *D1* and *D2* shift functions are constructed. These shift functions are combined into a single shift. In the description in 4.4 the combining of the two would be done as the input string is scanned. They are combined at precomputation time for performance and space reasons.

Performance: The performance would be difficult to improve as the *D1* and *D2* shift functions are already combined at precomputation time.

□

Implementation class 9.19 (*CWShiftWBM*)

Files: `cwshwbm`

Description: Class *CWShiftWBM* implements the 'weak Boyer-Moore' shift distance — see Definition 4.136.

Implementation: The implementation is through *D1*, *D2*, and *CharBM*. The amount of shift distance contributed by each of the three functions are combined as the input string is scanned.

Performance: Since the shift distance is combined at string-scanning time, it is important that auxiliary functions such as *min* and *max* are inline functions.

□

Implementation class 9.20 (*CWShiftNorm*)

Files: `cwshnorm`

Description: This class implements the 'normal' Commentz-Walter shift function — see Definition 4.125.

Implementation: The implementation is similar to that of *CWShiftWBM*, with the exception that the *CharBM* is replaced by a *CharCW*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class 9.21 (*CWShiftOpt*)

Files: `cwshopt`

Description: Class *CWShiftOpt* is an implementation of the 'optimized' Commentz-Walter shift distance — see Definition 4.107.

Implementation: The implementation is similar to that of *CWShiftWBM*, except that the *D1* and *CharBM* are replaced by a *DOpt*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class 9.22 (*CWShiftRLA*)

Files: `cwshrla`

Description: This class implements the 'right lookahead' shift function as defined in Definition 4.140.

Implementation: The implementation is similar to *CWShiftOpt*, with the addition of a *CharRLA*.

Performance: See the performance clause for *CWShiftWBM*.

□

Implementation class 9.23 (*CharCW*, *CharBM*, *CharRLA*)

Files: `cwchar`, `cwcharbm`, `cwcharrl`

Description: These three classes are the shift functions which are based upon a single character (the first mismatching character). The definitions of the functions can be found (respectively) in Definitions 4.122, 4.128, and 4.138.

Implementation: Classes *CharBM* and *CharRLA* are implemented through a single array, while *CharCW* is implemented through two nested arrays. Their constructors all make breadth-first traversals of the reverse trie. The shift member functions are trivial lookups.

□

Implementation class 9.24 (*D1*, *D2*, *DOpt*)

Files: `cwd1`, `cwd2`, `cwdopt`

Description: These three classes implement the Commentz-Walter shift component functions d_1, d_2, and d_{opt} (see Definitions 4.97 and 4.105). Classes *D1* and *D2* map a *State* to an integer, while *DOpt* maps a *State* and a character to an integer.

Implementation: The implementations of *D1* and *D2* are in terms of *StateTo*< **int** >, with *DOpt* as *StateTo*< *SymbolTo*< **int** > >. The constructors take an *RTrie* which they traverse, implementing Algorithms 5.27 and 5.28 (for *D1* and *D2*) and the d_{opt} precomputation algorithm given in [WZ95].

Performance: The derivations of the precomputation algorithms in [WZ95] indicate that the currently implemented algorithms are not likely to be improved.

□

Implementation class 9.25 (*CWOutput*)

Files: `cwout`

Description: *CWOutput* implements the Commentz-Walter output function (which determines if a match has been found). The class maps a *State* (from the reverse trie) to a string if the *State* corresponds to a keyword. There is a member function which reports if a given *State* corresponds to a keyword.

Implementation: The implementation uses a *StateTo<String*>*. If the entry is 0, then the corresponding *State* does not correspond to a keyword. If the entry is not 0, the entry points to the corresponding keyword. The constructor takes an *EFTrie* and the set of keywords.

□

9.4.5 The BM pattern matchers

Like the Aho-Corasick and the Commentz-Walter algorithms, all variants of the Boyer-Moore algorithm share a common skeleton. Again, the skeleton is implemented as a template class, with the template parameters being used to instantiate the different possible variants. Examples of how to instantiate some of the variants are given in header file `bms.hpp`. The structure of these variants of the BM algorithms are derived fully in Section 4.5.

User class 9.26 (*PMBM*)

Files: `pm-bm`

Description: This template function implements the Boyer-Moore variants derived in Section 4.5. It inherits from *PMSingle* and implements the public interface defined there. The class takes three template parameters:

- A 'match order' which is used to compare the keyword to the input string. It must be one of the *STrav...* classes.
- A 'skip loop' which is used to skip portions of the input text that cannot possibly contain a match. The argument must be one of the *SL...* classes.
- A 'match information' shift distance class which is used to make larger shifts through the input string, after a match attempt. The argument must be one of the *BMShift...* classes.

For more on each of these three components, see Section 4.5.

Implementation: The implementation contains a copy of the keyword and an object of each of the template arguments. The implementation is taken directly from Algorithm 4.177.

Performance: A fast implementation relies on the skip loop, match order, and shifter member functions all being inline.

□

The Boyer-Moore class *PMBM* takes three template arguments, making it one of the more complex template classes. We now consider an example of an instantiation of the class.

Example 9.27 (Instantiating template class *PMBM*): The first of the template arguments is the match order. For this, we select the 'reverse' match order class *STravREV*. As our 'skip loops', we select class *SLFast1*. For our shifter, we select *BMShift11* shifter class. We can now declare a pattern matching object for string `hehshe` as follows:

```
#include "bms.hpp"

static PMBM< STravREV, SLFast1, BMShift11<STravREV> > M( "hehshe" );
```

Note that the same string traverser class must appear as the first template argument to *PMBM* and as the template argument to the shifter class. □

9.4.5.1 Safe shifters and auxiliary functions

The match orders, the skip loops, and the shifters form the core of the implementation of the BM algorithm variants. The match order classes will be described in Implementation classes 9.39. The skip loop classes (class names starting with *SL*...) and the shifters (class names starting with *BMShift*...) are described below, along with some of the shift components.

Implementation class 9.28 (*SLNone*, *SLSFC*, *SLFast1*, *SLFast2*)

Files: `bmslnone`, `bmslsfc`, `bmslfst1`, `bmslfst2`

Description: The skip loop classes are used to skip portions of the input string in which no matches are possible. *SLNone* makes no shift through the input string (it is only included for completeness, since it is derived in Section 4.5.1 and in [HS91]).

Implementation: All of the implementations follow directly from Section 4.5.1. The constructors simply take the pattern keyword. Some of the skip loops store lookup tables to compute the shift distance. In the case of *SLSFC*, the shift distance is 1.

Performance: The current implementations are for maximum performance in time. Space is sacrificed in favour of speed. All of the member functions should be inlined.

□

Implementation class 9.29 (*BMShiftNaive*)

Files: `bmshnaiv`

Description: The naïve shift distance class, *BMShiftNaive*, provides a safe shift distance of 1. It is intended only for benchmarking purposes.

□

Implementation class 9.30 (*BMShift11*, *BMShift12*)

Files: `bmsh1-1`, `bmsh1-2`

Description: These two classes implement two of the possible shift distances considered in Section 4.5.2 on page 110. Both classes are template classes, which expect a string traverser (*STrav...*) class as their template argument. The particular traverser used in the instantiation must be the same traverser class used as the match order in *PMBM*.

Implementation: Class *BMShift11* makes use of an *S1* and a *Char1*, while *BMShift12* makes use of an *S1* and a *Char2*. In both cases, the two shift components are combined during the scanning of the string. The template argument is used to instantiate the correct versions of the component shift functions.

Performance: Since the shift components are combined during the scanning of the input string, the performance could be improved by combining them in the constructor.

□

Implementation class 9.31 (*Char1*, *Char2*)

Files: `bmchar1`, `bmchar2`

Description: These two classes are shift components, implementing functions $char_1$ and $char_2$ given in Definition 4.174. Since these two functions depend upon the particular match order in use, these two classes are defined as template classes. The template argument must be one of the *STrav...* classes.

Implementation: The definitions of functions $char_1$ and $char_2$ contain **MIN** quantifications. As a result, the constructors perform a linear search to compute the functions. The linear search is general, since it makes use of the template argument (the string traverser).

Performance: When *STravFWD* or *STravREV* are used as the template argument, the linear search can be performed more efficiently (as in the classical Boyer-Moore precomputation). Such traverser specific precomputation can be written as template instantiation overriding functions. (This has not yet been done — it should be done by clients who require higher performance from the SPARE Parts.)

□

Implementation class 9.32 (*S1*)

Files: `bms1`

Description: This class implements the s_1 shift component given in Definition 4.174. As with shift components $char_1$ and $char_2$, this class depends upon the particular string traverser in use. *S1* is a template class which expects a string traverser as its template argument.

Implementation: See the implementation of classes *Char1* and *Char2*.

Performance: See *Char1* and *Char2*.

□

9.4.6 Summary of user classes

In this section, we present two tables which summarize the various user classes for pattern matching. The first table summarizes the descendents of abstract single keyword pattern matching class *PMSingle*. All three of the concrete classes are described along with their possible template arguments (if any):

Class	*Description*	
PMBFSingle	Brute-force pattern matcher	
PMKMP	Knuth-Morris-Pratt pattern matcher	
PMBM	Boyer-Moore pattern matcher template Three template arguments required, as follows:	
	Match orders	
	STravFWD	Forward (left to right)
	STravREV	Reverse (right to left)
	STravOM	Optimal mismatch (increasing frequency)
	STravRAN	Random
	Skip loops	
	SLNone	No skip
	SLSFC	Leftmost keyword symbol compared (always 1 symbol shift)
	SLFast1	Rightmost symbol compared
	SLFast2	Rightmost symbol compared (greater shift)
	Shifters	
	BMShiftNaive	Shift of one symbol
	BMShift11	Shift without mismatching symbol information
	BMShift12	Shift with mismatching symbol information

The following table summarizes the concrete class descendents of the abstract multiple keyword pattern matcher class *PMMultiple*. Two of them are template classes and their possible template arguments are summarized as well:

Class	*Description*	
PMBFMulti	Brute-force pattern matcher	
PMAC	Aho-Corasick pattern matcher template Single template argument required:	
	Transition machines	
	ACMachineOpt	Optimal transition function
	ACMachineFail	Failure function with extended trie
	ACMachineKMPFail	Knuth-Morris-Pratt failure function with forward trie
PMCW	Commentz-Walter pattern matcher template Single template argument required:	
	Shifters	
	CWShiftNaive	Shift of one symbol
	CWShiftNLA	No lookahead symbol used
	CWShiftWBM	Weak Boyer-Moore
	CWShiftNorm	Normal Commentz-Walter
	CWShiftOpt	Optimized Commentz-Walter
	CWShiftRLA	Lookahead right one symbol

9.5 Foundation classes

In this section, we consider the design and implementation of the foundation classes and functions. These classes and functions are not of primary concern to the client, but are used to construct classes which form the client interface. Some of these classes are reused in the FIRE Lite — a toolkit of finite automata algorithms described in Chapter 10.

A number of these classes will be replaceable by standard library classes once the draft C++ standard becomes stable and implementations of the draft standard start to appear.

9.5.1 Miscellaneous

A number of header files and their corresponding definitions do not fall into a particular category. Header `com-misc.hpp` contains definitions of constants *TRUE* and *FALSE* and integer maximum and minimum functions *max* and *min*.

Implementation class 9.33 (*State*)

Files: `state`

Description: Tries and finite automata require the definition of states. This header contains a definition of states and some constants, in particular an *INVALIDSTATE* and a *FIRSTSTATE*. The *FIRSTSTATE* is used (by convention) as the start state in tries and finite automata.

Implementation: *State* is not defined as a class. Instead, it is **typedef**'d to be an integer for efficiency reasons.

□

User class 9.34 (*String*)

Files: `string`

Description: The raw string conventions in C and C++ are too rudimentary to be used effectively and safely. This class provides a higher level (and safer) mechanism for using strings. The interfaces provides members for indexing the individual characters in the string, copying strings, assignment, length of the string, and an output operator (stream insertion).

Implementation: The class is implemented through use-counting with a private class. This makes assignment and copying of strings particularly efficient, but it adds an additional level of indirection to many of the operations. The cost of the extra indirection was found to be negligible compared to the cost of creating complete copies of strings.

Performance: The length of the string is kept in the private class. It requires a complete traversal of the string, using standard function *strlen*. Since this can be particularly inefficient for very large strings, it could be replaced with more efficient methods of determining the length when the string is constructed from a file. The inefficiency of the extra indirection will very likely be removed (when examining individual characters of the string) by a good optimizing compiler.

□

9.5.2 Arrays, sets, and maps

In this section, we describe the basic template classes used to construct more complex objects.

Implementation class 9.35 (*Array*)

Files: `array`

Description: As with strings, the raw C and C++ facilities for arrays are not safe and flexible enough for our purposes. An *Array* template class constructs arrays of objects of class T. The operators available in raw arrays are provided. Notable additions are: bounds-checked indexing into the array, a stream insertion operator (assuming that class T has an insertion operator), and the ability to resize the array dynamically.

Implementation: An *Array* is implemented by an array of objects, an apparent size (to the client), and real size (for dynamic resizing). The class constant *growthSize* is used during resize operations to allocate some extra elements; these extra elements can be used later to avoid another call to the memory allocator.

Performance: The copy constructor and the assignment operator both make copies of the underlying array. This costly operation could be avoided through the use of use-counting. The class constant *growthSize* is a tuning constant which is 5 by default. Other values may provide higher performance in certain circumstances.

□

Implementation class 9.36 (*Set*)

Files: `set`

Description: Sets of objects are used in a variety of places. The template class *Set* implements a set of objects of class *T*. Common set operations (such as element add, remove, union, membership tests, size) are available, as well as a rudimentary iterator facility, and an insertion operator. The class is replaceable by a standard one, once the draft C++ standard is stable. The current draft of the standard proposes to use the Standard Template Library by Stepanov and Lee [SL94]. The Standard Template Library definition puts forth a more complex set of iterators than needed in the SPARE Parts.

Implementation: The implementation is via an *Array*. This makes management of the size of the set particularly simple. Most of the member functions of *Set* are simple pass-throughs to the corresponding *Array* members.

Performance: The performance is most easily improved through modifications in class *Array*.

□

Implementation class 9.37 (*StateTo*)

Files: `stateto`

Description: Template class *StateTo* implements a function mapping a *State* to an object of class *T*. Member functions are provided for setting up the function, applying the function, and for adjusting the range of states in the domain.

Implementation: *StateTo* is implemented in terms of an *Array*. The member functions are mostly pass-throughs to the *Array* members.

Performance: Since the implementation is through class *Array*, the performance of *StateTo* is most easily improved by improving *Array*.

□

Implementation class 9.38 (*SymbolTo*)

Files: `symbolto`

Description: Template class *SymbolTo* implements a function from characters to an object of class T. The characters in the domain are assumed to be in the range $[0, \mathit{ALPHABETSIZE})$. Member functions are provided for setting up the function and for applying it.

Implementation: Class *SymbolTo* is implemented in terms of an *Array*. The member functions are mostly pass-throughs to the *Array* members.

Performance: The performance is most easily improved through improvements to class *Array*.

□

9.5.3 Tries and failure functions

Tries and failure functions form the basis for the multiple keyword pattern matching algorithms. Tries and match orders (for use in the Boyer-Moore algorithms) are both implemented in terms of string traversers. String traversers, tries, and failure functions are described in this section.

Implementation class 9.39 (*STravFWD*, *STravREV*, *STravOM*, *STravRAN*)

Files: `stravfwd`, `stravrev`, `stravom`, `stravran`

Description: *String traversers* are synonymous with match orders (from Chapter 4). For a given string of length n, a string traverser is a bijection on $[0, n)$. It can be used to traverse the characters of the string in a particular order. Class *STravFWD* corresponds to the identity function, which traverses the string in the left to right direction. Class *STravREV* allows one to traverse the string in the right to left direction. Classes *STravOM* and *STravRAN* correspond (respectively) to an optimal mismatch order (see Algorithm detail 4.151) and a random order. The latter two classes are used primarily in the Boyer-Moore algorithms, while the first two find use in class *Trie*. All of the traversers have constructors which take a keyword.

Implementation: Since class *STravFWD* implements the identity function, it does not contain any private data. Class *STravREV* only contains the length of the keyword in order to implement the bijection. The other traversers have not yet been implemented.

Performance: Since these classes are used to consider the characters in strings, it is important that they are inlined.

□

Implementation class 9.40 (*Trie*)

Files: `trie`, `tries`

Description: Template class *Trie* implements tries [Fred60]. The template parameter must be one of the string traverser classes. *Trie* has a constructor which takes a set of strings, yielding the corresponding trie. The string traverser determines in which order the strings are traversed in the constructor. Using *STravFWD* as the template parameter gives a forward trie (see Definition 4.26), while using *STravREV* gives a reverse trie (see Definition 4.13). Forward and reverse tries are **typedef**'d in header `tries.hpp`; this header is intended for use by clients.

There are a special set of member functions to perform breadth-first traversals of the trie, and to determine the depth (when the trie is considered as a tree) of a particular *State* (since each *State* corresponds to a string, the depth of the *State* is equal to the length of the string).

Implementation: *Trie* is implemented using a *StateTo*< *SymbolTo*<*State*> >. An additional *StateTo* maps *State*s to integers, keeping track of the depth of the *State*s for the breadth-first traversals. Almost all of the member functions are simple pass-throughs. The constructor uses the depth-first method of constructing the trie from the set of strings.

Performance: The use of nested mappings (*StateTo*s and *SymbolTo*s) is highly efficient in time, but it is known that tries can be implemented much more space efficiently, as in [AMS92]. The disadvantage to such an implementation is that each transition requires more time.

□

Implementation class 9.41 (*Fail*)

Files: `fail`, `fails`

Description: Failure functions are implemented using template class *Fail*. *Fail* has a constructor which takes a *Trie* as parameter. The type of trie (forward or reverse) determines the type of failure function constructed (forward or reverse). Consequently, the template argument to *Fail* must be either *STravFWD* or *STravREV*. The template argument is only used to determine the type of the *Trie* taken as parameter by the constructor. Header `fails.hpp` contains **typedef**s of the forward and reverse tries; this header file is intended for use by clients.

Implementation: A failure function is implemented as a *StateTo*<*State*>. The constructor uses a standard breadth-first traversal of the *Trie* (see [WZ92] or Chapter 5 for algorithms constructing failure functions from tries).

Performance: The performance can only be improved through improvements to class *StateTo*.

□

9.6 Experiences and conclusions

Designing and coding the SPARE Parts lead to a number of interesting experiences in class library design. In particular:

- The SPARE Parts comprises 5787 lines of code in 59 `.hpp`, 32 `.cpp`, 43 `.ppp`, and 49 `.ipp` files.
- Compiling the files, with the WATCOM C++32 Version 9.5b compiler, shows that the size of the object code varies very little for the various types of pattern matchers.
- The taxonomy presented in Chapter 4 was critical to correctly implementing the many complex precomputation algorithms.
- Designing and structuring generic software (reusable software such as class libraries) is much more difficult than designing software for a single application. The general structure of the taxonomy proved to be helpful in guiding the structure of the SPARE Parts.
- One of the debugging session lead to the discovery of a bug in the code for precomputation of failure functions. Further inspection showed that the C++ code was correctly implemented from the abstract algorithm presented in [WZ92, Part II]. Unfortunately, part of the abstract algorithm used a depth-first traversal of a trie, while the postcondition called for a breadth-first traversal.
- In Chapter 13, we consider the relative performance of the algorithms implemented in the SPARE Parts. It is also helpful to consider how the implementations in the SPARE Parts fare against commercially available tools such as the `fgrep` program. Four `fgrep`-type programs were implemented (using the SPARE Parts), corresponding to the Knuth-Morris-Pratt, Aho-Corasick, Boyer-Moore, and Commentz-Walter algorithms. The four tools were benchmarked informally against the `fgrep` implementation which is sold as part of the MKS toolkit for MS-DOS. The resulting times (to process a 984149 byte text file, searching for a single keyword) are:

`fgrep` *variant*	MKS	KMP	BM	AC	CW
Time (sec)	3.9	5.1	4.2	4.7	4.0

  These results indicate that using a general toolkit such as the SPARE Parts will result in performance which is similar to carefully tuned C code (such as MKS `fgrep`).

Detailed records were kept on the time required for designing, typing, compiling (and fixing syntax errors), and debugging the toolkit. The time required to implement the toolkit is broken down as follows (an explanation of each of the tasks is given below):

Task	Design	Typing	Compile/Syntax	Debug	Total
Time (hrs:min)	6:00	13:40	10:05	5:15	35:00

Most of these times are quite short compared to what a software engineer could expect to spend on a project of comparable size. The following paragraphs explain exactly what each of the tasks entailed:

- The *design* phase involved the creation of the inheritance hierarchy and the declaration (on paper) of all of the classes in the toolkit. (A C++ declaration provides names and signatures of functions, types, and variables, whereas a definition provides the implementation of these items.) The design phase proceeded exceptionally smoothly, thanks to a number of things:
  - The inheritance hierarchy followed directly from the structure of the taxonomy.
  - The decisions on the use of templates (instead of virtual functions — see Chapter 8) and call-backs were made on the basis of experience gained with the FIRE Engine. These decisions were also somewhat forced by the efficiency requirements on the toolkit, as well as the need for multi-threading.
  - Representation issues, such as the selection of data structures, were resolved using experience gained with the earlier Eindhoven Pattern Kit.
- Once the foundation classes were declared and defined, typing the code amounted to a simple translation of guarded commands to C++.
- The times required for compilation and syntax checking were minimized by using a very fast integrated environment (BORLAND C++) for initial development. Only the final few compilations were done using the (slower, but more thoroughly optimizing) WATCOM C++ compiler. The advantages of using a fast development environment on a single user personal computer should not be underestimated.
- Since the C++ code in the toolkit was implemented directly from the abstract algorithms (for which correctness arguments are given), the only (detected) bugs were those involving typing errors (such as the use of the wrong variable, etc.). Correspondingly, little time needed to be spent on debugging the toolkit.

9.7 Obtaining and compiling the toolkit

The SPARE Parts is available for anonymous ftp from `ftp.win.tue.nl` in directory:

`/pub/techreports/pi/watson.phd/spare/`

The toolkit, and some associated documentation, are combined into a `tar` file. A number of different versions of this file are stored — each having been compressed with a different compression utility.

The SPARE Parts has been successfully compiled with BORLAND C++ Versions 3.1 and 4.0, and WATCOM C++32 Version 9.5b on MS-DOS and MICROSOFT WINDOWS 3.1 platforms. Since the WATCOM compiler is also a cross-compiler, there is every reason to believe that the code will compile for WINDOWS NT or for IBM OS/2. The implementation of the toolkit makes use of only the most basic features of C++, and it should be compilable using any of the template-supporting UNIX based C++ compilers.

A version of the SPARE Parts will remain freely available (though not in the public domain). Contributions to the toolkit, in the form of new algorithms or alternative implementations, are welcome.

Chapter 10

FIRE Lite: *FA*s and *RE*s in C++

This chapter describes a C++ finite automata toolkit known as FIRE Lite (FInite automata and Regular Expressions; Lite since it is the smaller and newer cousin of the FIRE Engine toolkit, also from the Eindhoven University of Technology). The client interface and aspects of the design and implementation are also described.

10.1 Introduction and related work

FIRE Lite is a C++ toolkit implementing finite automata and regular expression algorithms. The toolkit is a computing engine, providing classes and algorithms of a low enough level that they can be used in most applications requiring finite automata or regular expressions. Almost all of the algorithms derived in Chapter 6 are implemented. This chapter serves as an introduction to the client interface of the toolkit and the design and implementation issues of the toolkit.

10.1.1 Related toolkits

There are several existing finite automata toolkits. They are:

- The Amore system, as described in [JPTW90]. The Amore package is an implementation of the semigroup approach to formal languages. It provides procedures for the manipulation of regular expressions, finite automata, and finite semigroups. The system supports a graphical user-interface on a variety of platforms, allowing the user to interactively and graphically manipulate the finite automata. The program is written (portably) in the C programming language, but it does not provide a programmer's interface. The system is intended to serve two purposes: to support research into language theory and to help explore the efficient implementation of algorithms solving language theoretic problems.

- The Automate system, as described in [CH91]. Automate is a package for the symbolic computation on finite automata, extended regular expressions (those with the intersection and complementation operators), and finite semigroups. The system provides a textual user-interface through which regular expressions and finite automata can be manipulated. A single finite automata construction algorithm (a variant of Thompson's) and a single deterministic finite automata minimization algorithm is provided (Hopcroft's). The system is intended for use in teaching and language theory research. The (monolithic) program is written (portably) in the C programming language, but provides no function library interface for programmers.

According to Pascal Caron (at the Université de Rouen, France), a new version of Automate is being written in the MAPLE symbolic computation system.

- The FIRE Engine, as described in [Wat94b, Wat94c]. The FIRE Engine was the first of the toolkits from the Computing Science Faculty in Eindhoven. It is an implementation of all of the algorithms appearing in two early taxonomies of finite automata algorithms which appeared in [Wat93a, Wat93b]. The toolkit is somewhat larger than FIRE Lite (the FIRE Engine is 9000 lines of C++) and has a slightly larger and more complex public interface. The more complex interface means that the toolkit does not support multi-threaded use of a single finite automaton.

- The Grail system, as described in [RW93]. Grail follows in the tradition of such toolkits as Regpack [Leis77] and INR [John86], which were all developed at the University of Waterloo, Canada. It provides two interfaces:

  - A set of 'filter' programs (in the tradition of UNIX). Each filter implements an elementary operation on finite automata or regular expressions. Such operations include conversions from regular expressions to finite automata, minimization of finite automata, etc. The filters can be combined as a UNIX 'pipe' to create more complex operations; the use of pipes allows the user to examine the intermediate results of complex operations. This interface satisfies the first two (of three) aims of Grail [RW93]: to provide a vehicle for research into language theoretic algorithms, and to facilitate teaching of language theory.
  - A raw C++ class library provides a wide variety of language theoretic objects and algorithms for manipulating them. The class library is used directly in the implementation of the filter programs. This interface is intended to satisfy the third aim of Grail: an efficient system for use in application software.

The provision of the C++ class interface in Grail makes it the only toolkit with aims similar to those of the FIRE Engine and of FIRE Lite. In the following section, we will highlight some of the advantages of FIRE Lite over the other toolkits.

10.1.2 Advantages and characteristics of FIRE Lite

The advantages to using FIRE Lite, and the similarities and differences between FIRE Lite and the existing toolkits are:

- FIRE Lite does not provide a user interface[1]. Some of the other toolkits provide user interfaces for the symbolic manipulation of finite automata and regular expressions. Since FIRE Lite is strictly a computing engine, it can be used as the implementation beneath a symbolic computation application.

- The toolkit is implemented for efficiency. Unlike the other toolkits, which are implemented with educational aims, it is intended that the implementations in FIRE Lite are efficient enough that they can be used in production quality software.

- Despite the emphasis on efficiency in FIRE Lite the toolkit still has educational value. The toolkit bridges the gap between the easily understood abstract algorithms appearing in Chapter 6 and practical implementation of such algorithms. The C++ implementations of the algorithms display a close resemblance to their abstract counterparts.

- Most of the toolkits implement only one of the known algorithms for constructing finite automata. For example, Automate implements only one of the known constructions. By contrast, FIRE Lite provides implementations of almost all of the known algorithms for constructing finite automata. Implementing many of the known algorithms has several advantages:
  - The client can choose between a variety of algorithms, given tradeoffs for finite automata construction time and input string processing time.
  - The efficiency of the algorithms (on a given application) can be compared.
  - The algorithms can be studied and compared by those interested in the inner workings of the algorithms.

10.1.3 Future directions for the toolkit

A number of improvements to FIRE Lite will appear in future versions:

- Presently, FIRE Lite implements only acceptors. Transducers (as would be required for some types of pattern matching, lexical analysis, and communicating finite automata) will be implemented in a future version.

- A future version of the toolkit will include support for extended regular expressions, i.e. regular expressions containing intersection or complementation operators.

- Basic regular expressions and automata transition labels are represented by character ranges. A future version of FIRE Lite will permit basic regular expressions and transition labels to be built from more complex data-structures. For example, it will be possible to process a string (vector) of structures. (Version 2.0 of Grail included a similar improvement.)

[1] A rudimentary user interface is included for demonstration purposes.

10.1.4 Reading this chapter

The toolkit is presented largely in a top-down fashion. The chapter is structured as follows:

- Section 10.2 gives an introduction to the client interface of the toolkit. It includes some examples of programs which use FIRE Lite.
- Section 10.3 gives an overview of the structure of the toolkit.
- Section 10.4 outlines the client interfaces to regular expressions and finite automata.
- Section 10.5 presents the concrete classes which implement the interface defined in the abstract classes.
- Section 10.6 outlines the design and implementation of the foundation classes. Some of the foundation classes in the SPARE Parts have been reused in FIRE Lite. Those classes have not been described here again — they can be found in Chapter 9.
- Section 10.7 presents some experiences with using the toolkit, and the conclusions of this chapter.
- Some information on how to obtain and compile the toolkit is given in Section 10.8.

10.2 Using the toolkit

In this section, we describe the client interface to the toolkit — including some examples which use the toolkit. The issues in the design of the current client interface are described in Section 10.4.

There are two components to the client interface of FIRE Lite: regular expressions (class *RE*) and finite automata (classes whose names begin with *FA*...). We first consider regular expressions and their construction.

Regular expressions are implemented through class *RE*. This class provides a variety of constructors and member functions for constructing complex regular expressions. Stream insertion and extraction operators are also provided. (The public interface of *RE* is rather fat, consisting of a number of member functions intended for use by the constructors of finite automata.) The following example constructs a regular expression.

Example 10.1 (Regular expression): The following program constructs a simple regular expression and prints it:

```
#include "re.hpp"
#include <iostream.h>

int main(void) {
    auto RE e( 'B' );
    auto RE f( CharRange( 'a', 'z' ) );
```

```
    e.concatenate( f );
    e.or( f );
    e.star();
    cout << e;
    return( 0 );
}
```

The header `re.hpp` defines regular expression class *RE*. The program first constructs two regular expressions. The first is the single symbol B. The second regular expression is a *CharRange* — a character range. The *RE* constructed corresponds to the range $[a, z]$ of characters. No particular ordering is assumed on the characters, though most platforms use the ASCII ordering. Character ranges can always be used as atomic regular expressions. The program concatenates regular expression f onto e and then unions f onto e. Finally, the Kleene closure operator is applied to regular expression e. The final regular expression, which is $((B \cdot [a, z]) \cup [a, z])^*$, is output (in a prefix notation) to standard output. □

The abstract finite automata class defines the common interface for all finite automata; it is defined in `faabs.hpp`. A variety of concrete finite automata are provided in FIRE Lite; they are declared in header `fas.hpp`. There are two ways to use a finite automaton. In both of them the client constructs a finite automaton, using a regular expression as argument to the constructor. The two are outlined as follows:

1. In the simplest of the two, the client program calls finite automaton member function *FAAbs::attemptAccept*, passing it a string and a reference to an integer. The member function returns *TRUE* if the string was accepted by the automaton, *FALSE* otherwise. Into the integer reference it places the index (into the string) of the symbol to the right of the last symbol processed.

2. In the more complex method, the client takes the following steps (which resemble the steps required in using a pattern matcher mentioned in Chapter 9):

   (a) The client calls the finite automaton member function *FAAbs::reportAll*, passing it a string and a pointer to a function which takes an integer and returns an integer. As in the SPARE Parts (Chapter 9), the function is the call-back function.

   (b) The member function processes the input string. Each time the finite automaton enters a final state (while processing the string), the call-back function is called. The argument to the call is the index (into the input string) of the symbol immediately to the right of the symbol which took the automaton to the final state.

   (c) To continue processing the string, the call-back function returns *TRUE*, otherwise *FALSE*.

   (d) When the input string is exhausted, the call-back function returns *FALSE*, or the automaton becomes stuck (unable to make a transition on the next input symbol) the member function *FAAbs::reportAll* returns the index of the symbol immediately to the right of the last symbol on which a successful transition was made.

The following is an example of the use of a finite automaton.

Example 10.2 (Finite automaton): The following program fragment takes a regular expression, constructs a finite automaton, and processes an input string:

```
#include "com-misc.hpp"
#include "re.hpp"
#include "fas.hpp"
#include <iostream.h>

static int report( int ind ) {
    cout << ind << '\n';
    return( TRUE );
}

void process( const RE& e ) {
    auto FARFA M( e );
    cout << M.reportAll( "hishershey", &report );
    return;
}
```

Header `com-misc.hpp` provides the definition of constants *TRUE* and *FALSE*, while header `fas.hpp` gives the declarations of a number of concrete finite automata. Function *report* is used as the call-back function; it simply prints the index and returns *TRUE* to continue processing. Function `process` takes an *RE* and constructs a local finite automaton (of concrete class *FARFA*). It then uses the automaton processes string `hishershey`, writing the final index to standard output before returning. □

Given these examples, we can now demonstrate a more complex use of a finite automaton.

Example 10.3 (Regular expression pattern matching): In this example, we implement a generalized Aho-Corasick pattern matcher (GAC — as in Section 5.1) which performs regular expression pattern matching. Since regular expression pattern matching is not presently included in the SPARE Parts, this example illustrates how FIRE Lite could be used to implement such pattern matching for a future version of the SPARE Parts. (This example also highlights the fact that the call-back mechanism in FIRE Lite is very similar to the mechanism in the SPARE Parts.)

```
#include "re.hpp"
#include "fas.hpp"
```

```
#include "string.hpp"

class PMRE {
public:
    PMRE( const RE& e ) : M( e ) {}
    int match( const String& S, int cb( int ) ) {
        return( M.reportAll( S, cb ) );
    }
private:
    FARFA M;
};
```

Headers `re.hpp`, `fas.hpp`, and `string.hpp` have all been explained before. Class *PMRE* is the regular expression pattern matching class. Its client interface is modeled on the pattern matching client interfaces used in Chapter 9. The class has a private finite automaton (in this case an *FARFA*) which is constructed from an *RE*. (Note that the constructor of class *PMRE* has an empty body, since the constructor of *FARFA M* does all of the work.) The member function *PMRE::match* takes an input string and a call-back function. It functions in the same way as the call-back mechanism in Chapter 9. The member function is trivial to implement using member *FAAbs::reportAll*. Whenever the finite automaton enters an accepting state, a match has been found and it is reported. □

An important feature of FIRE Lite (like SPARE Parts) is that the call-back client interface implicitly supports multi-threading. See Section 9.2.1 for a discussion of call-back functions and multi-threading.

10.3 The structure of FIRE Lite

It is helpful to have an overview of the structure of FIRE Lite and some of the main classes in the toolkit. In this section, we give such an overview.

In the construction algorithms of Chapter 6, the finite automata that are produced have states containing extra information. In particular, the canonical construction produces automata whose states are dotted regular expressions, or items. Some of the other constructions produce automata with sets of items for states, or sets of positions for states, and so on.

The constructions of Chapter 6 appear as the constructors (taking a regular expression) of various concrete finite automata classes in FIRE Lite. It seems natural that mathematical concepts such as items, sets of items, positions, and sets of positions will also appear in such an implementation. The only potential problem is the performance overhead inherent in implementing automata with states which are sets of items, etc.

The solution used in FIRE Lite is to implement states with internal structure as *abstract-states* during the construction of a finite automaton. The finite automata is constructed using the abstract-states (so that the constructor corresponds to one of the algorithms in

Chapter 6). Once the automaton is fully constructed, the abstract-states are too space and time consuming for use while processing a string (for acceptance by the automaton). Instead, we map the abstract-states (and the transition relations, etc) to *State*s (which are simply integers) using a *StateAssoc* object. Once the mapping is complete, the abstract-states, their transition relations, and the *StateAssoc* object can be destroyed.

For all of the finite automata which are constructed from abstract-states, the constructor takes an initial abstract-state which is used as a 'seed' for constructing the remaining states and the transition relation. For performance reasons, we wish to make the finite automaton constructor a template class whose template argument is the abstract-state class (this would avoid virtual function calls). Unfortunately, most compilers which are presently available do not support template member functions (which have recently been added to the draft C++ standard). As a result, we are forced to make the entire finite automata class into a template class. The main disadvantage is that this reduces the amount of object code sharing (see Chapter 8 for a discussion of the differences between source and object code sharing and templates versus inheritance).

Most of the abstract-state classes have constructors which take a regular expression. As a result, an *RE* can be used as argument to most of the finite automata classes — a temporary abstract-state will be constructed automatically.

There are three types of abstract-states, corresponding to finite automata with ε-transitions (see class *FAFA*), ε-*free* finite automata, and deterministic finite automata. Classes of a particular variety of abstract-state all share the same public interface by convention; the template instantiation phase of the compiler detects deviations from the common interface. For more on the three types of abstract-state, see Implementation classes 10.8, 10.11, and 10.13. For examples of particular abstract-state classes, see Section 10.5.4.

The transition relations on *State*s are implemented by classes *StateStateRel* (for ε-transitions), *TransRel* (for nondeterministic transitions), and *DTransRel* (for deterministic transitions).

10.4 *RE*s and abstract *FA*s

In this section, we consider the client interface to regular expressions and the abstract finite automaton class. The multi-threading aspects of the finite automaton interface is not discussed since these aspects are parallel to those presented in Section 9.2.1.

User class 10.4 (*RE*)

File: `re`, `reops`

Description: Class *RE* supports regular expressions. The client interface supports the construction of regular expressions, using all of the operators given in Chapter 2. Operators for stream insertion and extraction are also supported. A number of special-purpose member functions provide information that is primarily used in finite automata constructions. These member functions could have been made protected or private (to hide them from clients), however, this would have required the finite automata classes to be friends of class *RE*.

Implementation: Regular expression are implemented as expression trees, with *RE* being a node in the tree. (Note that this corresponds to the tree definition of regular expressions presented in Chapter 6.) *RE* contains an operator (the operator enumerations are defined as class *REops* in header `reops.hpp`), instead of deriving specific operator nodes from *RE*; this was done for simplicity. *RE* contains pointers to left and right subexpressions. Some of the member functions (which are used by finite automata constructors) perform a tree traversal on their first invocation; the information is cached in the *RE* node, implementing a form of memoization. An alternative implementation would be to store the regular expression as a string in prefix notation — as is done in Grail [RW93].

Performance: For higher performance, more of the member functions could make use of memoization. It is not clear if the regular expression data structures used in FIRE Lite are more efficient than those used in Grail.

□

User class 10.5 (*FAAbs*)

File: `faabs`

Description: This abstract class provides the client interface to all of the finite automata classes. It defines the member functions shown in the examples in Section 10.2. For clarity, we present the header `faabs.hpp` here:

```
/* (c) Copyright 1995 by Bruce W. Watson */
// FIRE Lite class library.
// $Revision:$
// $Date:$
#ifndef FAABS_HPP
#define FAABS_HPP
#define IN_FAABS_HPP

#include "string.hpp"

// Give a generic interface to the finite automata in FIRE
// Lite. This interface differs from the one in the FIRE
// Engine. The new interface makes use of 'call-backs' in
// the same way that the SPARE Parts class library does.

class FAAbs {
public:
```

```
        // In the concrete classes, there will be constructors
        // from regular expressions, etc.

        // Is the input String accepted? This member returns the
        // TRUE if the string is accepted. The int& parameter will
        // contain the index to the right of the last character
        // processed.
        virtual int attemptAccept( const String& S,
                        int& index ) const = 0;

        // Process the input String, calling back whenever an
        // accepting state is entered. The return value is the
        // index to the right of the last character processed.
        // At each call-back, the index to the right of the last
        // processed symbol is passed. If the call-back function
        // returns FALSE, the acceptance attempt is aborted.
        virtual int reportAll( const String& S,
                        int callBack (int) ) const = 0;

        // How many states in this finite automaton?
        virtual int numStates() const = 0;
};

#undef IN_FAABS_HPP
#endif
```

Implementation: As an abstract class, there is no implementation.

□

10.5 Concrete *FA*s

A number of concrete finite automata are provided to implement the client interface defined by *FAAbs*. A summary of the classes and their template arguments (if any) is given in Section 10.5.5. The automata are divided into three types: automata with ε-transitions, automata without ε-transitions, and deterministic automata. All of the classes have names that are prefixed by *FA*. Some of the classes are in fact templates. The following sections contain descriptions of the different classes. All finite automata are declared in header `fas.hpp`.

10.5.1 Finite automata

There are two classes (User class 10.6 — a non-template class, and User class 10.7 — a template class) implementing finite automata with ε-transitions.

User class 10.6 (*FACanonical*)

File: `fa-canon`

Description: This class is a non-template finite automaton class. It implements the canonical finite automata construction algorithm (see Construction 6.15). It inherits from *FAAbs* and implements the required interface.

Implementation: The class is implemented by maintaining the dot movement relation (as an *ItemItemRel*), the set of *CharRange* nodes (as a *NodeSet*), and the set of labels of these nodes (as a *NodeTo*<*CharRange*>). The simulation of the automaton is trivially implemented, with a single helper function.

Performance: The implementation of the main member functions is straight-forward. Performance can be improved by improving the components.

□

User class 10.7 (*FAFA*)

File: `fa-fa`

Description: This finite automata template class inherits from *FAAbs* and implements that public interface. The template argument must be one of the abstract-state classes (names beginning with *AS*...). The constructor takes an object of class *AS*... and uses it to construct the ε-transition relation and the labeled transition relation as well as a single start state and the set of final states. Most of the abstract-state classes have constructors which take an *RE*, enabling us to use an *RE* as the argument to the *FAFA* constructor.

Implementation: The abstract-states are described further in Implementation class 10.8. Making the whole of class *FAFA* a template class would not be necessary if we could make the constructor a template member function. The template argument is only used in the construction of local variables in the constructor. At this time, most C++ compilers do not yet support template member functions. Making the whole class a template class has the disadvantage that code sharing is only done at the source level and objects of two different instantiations of the template are not interchangeable according to the language definition, even though their subobjects have identical types.

The constructor assumes that its argument (an object of the abstract-state class) is the start state. It uses a reachability based algorithm to construct the rest of the transition relations, and the set of final states. A *StateAssoc* object (see Implementation class 10.24) is used to give names to each of the abstract-states.

Performance: The performance is already highly tuned. It may be improved by using use-counting in the abstract-states.

□

Implementation class 10.8 (*AS*...)

File: `as...`

Description: This family of classes implements abstract-states. An abstract-state is one which contains more information than a usual *State*. The various classes do not have a common inheritance ancestor to ensure a consistent interface since the return types of some of the member functions must vary depending upon the abstract-state class. Instead, the common interface is established by convention — errors are detected by the template instantiating phase of the compiler.

Implementation: Each of the abstract-states is described individually in Section 10.5.4.

□

10.5.2 ε-*free* finite automata

The ε-*free* finite automata are also implemented via both a template and a non-template class. All of the automata of this type (including instantiations of the template) are declared in header `fas.hpp`.

User class 10.9 (*FARFA*)

File: `fa-rfa`

Description: This non-template ε-*free* finite automaton class inherits from *FAAbs* and implements the public interface defined there. This class corresponds to the *reduced finite automata* defined in the original taxonomy of construction algorithms, [Wat93a]. The constructor takes a regular expression.

Implementation: This class is implemented through a pair of *PosnSet*s (representing the sets *First* and *Last* from Chapter 6), a *PosnPosnRel* (representing the *Follow* relation), a *NodeTo*<*CharRange*> (mapping the 'positions' to their labels in the regular expression), and an **int** (representing the Boolean *Null* — whether the automaton should accept ε). These components are constructed from the *RE* argument to the constructor.

Performance: The implementation is already highly tuned. The only area for improvement would be in use-counting the subobjects.

□

User class 10.10 (*FAEFFA*)

File: `fa-effa`

Description: This template class inherits from *FAAbs* and implements the finite automata interface. The template argument must be one of the ε-*free* abstract-states (classes with names beginning with *ASEF*...). Again, most of the abstract-states have constructors which take an *RE*, allowing us to use a regular expression as argument to the *FAEFFA*.

Implementation: The implementation corresponds very closely to that of *FAFA* (see User class 10.7).

□

Implementation class 10.11 (*ASEF*...)

File: `asef...`

Description: This family of classes implement ε-*free* states for use as template argument to class *FAEFFA*. Their interface closely parallels that of the *AS*... classes.

Implementation: Same as the *AS*... classes.

□

10.5.3 Deterministic finite automata

The deterministic finite automata are only implemented by a template class. Instead of manually instantiating the template, the client should include the header `fas.hpp` which declares the different variants.

User class 10.12 (*FADFA*)

File: `fa-dfa`

Description: *FADFA* is a template class which inherits from *FAAbs* and implements the interface defined there. The template argument must be one of the deterministic abstract-state classes — classes with names beginning with *ASD*.... As with the other abstract-state classes, the deterministic ones usually have constructors which take an *RE* — meaning that we can use an *RE* as argument to the *FADFA* constructor.

Implementation: The implementation parallels that of class *FAFA* (see User class 10.7).

□

Implementation class 10.13 (*ASD*...)

File: `asd...`

Description: These classes implement deterministic abstract-states for use as template arguments to class *FADFA*. Their (common) interface parallels that of the *AS*... classes.

Implementation: Same as the *AS*... classes.

□

10.5.4 Abstract-states classes

Three of the finite automata classes defined so far are template classes, which expect their template argument to be an *abstract-state*. As a result, there are three primary types of abstract-state. The ε-*free* and deterministic abstract-states have names prefixed (respectively) with *ASEF*... and *ASD*.... The other abstract-states have names prefixed with *AS*.... The types of abstract-states have been grouped together according to their general implementation ideas (as taken from Chapter 6).

Implementation class 10.14 (*ASItems*, *ASEFItems*, *ASDItems*)

File: `asitems`, `asefitem`, `asditems`

Description: These abstract-states all implement constructions which are based upon sets of items (dotted regular expressions).

Implementation: They all maintain a *ItemItemRel* (representing the dot movement relation). The first abstract-state to be constructed must have an *RE* as its argument; the dot movement relation is obtained from the *RE*. The remaining abstract-states are constructed from a private constructor.

Performance: These classes all pay a large penalty due to keeping their own *ItemItemRel* (the dot movement relation). Use-counting *ItemItemRel* or its subobjects would make a significant improvement.

□

Implementation class 10.15 (*ASDItemsDeRemer*, *ASDItemsWatson*)

File: `asditder`, `asditwat`

Description: These two abstract-states implement the filtered deterministic abstract-states based upon sets of items.

Implementation: Their implementation is similar to those in Implementation class 10.14. Additionally, objects of these classes all maintain their own copy of the filter, which is applied after the closure operation.

Performance: These classes suffer from the same performance penalties as the ones in Implementation class 10.14.

□

Implementation class 10.16 (*ASEFPosnsBS*, *ASEFPosnsBSdual*)

File: `asefpbs`, `asefpbsd`

Description: These two abstract-states are used in the constructions which are based upon 'positions' in a regular expression (see Constructions 6.39 and 6.65). The first class implements the Berry-Sethi construction, while the second one implements the dual of the Berry-Sethi construction (see Chapter 6).

Implementation: The implementation is via a local *PosnPosnRel* (the *Follow* relation) and a local *PosnSet* (the *Last* relation). The first abstract-state is constructed from an *RE* (from which it obtains the *PosnPosnRel* and the *PosnSet*). The remaining ones are constructed using a private constructor.

Performance: As with the item based abstract-states (Implementation class 10.14, for example), these classes suffer a penalty for keeping a local copy of the *PosnPosnRel*. This can be alleviated by use-counting *PosnPosnRel*.

□

Implementation class 10.17 (*ASDPosnsMYG*, *ASDPosnsASU*)

File: `asdpmyg`, `asdpasu`

Description: These two abstract-states are deterministic versions of the abstract-state classes described in Implementation class 10.16. The first implements the McNaughton-Yamada-Glushkov construction, while the second one implements the Aho-Sethi-Ullman construction (see Chapter 6).

Implementation: The implementation parallels that given in Implementation class 10.16.

Performance: These classes have the same performance problems as *ASEFPosnsBS* and *ASEFPosnsBSdual*.

□

Implementation class 10.18 (*ASEFPDerivative*, *ASDDerivative*)

File: `asefpder`, `asdderiv`

Description: These abstract-state classes are used to represent a derivative (a regular expression or a set of regular expressions) in Antimirov's and Brzozowski's constructions. Most of the member functions (required of the *ASEF*... and *ASD*... interfaces) are derivative operations on the regular expression representation.

Implementation: This class contains a local copy of the regular expression that it represents. The abstract-state member functions are pass-throughs to the derivatives member functions of class *RE*.

Performance: The copying and comparison operations on regular expressions can be particularly slow. The performance of this class could be significantly improved by use-counting *RE*.

□

10.5.5 Summary of concrete automata classes

The following table presents a summary of the various concrete automata classes and their template arguments (if any):

Class	*Description*	
FACanonical	Canonical automaton	
FAFA	Automaton template, with ε-transitions Single template argument required:	
	Abstract states	
	ASItems	Item sets (canonical)
FARFA	ε-*free* automaton	
FAEFFA	Automaton template, without ε-transitions Single template argument required:	
	Abstract states	
	ASEFItems	Items sets (no filter)
	ASEFPosnsBS	Berry-Sethi
	ASEFPosnsBSdual	dual of Berry-Sethi
	ASEFPDerivative	Antimirov
FADFA	Deterministic automaton template Single template argument required:	
	Abstract states	
	ASDItems	Items sets (no filter)
	ASDItemsDeRemer	Items sets (DeRemer filter)
	ASDItemsWatson	Items sets ($\mathcal{W}$ filter)
	ASDPosnsMYG	McNaughton-Yamada-Glushkov
	ASDPosnsASU	Aho-Sethi-Ullman
	ASDDerivative	Brzozowski

10.6 Foundation classes

A number of the foundation classes presented in Chapter 9 and used in the SPARE Parts have been reused in FIRE Lite. They are: *State*, *String*, *Array*, *Set*, and *StateTo*. In addition to the corresponding headers, FIRE Lite also makes use of headers `com-misc.hpp` and `com-opt.hpp` for various miscellaneous declarations and definitions.

In this section, we consider the additional foundation classes that have been defined for use in FIRE Lite. We begin with character ranges (and sets of them) which are used as atomic regular expressions. We then continue with bit vectors (bit strings) and sets of integers, followed by transition relations and other relations.

10.6.1 Character ranges

Instead of restricting atomic regular expression and transition labels (in finite automata) to single characters, we permit the use of a range of characters (assuming some ordering on the characters, such as the ASCII ordering). Such a range of characters is denoted by it upper and lower (inclusive) bounds, represented by a *CharRange*. Sets of such character ranges can be combined in a *Set<CharRange>*, or in *CRSet* in which the *CharRange*s may be split so that they are disjoint.

Implementation class 10.19 (*CharRange*)

File: `charrang`

Description: This class is used to represent a range of characters (under the character ordering of the execution platform — usually ASCII). A *CharRange* can be constructed by specifying the lower and upper (inclusive) bounds, or a single character. Member functions include one (taking a character) which determines if the character falls within the range. An ordering is also defined on the *CharRange*s.

Implementation: The lower and upper bounds are simply stored privately. The ordering member function implements the lexicographic order on pairs of characters.

Performance: This class is used so heavily that all of the (small) member functions must be inline.

□

Implementation class 10.20 (*CRSet*)

File: `crset`

Description: *CharRange*s can be combined into a *CRSet*. As the *CharRange*s are added, they may be split so that none of them in a *CRSet* overlap. (Splitting, instead of joining, is used since *CRSet*s are used to implement deterministic finite automata.) A *CRSet* is useful for constructing deterministic finite automata. Two *CRSet*s can be combined into one. There are member functions to iterate over the set.

Implementation: The class is implemented via an *Array<CharRange>*. The member functions to add new character ranges ensures that the elements of the array do not overlap.

Performance: All of the member functions are small enough to be inline.

□

10.6.2 States, positions, nodes, and maps

The definition of *State* is borrowed from the SPARE Parts. The concepts of items and positions (in a regular expression) are defined in a similar way. Additionally, mappings from a state, a position, or a node to some type T are defined.

Implementation class 10.21 (*StatePool*)

File: `st-pool`

Description: This class represents a set of states (starting at constant *FIRSTSTATE*) which can be assigned one-by-one for use in constructing finite automata.

Implementation: An integer is stored to track the last state allocated.

□

Implementation class 10.22 (*Node*, *Posn*)

File: `node`, `posn`

Description: The nodes and the positions (those nodes that are labeled with a symbol) of the tree representation of a regular expression are denoted by a *Node* (respectively *Posn*).

Implementation: For simplicity, nodes and positions are encoded as integers representing their pre-order traversal order. Their definition is via **typedef**s.

□

Implementation class 10.23 (*NodeTo*, *PosnTo*)

File: `nodeto`, `posnto`

Description: These two classes operate in a manner similar to *StateTo* (see Implementation class 9.37). They are template classes that implement maps from nodes (respectively positions) to objects of the argument class. The member classes correspond roughly to those in *StateTo*.

Implementation: As with *StateTo*, classes *NodeTo* and *PosnTo* are implemented using *Array*.

□

Implementation class 10.24 (*StateAssoc*)

File: `st-assoc`

Description: This template class takes a single template parameter — usually one of the abstract-state classes. The constructor of *StateAssoc* takes a reference to a *StatePool*. It associates *State*s with objects of the template argument class by assigning a new *State* to each new object that is looked up. Due to possible ambiguity, it is not possible to use *State* as the template argument.

Implementation: The implementation is through an *Array*, which is linearly searched during lookups. Any other (perhaps more efficient) implementation would have to make some assumptions on the template argument.

□

10.6.3 Bit vectors and sets

In this section, we describe bit vectors and integer sets for the basis for storing sets of *State*s.

Implementation class 10.25 (*BitVec*)

File: `bitvec`

Description: Class *BitVec* implements strings of bits. Typical operations, such as bitwise and, or, exclusive or, complement, and shift are provided. Additional member functions are provided to determine if a particular bit is set, and to set a particular bit. The binary operators require that the two *BitVec*s be of the same width (in number of bits). This class can be replaced by the bit vector class in the draft of the C++ standard.

Implementation: The class is implemented as *Array*< **unsigned int** >. Taken end-to-end, these integers form the bit vector. Most of the member functions operate at the array level. Special member functions are provided to index a particular bit, according to its position in the array of integers and its shift distance from the right.

Performance: A low level class like this should be implemented in assembly language for best performance. The bit index calculation member functions are heavily used and must be inline. The current performance is almost optimal for portable C++.

□

Implementation class 10.26 (*IntSet*)

File: `intset`

Description: Class *IntSet* implements set of unsigned integers. Typical set operations are supported. This class performs more efficiently than would *Set*< **unsigned int** >. There is a restriction: the largest integer that can be stored must be set by the client (it will not automatically grow as with template class *Set*).

Implementation: The class is implemented through *BitVec*. Most of the member functions are simple pass-throughs to *BitVec* members.

Performance: Most of the member functions are simple and should be inlined.

□

Implementation class 10.27 (*StateSet*, *NodeSet*, *PosnSet*)

File: `stateset`, `nodeset`, `posnset`

Description: These three classes support sets of *States*, *Nodes*, and *Posns* (respectively). They are used in the construction of automata and in the processing of strings. Most of the typical set operations, such as union, intersection, difference, membership, and cardinality, are supported.

Implementation: Since *State*, *Node*, and *Posn* are all **typedef**'d as **int**, these sets are all **typedef**'d as *IntSet*.

□

Implementation class 10.28 (*ItemSet*)

File: `itemset`

Description: Sets of items (dotted regular expressions) are represented by *ItemSet*. There are two types of items: those that are a dot before a node in the regular expression tree, and those that are a dot after a node. The usual set operations are available. This class is more efficient than using template class *Set*.

Implementation: The dots before and the dots after are stored as two separate *NodeSets*. The set operations are simple pass-throughs to the underlying *NodeSet* members.

Performance: The member functions are simple enough to be inlined.

□

10.6.4 Transitions

Individual transitions and sets of transitions are implemented as individual classes for use in transition relations of finite automata.

Implementation class 10.29 (*TransPair*)

File: `tr-pair`

Description: A *TransPair* represents a transition: a destination *State* and a *CharRange* label. The member functions consist only of access members.

Implementation: The class is little more than a **struct** with access member functions.

Performance: All of the member functions must be inline.

□

Implementation class 10.30 (*Trans*)

File: `trans`

Description: A *Trans* implements a set of transitions. Over and above the usual set operations, it includes members to compute the image (a *StateSet*) of the *Trans* under a character. The class is used to implement nondeterministic transition relations.

Implementation: The implementation is through a *Array*<*TransPair*>. Additionally, an integer is used to store the width of the *StateSet* returned by the image function.

□

Implementation class 10.31 (*DTrans*)

File: `dtrans`

Description: *DTrans* represents a deterministic set of transitions. As with *Trans*, it includes a number of set operations. A member function computes the image (a *State*) of the *DTrans* under a character; if there is no applicable transition, the image is constant *INVALIDSTATE*. The class is used to implement deterministic transition relations.

Implementation: The implementation uses a *Array*<*TransPair*>. The member functions used to construct the set of transition functions assume that the caller (the code adding the transitions) will ensure that the set remains deterministic.

□

10.6.5 Relations

Relations are used to implement ε-transitions and labeled transitions. The different types of relations are used for different types of automata, depending upon the type of states used in the automata.

Implementation class 10.32 (*IntIntRel*)

File: `in-inrel`

Description: A binary relation on integers is implemented using a *IntIntRel*. The image (or the reflexive and transitive closure) of an integer or an *IntSet* can be computed (as an *IntSet*). Pairs of integers can be added or removed from the relation. Additionally, the cross product of two *IntSet*s can be added to the relation.

Implementation: The implementation uses an *Array*<*IntSet*>. Most of the member functions simply make use of the underlying *Array* or *IntSet* member functions.

Performance: The reflexive and transitive closure member function could be implemented more efficiently.

□

Implementation class 10.33 (*StateStateRel*, *NodeNodeRel*, *PosnPosnRel*)

File: `st-strel`, `no-norel`, `po-porel`

Description: The four classes represent binary relations on *State*s, *Node*s, and *Posn*s (respectively). See *IntIntRel* for an explanation of the member functions available. These classes are used directly in the finite automata implementations.

Implementation: The classes are **typedef**s of *IntIntRel*.

□

Implementation class 10.34 (*ItemItemRel*)

File: `it-itrel`

Description: *ItemItemRel*s are binary relations on items (dotted regular expressions). As with *IntIntRel*, the usual relation member functions are implemented. Additional member functions allow new pairs to be added to the relation.

Implementation: Since the *ItemSet*s are already split into the before and after the node components, we store the *ItemItemRel* as four *NodeNodeRel*s representing the following pair types:

- Before-nodes and before-nodes

- Before-nodes and after-nodes
- After-nodes and before-nodes
- After-nodes and after-nodes

The member functions make use of the three sub-objects.

Performance: The performance depends entirely upon that of *NodeNodeRel*.

□

Implementation class 10.35 (*TransRel*)

File: `transrel`

Description: Class *TransRel* implements a nondeterministic labeled transition relation. The class is used to implement the (non-ε) transitions in finite automata. Member functions are available to compute the image of a *StateSet* and a character and to add transition triples (source *State*, label, destination *State*) to the relation.

Implementation: The class uses a *StateTo*<*Trans*>. Most of the member functions are pass-throughs to the *Trans* members.

□

Implementation class 10.36 (*DTransRel*)

File: `dtransre`

Description: Class *DTransRel* is a deterministic labeled transition relation. The class is used as the transition relation in deterministic finite automata. The member functions correspond to those in *TransRel*.

Implementation: The class uses a *StateTo*<*DTrans*>. The member functions are pass-throughs to the *DTrans* member functions.

□

10.7 Experiences and conclusions

A large number of people (worldwide) have made use of the FIRE Engine, and a number of people have started using FIRE Lite. As a result, a great deal of experience and feedback has been gained with the use of the finite automata toolkits. Some of these are listed here.

- The FIRE Engine and FIRE Lite toolkits were both created before the SPARE Parts. Without experience writing class libraries, it was difficult to devise a general purpose toolkit without having a good idea of what potential users would use the toolkit for. FIRE Lite has evolved to a form which now resembles the SPARE Parts (for example, the use of call-back functions).

- The FIRE Engine interface proved to be general enough to find use in the following areas: compiler construction, hardware modeling, and computational biology. The additional flexibility introduced with the FIRE Lite (the call-back interface and multi-threading) promises to make FIRE Lite even more widely applicable.
- Thanks to the documentation and structure of the FIRE Engine and FIRE Lite, they have both been useful study materials for students of introductory automata courses.
- The SPARE Parts was developed some two years after the taxonomy in Chapter 4 had been completed. By contrast, FIRE Lite was constructed concurrently with the taxonomy presented in Chapter 6. As a result, the design phase was considerably more difficult (than for the SPARE Parts) without a solid and complete foundation of abstract algorithms.
- After maintaining and modifying several upgrades of the FIRE Engine, FIRE Lite is likely to be considerably easier to maintain and enhance.

10.8 Obtaining and compiling FIRE Lite

FIRE Lite is available for anonymous ftp from `ftp.win.tue.nl` in directory:

`/pub/techreports/pi/watson.phd/firelite/`

The toolkit (and any related documentation) is combined into a `tar` file. A number of different versions of this file are stored — each having been compressed with a different compression utility.

FIRE Lite was primarily developed under WATCOM C++32 Version 9.5b on MS-DOS. The toolkit was also successfully compiled with BORLAND C++ Versions 3.1 and 4.0 under MICROSOFT WINDOWS 3.1. A number of people have successfully ported the FIRE Engine to UNIX platforms and there is every reason to believe that FIRE Lite will also be easy to port to any platform with a good C++ compiler.

As with the SPARE Parts, a version of FIRE Lite will remain freely available (though I retain the copyright on my code). Contributions to FIRE Lite are welcome.

Chapter 11

DFA minimization algorithms in FIRE Lite

In this chapter, we describe the implementation of the *DFA* minimization algorithms in the finite automata toolkit known as FIRE Lite.

11.1 Introduction

The FIRE Lite finite automata toolkit contains a number of implementations of *DFA* minimization algorithms as member functions of the C++ class *FADFA*.

Other toolkits that are available are also described (and compared to FIRE Lite) in Chapter 10. Only the Grail toolkit supports C++ class libraries of finite automata in the same way as FIRE Lite. While Grail supports a number of things that FIRE Lite does not (such as string elements that are not characters, but more complex structures), it does not have a selection of minimization algorithms as extensive as those in FIRE Lite.

Providing clients of FIRE Lite with a number of different minimization algorithms has proven to be useful. As we shall see in Chapter 15, each of the algorithms has a different performance profile (when graphed against statistics on the *DFA*s to be minimized). As a result, the choice of which algorithm to use will depend upon the client's particular application area.

To a large extent, these implementations are inherited from the previous Eindhoven finite automata toolkit known as the FIRE Engine. FIRE Lite was described in detail in Chapter 10.

This chapter is structured as follows:

- Section 11.2 provides a description of each of the algorithms.
- Section 11.3 describes the foundation classes that are used by the minimization algorithms.
- Section 11.4 presents the conclusions of this chapter.

Since the minimization member functions are part of the definition of class *FADFA* (and therefore the member functions are bundled with the toolkit), no separate information is provided on obtaining and compiling the minimization algorithms.

11.2 The algorithms

As in the taxonomy presented in Chapter 7, we implement two basic types of algorithms: Brzozowski's minimization algorithm and the minimization algorithms based upon computing an equivalence relation on states. In the next two sections, these two types of algorithms are presented.

11.2.1 Brzozowski's algorithm

Brzozowski's algorithm involves two applications of a function which reverses the *DFA* and makes it deterministic (since the reversal of a *DFA* is not necessarily deterministic). Template class *FADFA* has a member function to reverse the automaton and perform the subset construction. This member function is then used in Brzozowski's minimization algorithm.

User function 11.1 (void *FADFA::reverse***)**

File: `fa-dfa`

Description: This member function is used to reverse a deterministic finite automaton. It reverses all of the transitions of the automaton and makes the automaton deterministic. It has no return value.

Implementation: Member function *reverse* uses the deterministic abstract state mechanism to reverse the automaton. It uses class *ASDReverse*, which is local to class *FADFA*. The local class assists in constructing the reverse of the transition relation.

□

User function 11.2 (void *FADFA::minBrzozowski***)**

File: `fa-dfa`

Description: This member function implements Brzozowski's minimization algorithm. It minimizes the *FADFA* on which it is called and it has no return value.

Implementation: This function makes two calls to member function *FADFA::reverse*.

□

11.2.2 Equivalence relation algorithms

As described in Chapter 7, a number of the minimization algorithms are based upon the computation of an equivalence relation on states (of the *DFA*). In this section, we describe the implementation of a number of these algorithms. Some special member functions are provided to assist in the actual compression of the *FADFA* once the equivalence relation has been computed. In Section 11.3, we will consider some foundation classes which implement equivalence relations.

Implementation function 11.3 (void *FADFA::compress*)

File: `fa-dfa`

Description: Given some equivalence relation on *State*s, this member function compresses the *FADFA*. For each equivalence class of the relation, it constructs a *State* in the new automaton. Since some of the minimization member functions construct a *StateEqRel* and some others construct a *StateStateSymRel*, this member function is overloaded to deal with both.

Implementation: This member traverses the set of equivalence classes of its argument, constructing a new *State* for each of them. It also constructs a *DTransRel*, representing the transition relation, and a new set of final *State*s.

□

Implementation function 11.4 (*State FADFA::split*)

File: `fa-dfa`

Description: This member function is used by some of the minimization algorithms to split equivalence classes of the equivalence relation. It takes a pair of *State*s p and q, a *CharRange* a, and a reference to a *StateEqRel*. It assumes that the p and q are representatives of their particular equivalence classes. It splits the equivalence class of p into the set of *State*s that have a transition to the equivalence class of q on a, and those that do not. If there was a successful split, p will be a representative of one of the new equivalence classes (resulting from the split) and the unique representative of the other new equivalence class is returned. The special *State INVALIDSTATE* is returned if the split was not successful.

Implementation: The implementation is a trivial one. It makes use of the *StateEqRel* received as argument, and the *DTransRel* of the automaton.

□

Given these basic helper member functions, we are now in a position to describe the minimization algorithms themselves.

User function 11.5 (void *FADFA::minDragon*)

File: `fa-dfa`

Description: This member function is an implementation of [ASU86, Algorithm 3.6, p. 141], presented as Algorithm 7.21 in this dissertation. It is named the 'Dragon' minimization algorithm after Aho, Sethi, and Ullman's 'Dragon book'.

Implementation: The algorithm is a straightforward implementation of Algorithm 7.21.

□

User function 11.6 (void *FADFA::minHopcroftUllman*)

File: `fa-dfa`

Description: This member function is an implementation of Hopcroft and Ullman's minimization algorithm [HU79], appearing as Algorithm 7.24 in this dissertation.

Implementation: This algorithm computes the distinguishability relation D. Initially, pairs of *State*s that are distinguishable are those pairs where one is final and the other is non-final. The transition relation is followed in reverse, marking distinguishable pairs. The iteration terminates when all distinguishable *State*s have been considered.

Performance: The algorithm can be expected to run quite slowly since the implementation of the transition relation *DTransRel* is optimized for forward transitions.

□

User function 11.7 (void *FADFA::minHopcroft*)

File: `fa-dfa`

Description: This function implements Hopcroft's minimization algorithm [Hopc71]. The algorithm is presented as Algorithm 7.26 in this dissertation.

Implementation: The member function uses some encoding tricks to efficiently implement the abstract algorithm. The combination of the out-transitions of all of the *State*s is stored in a *CRSet* named C. Set L from the abstract algorithm is implemented as a *StateTo*< **int** >. L is interpreted as follows: if *State* q is a representative, then the following pairs still require processing (they would be in set L in the abstract algorithm):

$$([q], C_0), \ldots, ([q], C_{L[q]-1})$$

The remaining pairs do not require processing:

$$([q], C_{L[q]}), \ldots, ([q], C_{|C|})$$

This implementation facilitates quick scanning of L for the next valid *State-CharRange* pair to be considered.

□

Implementation function 11.8 (int *FADFA::areEq*)

File: `fa-dfa`

Description: This member function is a helper to *FADFA::minWatson*. It implements Algorithm 7.27 described in Section 7.4.6.

Implementation: This member function takes two more parameters than the abstract algorithm: a *StateEqRel* and a *StateStateSymRel*. These parameters are used to discover equivalence (or distinguishability) of *State*s earlier than the abstract algorithm would.

Performance: This member function should use memoization.

□

User function 11.9 (void *FADFA::minWatson***)**

File: `fa-dfa`

Description: This member function implements the new minimization algorithm appearing in Section 7.4.7. This algorithm is particularly interesting since it computes the equivalence relation from the safe side. If follows that this algorithm is usable in real-time applications, where some minimization is desired once a deadline has expired (see Section 7.4.7). The present implementation does not support interruptions in the computation of the equivalence relation.

Implementation: Helper member function *FADFA::areEq* is used.

□

11.3 Foundation classes

Some simple foundation classes are needed only by the minimization member functions of FIRE Lite.

Implementation class 11.10 (*StateStateSymRel*)

File: `sssymrel`

Description: This class is used to implement symmetrical relations on *State*s. In some of the minimization algorithms, a symmetrical relation is used to keep track of the *State*s which have been compared to one another for equivalence.

Implementation: The implementation closely parallels that of class *StateStateRel* (see Implementation class 10.33). The member functions are modified to accomodate the symmetry requirement.

□

Implementation class 11.11 (*StateEqRel*)

File: `st-eqrel`

Description: Class *StateEqRel* implements an equivalence relation on *State*s. It is used to accumulate the approximations of relation E for minimizing an *FADFA*. There are member functions for accessing a unique set of equivalence class representatives and for iterating over the equivalence classes of the relation. Other members are provided to split equivalence classes and to merge equivalence classes.

Implementation: The implementation is via a *StateTo<StateSet*>*. Two *State*s that belong to the same equivalence class point to the same *StateSet*. This allows extremely fast tests for equivalence. The other member functions are simple manipulations of these structures.

□

Implementation class 11.12 (*ASDReverse*)

File: `asdrever`

Description: This abstract deterministic state (see Section 10.5.4 for more on abstract states) is used to compute the transition relation and new set of final states while reversing an *FADFA*.

Implementation: The implementation maintains a *StateSet* (representing the current states) and pointers to the components of the *FADFA*.

□

11.4 Conclusions

A number of the minimization algorithms derived in Chapter 7 have been implemented in FIRE Lite. Although the algorithms have been quite easy to present in an abstract manner in Chapter 7, the work required to implement them was anything but easy. The C++ implementations are usually several times more verbose than the abstract algorithms. The algorithms with the best running time analysis (such as Hopcroft's) also have the most intricate data structures — and therefore, require the most attention in a C++ implementation. Conversely, a minimization algorithm such as Brzozowski's has no data structures and is particularly simple to implement in FIRE Lite. In Chapter 15, we will see how this difference in difficulty of implementation can affect the running time in practice.

Part IV

The performance of the algorithms

Chapter 12

Measuring the performance of algorithms

We consider briefly some of the issues involved in collecting algorithm performance data. Little is known about the real-life performance of the algorithms derived and implemented in the preceding two parts of this dissertation. Such information is crucial to choosing an appropriate algorithm for a given application. In this part, we present performance data for a number of the most important algorithms.

The commercial success of software and hardware products is frequently dictated by the product's performance in practice. As a result, a great deal is known about methods for collecting performance data. Unfortunately, most of what is known relates to collecting benchmarking data for marketing purposes. Collecting and presenting benchmarking (performance) data in a fair way is particularly difficult. The guiding principles used in collecting and analyzing the data given in this part are as follows:

- The performance data should be normalized since we are primarily interested in relative performances of the algorithms.

- To present performance data that is independent of the particular machine used, we execute the benchmarks on a variety of hardware. The relative performance of the algorithms can then be compared across hardware platforms.

- Execute the benchmarks on a single-user machine with little or no operating system overhead.

- Since the benchmarking code should be compiled with compiler optimizations enabled, the assembly language output from the compiler should be inspected to ensure that the compiler does not eliminate significant amounts of code. (This actually occurred during the collection of the benchmarking data presented in Chapter 13. A great deal of work was required to prevent the optimizing compiler from 'optimizing away' the benchmarking code.)

- If a multiple-user operating system is used (such as UNIX), ensure that the benchmarker is the only person logged-in, and that no non-standard background tasks are executed.

- Use a machine with sufficient physical memory so that demand-paging effects are negligible. We ignore the effects of caching, since they are usually inherent in the hardware, and are affected minimally by the operating system.
- Make use of vast amounts of input data for the algorithms. The distribution of the input data is chosen to represent a typical use of the algorithm in practice, e.g. the pattern matching algorithms were tested using English language input data (and the resulting English distribution of words, word lengths, and letter frequencies). Since this is only representative of English text searches, we also make use of genetic sequence data as input; the genetic sequences typically consist of longer patterns than those in English, and have a four letter alphabet.
- Collect a number of statistics on the input data and the program performance. Later, statistical analysis can be used to determine the relevant parameters, which can then be selected for presentation.
- Using the presented performance data, make clear recommendations about which algorithm to use in a given situation.
- The performance data should be compared against the theoretical space and time predictions and against other benchmarking results.

This part is structured as follows:

- Chapter 13 presents performance data for some of the pattern matching algorithms implemented in the SPARE Parts. The algorithms selected were those expected to have the best tradeoff of precomputation time versus pattern matching performance.
- In Chapter 14, we consider the performance of a number of the *FA* construction algorithms implemented in FIRE Lite. We compare the performance of most of the well-known algorithms.
- The performance of the *DFA* minimization algorithms (implemented in FIRE Lite) is presented in Chapter 15.

Chapter 13

The performance of pattern matchers

This chapter presents performance data on some pattern matching algorithms, and recommendations for the selection of an algorithm (given a particular application). The pattern matching problem, and algorithms solving it, are considered in Chapter 4.

The performance of all of the algorithms (running on a variety of workstation hardware) was measured on two types of input: English text and genetic sequences. The input data, which is the same as that used in the benchmarks of Hume and Sunday [HS91], were chosen to be representative of two of the typical uses of pattern matching algorithms. The differences between natural language text and genetic sequences serve to highlight the strengths and weaknesses of each of the algorithms. Until now, the performance of the multiple-keyword algorithms (Aho-Corasick and Commentz-Walter) had not been extensively measured.

The Knuth-Morris-Pratt and Aho-Corasick algorithms performed linearly and consistently (on widely varying keyword sets), as their theoretical running time predicts. The Commentz-Walter algorithm (and its variants) displayed more interesting behaviour, greatly out-performing even the best Aho-Corasick variant on a large portion of the input data. The recommendations section of this chapter details the conditions under which a particular algorithm should be chosen.

An early version of this chapter appeared as [Wat94a].

13.1 Introduction and related work

Each of the algorithms tested involves some sort of precomputation on the set of keywords. Since the time involved in pattern matching usually far outweighs the time involved in precomputation, the performance of the precomputation algorithms is not discussed in this dissertation.

Performance data for the following selection of algorithms are presented in this chapter:

- The Knuth-Morris-Pratt (KMP) algorithm — Algorithm 4.84, appearing on page 77.
- Two variants of the Aho-Corasick (AC) algorithm: the 'optimized' version (AC-OPT — Algorithm 4.53 appearing on page 64), and the failure function version (AC-FAIL

— Algorithm 4.72 appearing on page 73).

- Two variants of the Commentz-Walter (CW) algorithm: the normal Commentz-Walter algorithm (CW-NORM — Algorithm 4.4.5, appearing on page 95), and the weak Boyer-Moore algorithm (CW-WBM — Algorithm 4.4.7, appearing on page 97).

All of the algorithms considered in this chapter have worst-case running time linear in the length of the input string. The running time of the AC-OPT algorithm is independent of the keyword set, while that of the KMP, AC-FAIL, CW-NORM, and CW-WBM algorithms depends (linearly in the case of CW-WBM and CW-NORM) upon the length of the shortest keyword in the keyword set. The KMP, AC-FAIL, CW-NORM, and CW-WBM algorithms can be expected to depend slightly on the keyword set size. Unfortunately, little is known about the relative performance (in practice) of the multiple-keyword algorithms. Only the Aho-Corasick algorithms are used extensively. The Commentz-Walter algorithms are used rarely (if ever), due to the difficulty in correctly deriving the precomputation algorithms for the CW algorithms.

The performance of the single-keyword algorithms in practice has been studied:

- In [Smit82], Smit compares the theoretical running time and the practical running time of the Knuth-Morris-Pratt algorithm, a rudimentary version of the Boyer-Moore algorithm, and a brute-force algorithm.

- In [HS91], Hume and Sunday constructed a taxonomy and explored the performance of most existing versions of the single-keyword Boyer-Moore pattern matching algorithm. Their extensive testing singled out several particularly efficient versions for use in practical applications.

- In [Pirk92], Pirklbauer compares several versions of the Knuth-Morris-Pratt algorithm, several versions of the Boyer-Moore algorithm, and a brute-force algorithm. Since Pirklbauer did not construct a taxonomy of the algorithms, the algorithms are somewhat difficult to compare to one another and the testing of the Boyer-Moore variants is not quite as extensive as the Hume and Sunday taxonomy.

We adopt the approach (due to Hume and Sunday) of evaluating the algorithms on two types of input data: natural language input strings, and input strings encoding genetic (DNA) information. In order to compare our test results with those of Hume and Sunday, we use a superset of the test data they used in [HS91].

This chapter is structured as follows:

- Section 13.2 briefly outlines the algorithms tested.

- Section 13.3 describes the testing methodology, including the test environment, test data (and related statistics), and testing problems that were encountered.

- Section 13.4 presents the results of the testing. Most of the results are presented in the form of performance graphs.

- Section 13.5 gives the conclusions and recommendations of this chapter.

13.2 The algorithms

To re-cap what is known from Chapter 4, the algorithms to be considered are:

- The Knuth-Morris-Pratt algorithm (KMP). This algorithm combines the use of indexing into the input string (and the single-keyword pattern) with a precomputed 'failure function' to simulate a finite automaton. The algorithm never backtracks in the input string.

- The optimized Aho-Corasick algorithm (AC-OPT). This algorithm uses a Moore machine to find matches. The Moore machine detects all matches ending at any given character of the input string. The algorithm never backtracks in the input string (it is an *on-line* algorithm) and examines each character of the input string only once.

- The failure function Aho-Corasick algorithm (AC-FAIL). This algorithm is similar to the AC-OPT algorithm. The Moore machine used in AC-OPT is compressed into two data-structures: a forward trie, and a failure function. These two data-structures can be stored more space-efficiently than the full Moore machine, with a penalty to the running time of the algorithm. The algorithm never backtracks in the input string, but it may examine a single character more than once before proceeding; despite this, it is still linear in the length of the input string.

- The Commentz-Walter algorithms (CW). In all versions of the CW algorithms, a common program skeleton is used with different shift functions. The CW algorithms are similar to the Boyer-Moore algorithm. A match is attempted by scanning backwards through the input string. At the point of a mismatch, something is known about the input string (by the number of characters that were matched before the mismatch). This information is then used as an index into a precomputed table to determine a distance by which to shift before commencing the next match attempt. The two different shift functions compared in this chapter are: the multiple-keyword Boyer-Moore shift function (CW-WBM) and the Commentz-Walter normal shift function (CW-NORM). Recall (from our weakening steps in Section 4.4) that the CW-NORM shift function always yields a shift that is at least as great as that yielded by the CW-WBM shift function.

The precomputation required for each of these algorithms is described in [WZ92].

We are interested in measuring the time required to find all matches in a large input string. In order to do this, we use an algorithm skeleton which repeatedly advances to the next match, registering each one. To eliminate the overhead of function calls, and obtain the raw performance of each of the algorithms, we inline all of the member function calls by hand. For each of the algorithms to be tested, this yields a program which finds all matches in an input string. Interestingly, after inlining, these algorithms are in C [ISO90, KR88], with all C++ features being eliminated during the inlining. The resulting C code is the same as that given in the earlier version of this paper [Wat94a, Appendix A]. The C language was used in order to extract the maximum performance from the implementations — on most workstations, the C compiler is the one with the highest quality of optimization. Having all function calls inlined means that the benchmarking versions of the toolkit will have higher performance than the version for client use. The data presented in this chapter is useful for considering the relative performance of the algorithms.

Efforts were made to implement the algorithms as efficiently as possible, while preserving readability. For example, in the algorithms used in the performance tests, indexing (as opposed to pointers) was used when accessing characters of the input string; most optimizing compilers are able to 'pointerize' such indices. (The pattern matching toolkit presented in [Wat94a, Appendix A] contains pointer versions of the algorithms which can be used with non-optimizing compilers.)

The performance of the precomputation algorithms was not extensively measured. Some simple measurements, however, indicate that all of the algorithms required similar precomputation times for a given set of keywords.

13.3 Testing methodology

The performance of each of the pattern matching algorithms is linear in the size of the input string. The performance of the AC variants and the KMP algorithm is largely independent of the keyword set, while the CW algorithms running time depends on the keyword set. The testing of these algorithms is intended to determine the relative performance of the algorithms on two types of test data (each having different characteristics):

- English text was chosen as the first type of input data since it is the most common input to pattern matching programs such as `fgrep`.
- DNA sequences were chosen as the second type of input data since genome mapping projects make heavy use of pattern matching algorithms, and the characteristics of the input data are unlike the natural language input data.

The testing of the algorithms is also intended to explore the dependence of the performance of the Commentz-Walter algorithms upon the keyword sets.

13.3.1 Test environment

The tests were performed on a DEC Alpha workstation (running OSF/1) with a 100 Mhz clock. A smaller number of tests were also performed on a HP Snake workstation and a Sun Sparc Station 1+. The tests showed that the relative performance data gathered on the Alpha is typical of what could be found on other high performance workstations.

During all tests, only one user (the tester) was logged-in. Typical Unix background processes ran during the tests. Since the running time of the algorithm was obtained using the `getrusage` system call, these background processes did not skew the algorithm performance data. The data-structures used in the testing were all in physical memory during the tests. No page faults were reported by `getrusage`, and all data was accessed before the timed run (to ensure that they were in physical memory). Methods of disabling the cache memory were not explored. All of the frequently accessed data-structures were too large (frequently a megabyte) to fit in a first level cache. The linear memory access behaviour of all of the algorithms means that performance skewing due to caching effects was negligible.

13.3.2 Natural language test data

The test data is a superset of that used by Hume and Sunday [HS91]. The input alphabet consists of the 52 upper-case and lower-case letters of the alphabet, the space, and the new-line characters. The input string is a large portion (999952 bytes) of the bible, organized as one word per line. Each algorithm was run 30 times over the input string, effectively giving an input string of approximately 28 Megabytes in length (assuming that a Megabyte is 2^{20} bytes). The bible was chosen as input since Hume and Sunday used it (and we wish to facilitate comparison of our data with that of Hume and Sunday), and it is freely redistributable.

Some data on the words making up the input string is shown in the following table:

Word length	*Number of words*
1	4403
2	32540
3	55212
4	43838
5	23928
6	13010
7	9946
8	6200
9	4152
10	1851
11	969
12	407
13	213
14	83
15	22
16	5
17	1

There are a total of 196780 words; the mean word length is 4.08 and the standard deviation is 1.97.

The single-keywords sets are the same as those used by Hume and Sunday. They consist of 500 randomly chosen words, 428 of which appear in the input string. Some data on the keywords are shown in the following table:

Word length	*Number of words*
1	0
2	1
3	14
4	47
5	68
6	103
7	79
8	79
9	49
10	29
11	14
12	9
13	7
14	0
15	0
16	1

Note that there are no words of length 1, 14, or 15. The mean word length is 6.95 and the standard deviation is 2.17.

The multiple-keyword sets were all subsets of the words appearing in the input string. Preliminary testing showed that the performance of the algorithms (on English text) is almost entirely independent of the number of matches in the input string (providing that some sensible method of registering matches is used). A total of 4174 different keyword sets were generated using a random number generator `ran1` appearing in [PTVF92, p. 280]. The relatively even distribution of keyword set sizes can be seen in the following table:

Keyword set size	*Number of sets*
1	212
2	217
3	206
4	193
5	199
6	200
7	217
8	206
9	234
10	202
11	235
12	219
13	210
14	210
15	177
16	221
17	196
18	222
19	200
20	198

The mean keyword set size is 10.47 and the standard deviation is 5.73.

An additional statistic (concerning the multiple-keyword sets) was recorded: the length of the shortest keyword in any given keyword set; for a given keyword set the CW-WBM and CW-NORM shift distances are bounded above by the length of the shortest keyword in the set. The data are as follows:

Length of shortest keyword	*Number of keyword sets*
1	4
2	196
3	1033
4	1698
5	719
6	286
7	100
8	70
9	47
10	12
11	5
12	2
13	1
14	1

The mean is 4.19 and the standard deviation is 1.36.

13.3.3 DNA sequence test data

The test data consists of the input string used by Hume and Sunday [HS91], and randomly generated keyword sets. The input alphabet consists of the four letters a, c, g, and t (standing for adenine, guanine, cytosine, and thymine, respectively) used to encode DNA, and the new-line character. The input string is a portion (997642 bytes) of the GenBank DNA database, as distributed by Hume and Sunday. Each algorithm was run 30 times over the input string, effectively giving an input string of approximately 28 Megabytes in length.

A total of 450 keyword sets were randomly chosen (the keywords are all substrings of the input string). Within each keyword set, all keywords were of the same length. The keyword sets were distributed evenly with set sizes ranging from 1 to 10 and keyword lengths ranging from 100 to 900 (in increments of 100).

13.4 Results

The performance of the algorithms was measured on thirty iterations over the input string. The running time on both types of test data was found to be independent of the number of matches found in the input string. This is mostly due to the fact that the algorithms register a match by recording one integer, and incrementing a pointer — cheap operations on most processors.

The performance of each algorithm was graphed against the size of the keyword sets, and against the lengths of the shortest keyword in a set. Graphing the performance against other statistics (such as the sum of the lengths of the keywords, or the length of the longest keyword in a set) was not found to be helpful in comparing the algorithms.

13.4.1 Performance versus keyword set size

For each algorithm, the average number of megabytes (of natural language input string) processed per second was graphed against the size of the keyword set. The four graphs (corresponding to AC-FAIL, AC-OPT, CW-WBM, and CW-NORM) are superimposed in Figure 13.1.

As predicted, the AC-OPT algorithm has performance independent of the keyword set size. The AC-FAIL algorithm has slightly worse performance, with a slight decline as the keyword set size increases. The CW-WBM and CW-NORM algorithms perform similarly to one another, with the CW-NORM algorithm performing slightly better. This follows from the fact that the shift predicate used in the CW-WBM algorithm is a weakening of the one used in the CW-NORM algorithm (see Section 4.4). The performance of both CW algorithms decreases noticeably with increasing keyword set sizes, eventually being outperformed by the AC-OPT algorithm at keyword set sizes greater than 13.

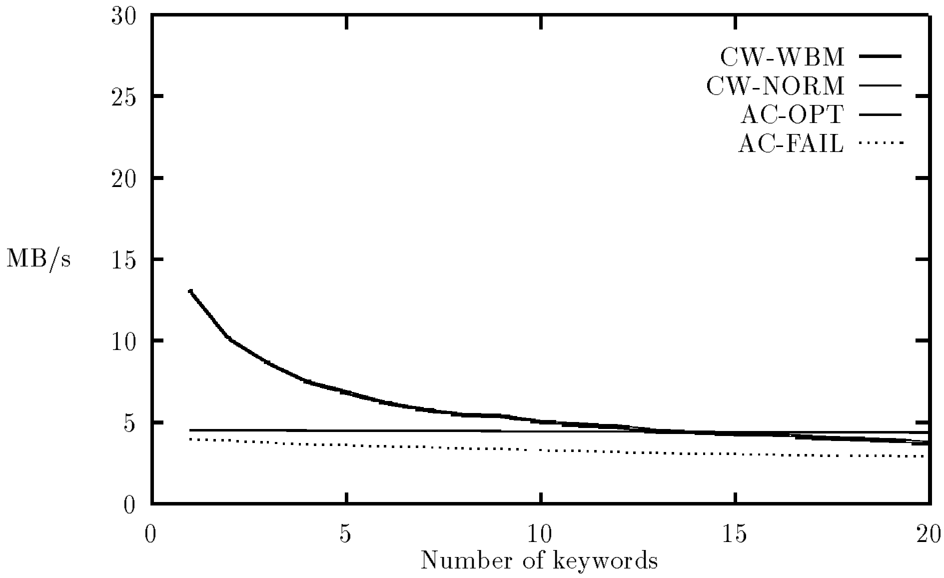


Figure 13.1: Algorithm performance (in megabytes/second) versus keyword set size. The performance lines of the CW-WBM and CW-NORM algorithms are almost coincidental (shown as the solid descending line).

Figure 13.2 presents the ratio of the CW-WBM performance to the CW-NORM performance — clearly showing CW-NORM outperforming CW-WBM. The figure also shows that the performance gap between the two algorithms widens somewhat with increasing keyword set size.

The AC algorithms displayed little to no variance in performance for a given keyword set size. The median performance data (with $+1$ and -1 standard deviation bars) for AC-FAIL are shown in Figure 13.3, while those for AC-OPT are shown in Figure 13.4. In both graphs, the standard deviation bars are very close to the median, indicating that both algorithms display very consistent performance for a given keyword set size.

The CW algorithms displayed a large variance in performance, as is shown in Figures 13.5 and 13.6 (for CW-WBM and CW-NORM, respectively). These figures show a noticeable narrowing of the standard deviation bars as keyword set size increases. The variance in both algorithms is almost entirely due to the variance in minimum keyword length for a given keyword set size.

For each algorithm, the average number of megabytes of DNA input string processed per second was graphed against keyword set size. The results are superimposed in Figure 13.7. The performance of the algorithms on the DNA data was similar to their performance on the natural language input data. The performance of the AC-OPT algorithm was independent of the keyword set size, while the performance of the AC-FAIL algorithm declined slightly with increasing keyword set size.

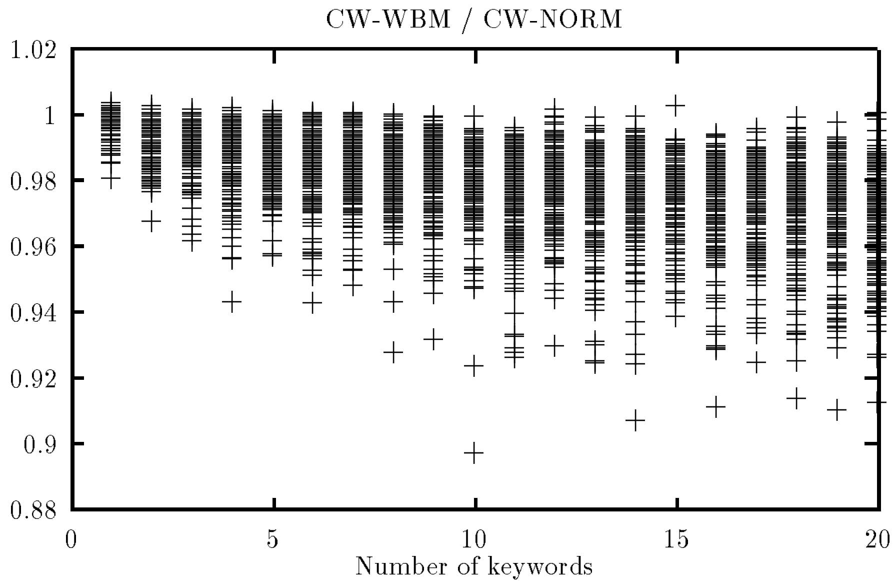


Figure 13.2: The ratio of CW-WBM performance to CW-NORM performance versus keyword set size. Some data-points are greater than 1.00 (although theoretically this should not occur), reflecting timing anomalies due to the limited timer resolution.

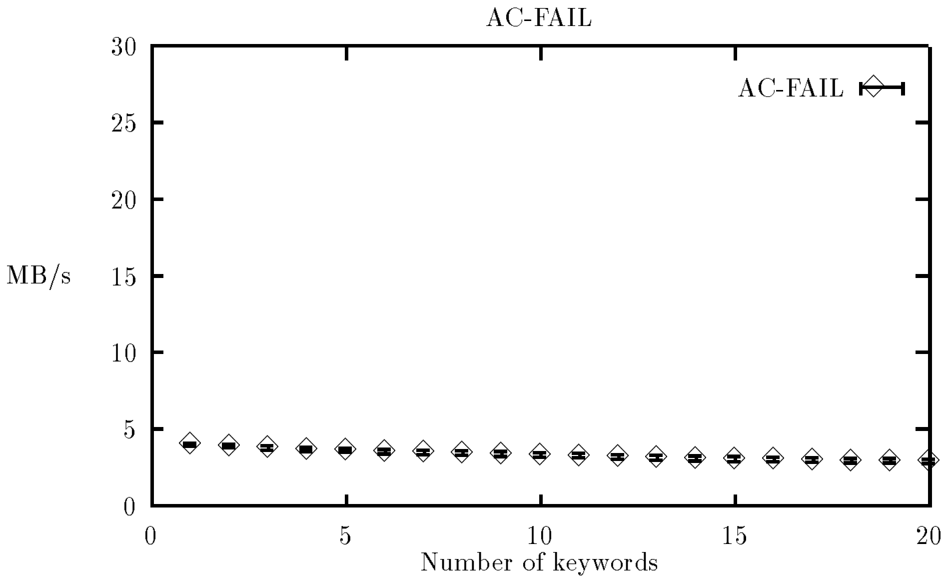


Figure 13.3: Performance variation (in megabytes/second) versus keyword set size for the AC-FAIL algorithm. Median performance is shown as a diamond, with +1 and −1 standard deviation bars.

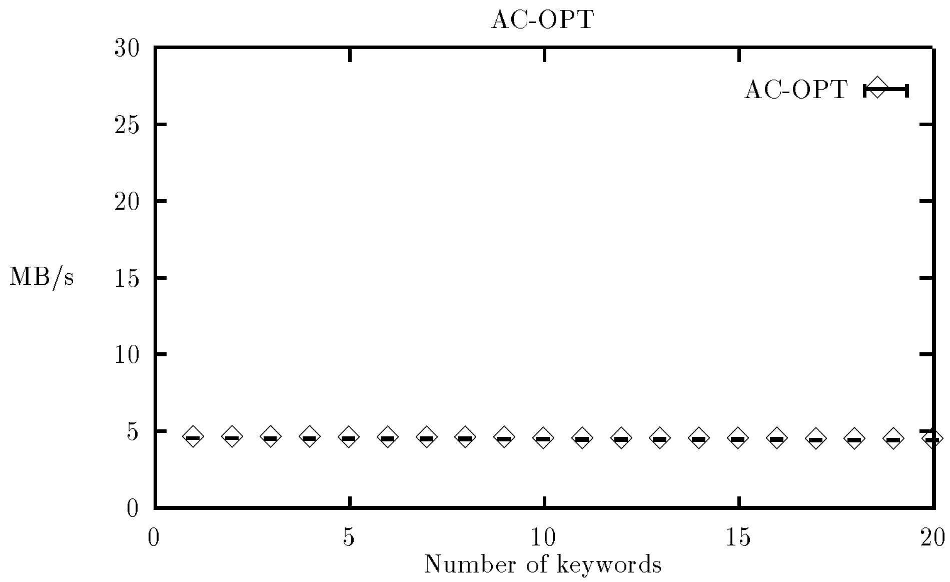


Figure 13.4: Performance variation (in megabytes/second) versus keyword set size for the AC-OPT algorithm.

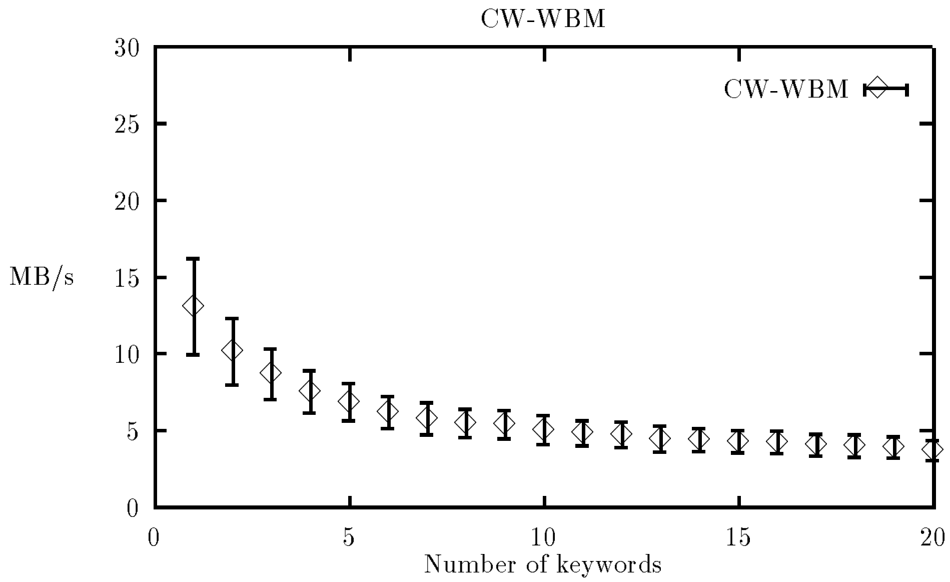


Figure 13.5: Performance variation (in megabytes/second) versus keyword set size for the CW-WBM algorithm.

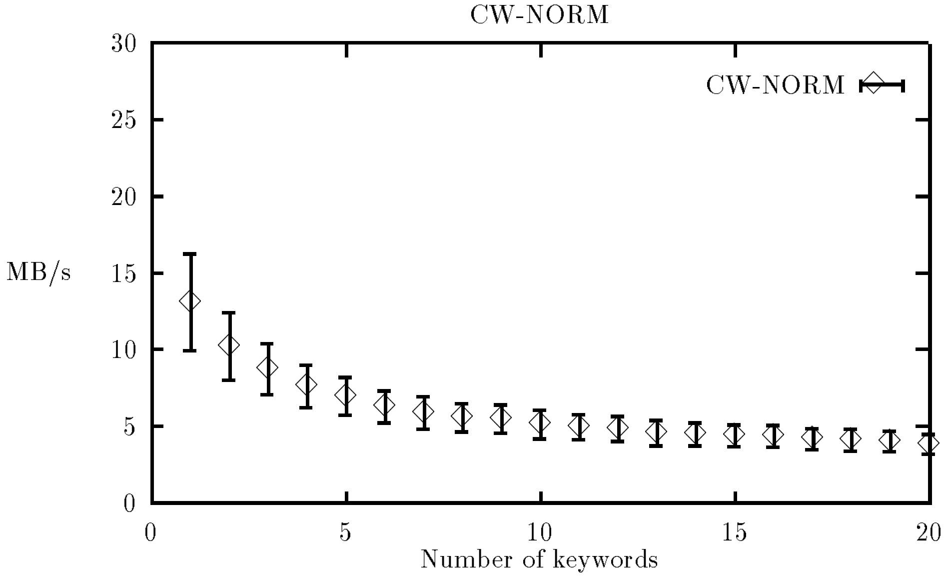


Figure 13.6: Performance variation (in megabytes/second) versus keyword set size for the CW-NORM algorithm.

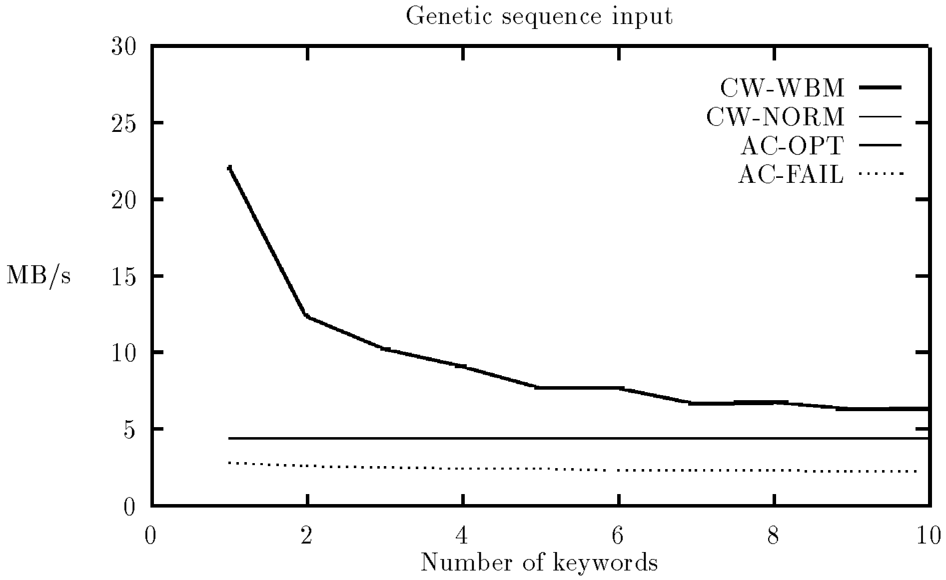


Figure 13.7: Algorithm performance (in megabytes/second) versus keyword set size, for the DNA test data. The performance of the CW-NORM and CW-WBM algorithms are almost coincidental, shown as the descending solid line.

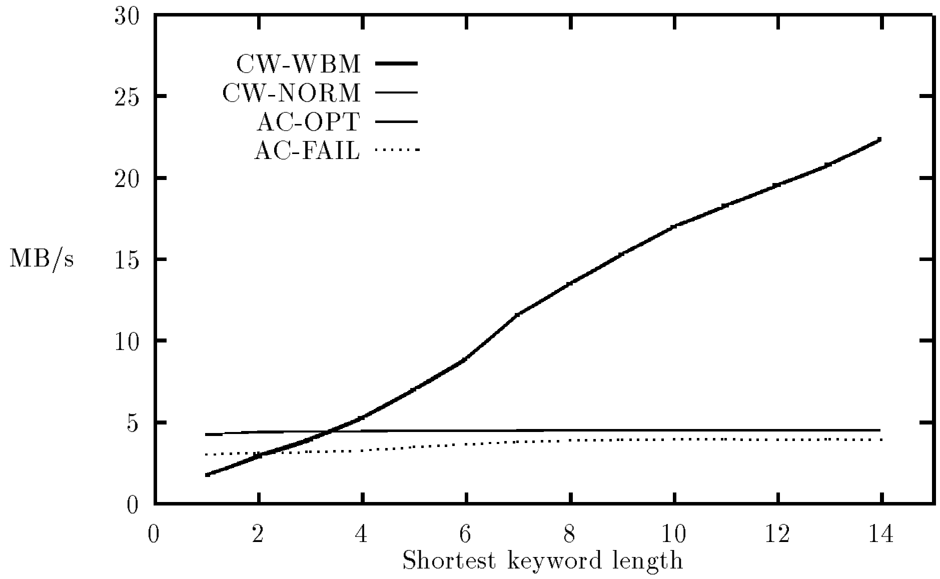


Figure 13.8: Algorithm performance (in megabytes/second) versus the length of the shortest keyword in a given set. The performance of the CW-WBM and CW-NORM algorithms are almost coincidental (shown as the ascending solid line).

The performance of the CW algorithms, which declined with increasing keyword set size, was consistently better than the AC-OPT algorithm. In some cases, the CW-NORM algorithm displayed a five to ten-fold improvement over the AC-OPT algorithm.

13.4.2 Performance versus minimum keyword length

For each algorithm, the average number of megabytes processed per second was graphed against the length of the shortest keyword in a set. For the multiple-keyword tests the graphs are superimposed in Figure 13.8.

Predictably, the AC-OPT algorithm has performance that is independent of the keyword set. The AC-FAIL algorithm has slightly lower performance, improving with longer minimum keywords. The average performance of the CW algorithms improves almost linearly with increasing minimum keyword lengths. The low performance of the CW algorithms for short minimum keyword lengths is explained by the fact that the CW-WBM and CW-NORM shift functions are bounded above by the length of the minimum keyword (see Chapter 4). For sets with minimum keywords no less than than four characters, the CW algorithms outperform the AC algorithms.

As predicted, the CW-NORM algorithm outperforms the CW-WBM algorithm. The performance ratio of the CW-WBM algorithm to the CW-NORM algorithm is shown in Figure 13.9. The figure indicates that the performance gap is wide with small minimum keyword lengths, and diminishes with increasing minimum keyword lengths. (This effect

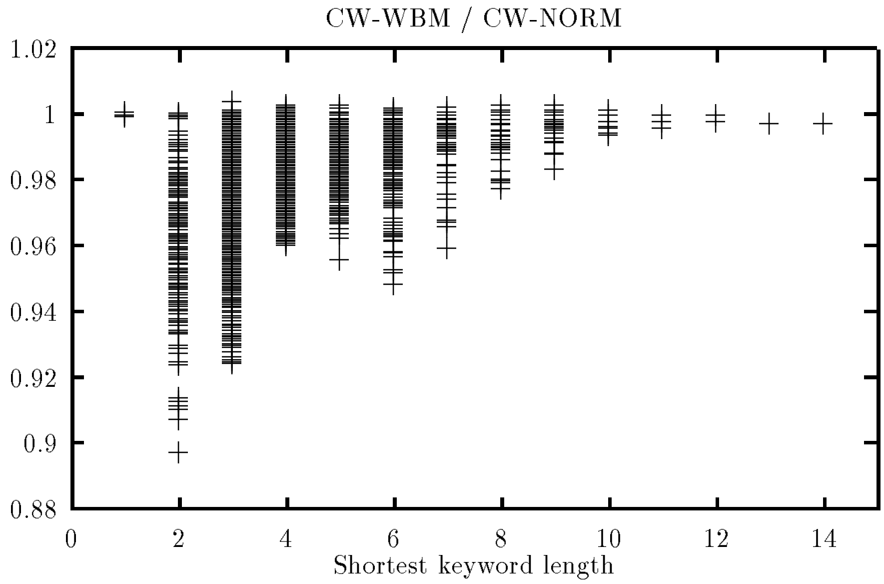


Figure 13.9: The ratio of CW-WBM performance to CW-NORM performance versus the length of the shortest keyword in a set. Some data-points are greater than 1.00, reflecting timing anomalies due to the limited timer resolution.

of the diminishing performance gap is partly due to the distribution of keyword lengths in the test data.)

The AC-FAIL algorithm displayed some variance in performance, as shown in Figure 13.10. The apparent greater variance for shorter minimum keyword lengths is partially due to the distribution of keyword lengths in the test data. The AC-OPT algorithm showed practically no variance in performance, as shown in Figure 13.11.

The CW algorithms displayed a large variance in performance for given minimum keyword lengths. The median performance (with $+1$ and -1 standard deviation bars) of the CW-WBM algorithm are shown in Figure 13.12, while those for CW-NORM are shown in Figure 13.13. The variance increases with increasing shortest keyword length. At a shortest keyword length of 11, the variance decreases abruptly due to the distribution of the shortest keyword lengths of the keyword sets; there are few keyword sets with shortest keyword length greater than 10 characters. The variance in the performance of the CW algorithms is due to the variance in keyword set size (for any given minimum keyword length).

The performance in megabytes of DNA input string processed per second of each algorithm was also graphed against keyword length[1]. The results are superimposed in Figure 13.14. The performance of the algorithms on the DNA data was similar (though not as dramatic) to their performance on the natural language input data. The performance of

[1]Recall that the keywords in a given keyword set were all of the same length.

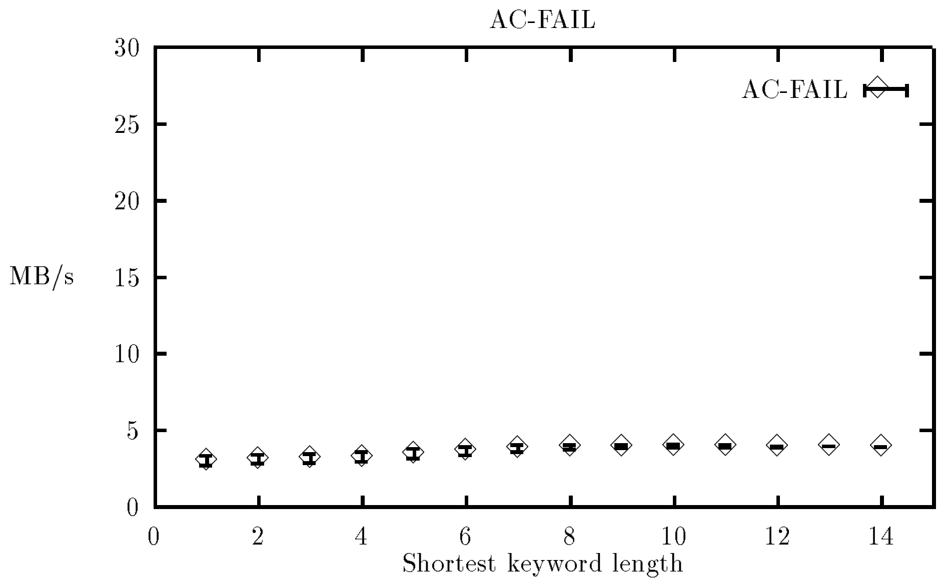


Figure 13.10: Performance variation (in megabytes/second) versus the length of the shortest keyword in a set for the AC-FAIL algorithm.

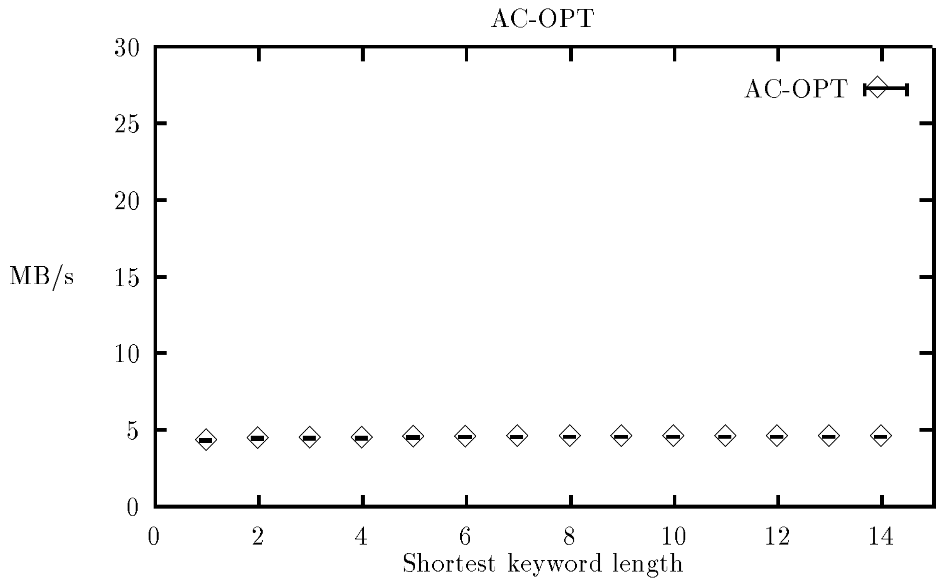


Figure 13.11: Performance variation (in megabytes/second) versus the length of the shortest keyword in a set for the AC-OPT algorithm.

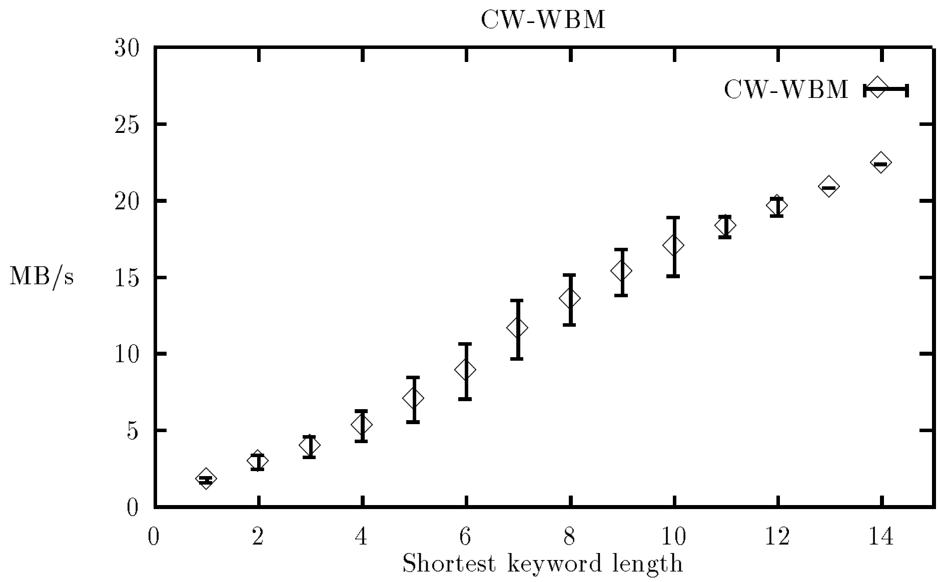


Figure 13.12: Performance variation (in megabytes/second) versus the length of the shortest keyword in a set for the CW-WBM algorithm.

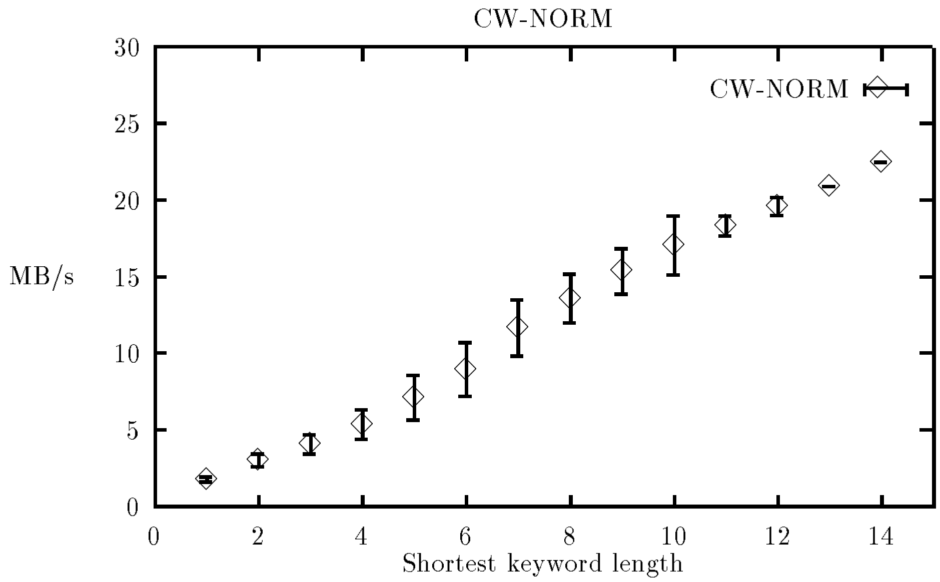


Figure 13.13: Performance variation (in megabytes/second) versus the length of the shortest keyword in a set for the CW-NORM algorithm.

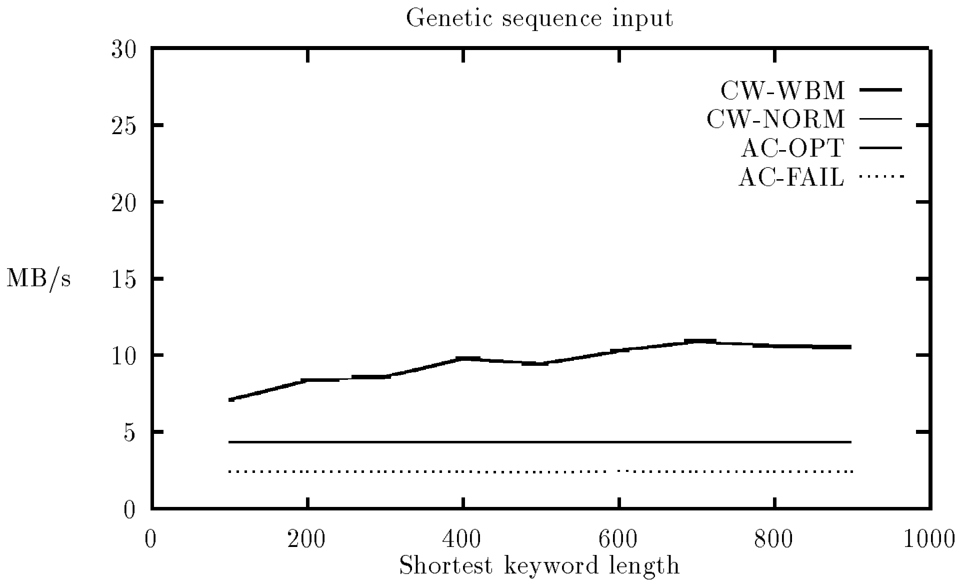


Figure 13.14: Algorithm performance (in megabytes/second) versus the length of the keywords in a given set, for the DNA test data. The performance of the CW-NORM and CW-WBM algorithms are almost coincidental, shown as the ascending solid line.

the AC-OPT algorithm was independent of the keyword length. Unlike on the natural language input, the AC-FAIL algorithm displayed no noticeable improvement with increasing keyword length; the performance of the AC-FAIL algorithm was little more than half of the performance of the AC-OPT algorithm.

As in the natural language tests, the performance of the CW algorithms improved with increasing keyword length. The rate of performance increase was considerably less than on the natural language input (see Figure 13.8). On the DNA input, the CW algorithms displayed median performance at least twice that of the AC-OPT algorithm.

13.4.3 Single-keywords

For the single-keyword tests, the average performance (of each algorithm) is graphed against the length of the keyword and superimposed in Figure 13.15.

The KMP, AC-FAIL, and AC-OPT algorithms displayed performance that was largely independent of the keyword length. The AC-OPT algorithm outperformed the other two, while the KMP algorithm displayed the worst performance. Although the KMP algorithm is similar in structure to the AC-FAIL algorithm, the heavy use of indexing (as opposed to the use of pointers in AC-FAIL) in the KMP algorithm degrades its performance. (The use of indexing makes the KMP algorithm more space efficient than the AC algorithms.) The performance of the CW algorithms improved almost linearly with the length of the keyword, with the CW-NORM algorithm outperforming the CW-WBM algorithm.

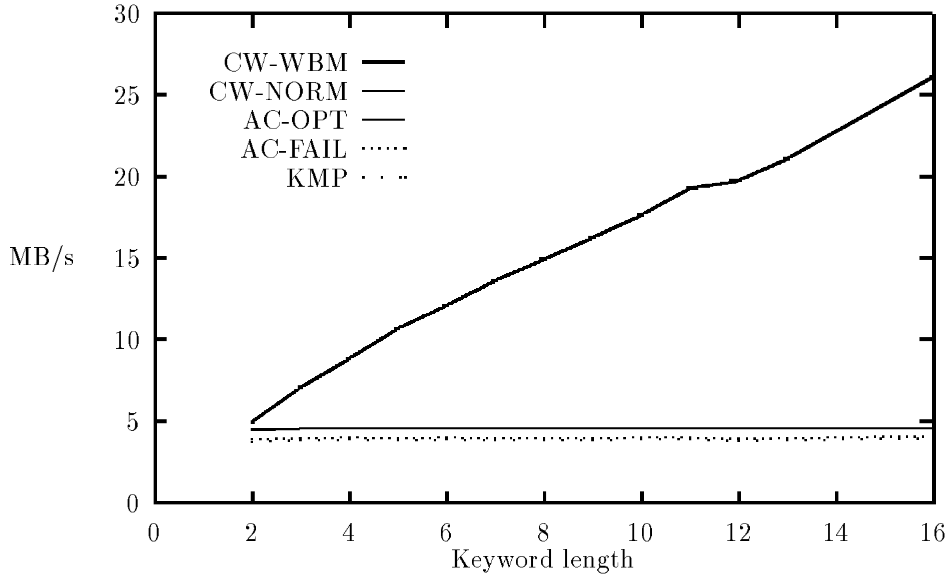


Figure 13.15: Algorithm performance (in megabytes/second) versus the length of the (single) keyword. The performance of the KMP and AC-FAIL algorithms are shown as the coincidental dotted horizontal line, while those of the CW-WBM and CW-NORM algorithms are shown as the coincidental ascending solid line.

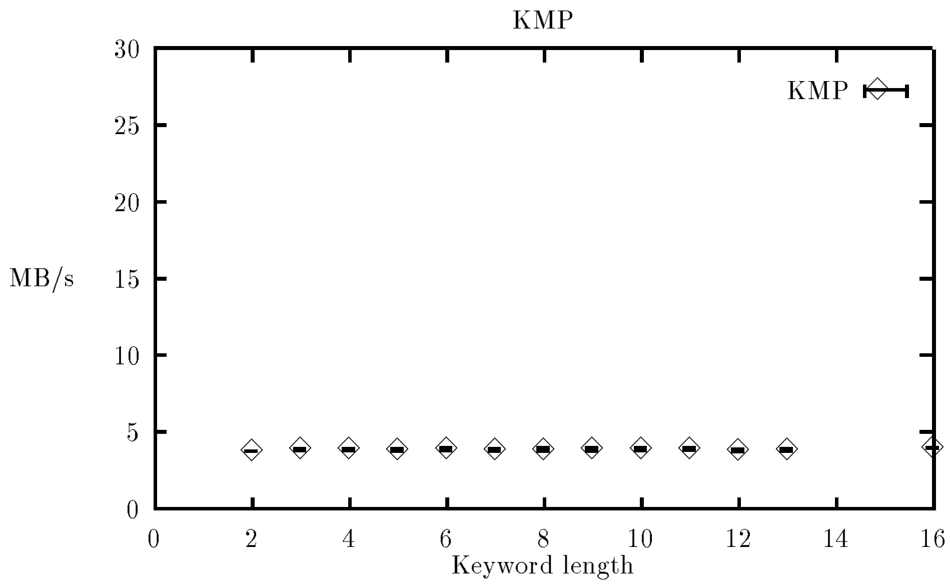


Figure 13.16: Performance variation (in megabytes/second) versus the (single) keyword length for the KMP-FAIL algorithm.

The variance of the performance of the KMP and the AC-FAIL algorithms was minor, as shown in Figures 13.16 and 13.17 (respectively). The AC-OPT algorithm displayed no noticeable variance over the entire range of keyword lengths, as is shown in Figure 13.18. The CW algorithms showed some variance (increasing with longer keyword lengths) as shown in Figures 13.19 and 13.20 respectively.

The performance of the algorithms on the single keyword test data is in agreement with the data collected by Hume and Sunday [HS91].

13.5 Conclusions and recommendations

The conclusions of this chapter fall into two categories: general conclusions regarding the algorithms and testing them, and conclusions relating to the performance of specific algorithms. The general conclusions are:

- The relative performance of the algorithms did not vary across the testing platforms (the DEC Alpha, HP Snake, and Sun Sparc Station 1+ workstations).

- Testing the algorithms on two vastly differing types of input (English text and DNA sequences) indicates that varying such factors as alphabet size, keyword set size, and smallest keyword length can produce very different rates of performance increase or decrease.

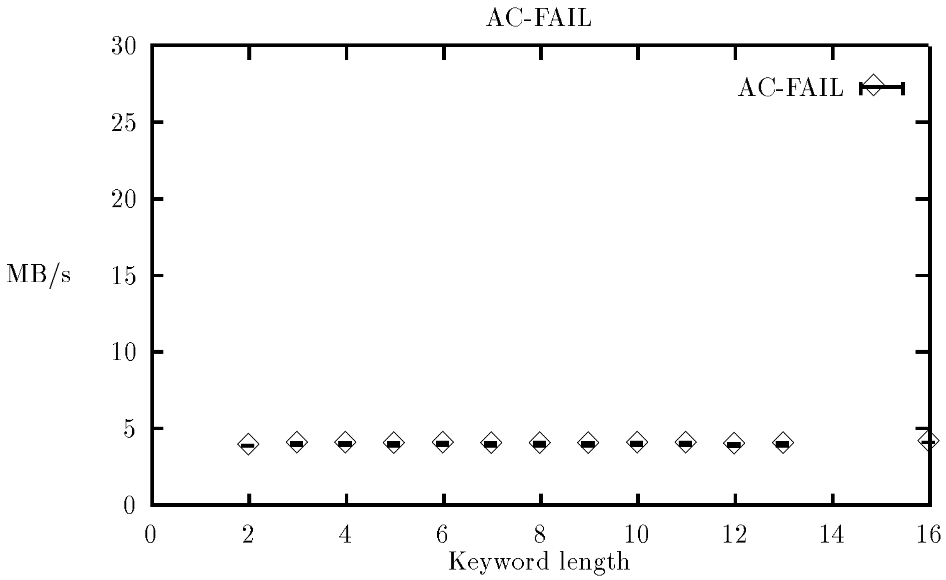


Figure 13.17: Performance variation (in megabytes/second) versus the (single) keyword length for the AC-FAIL algorithm.

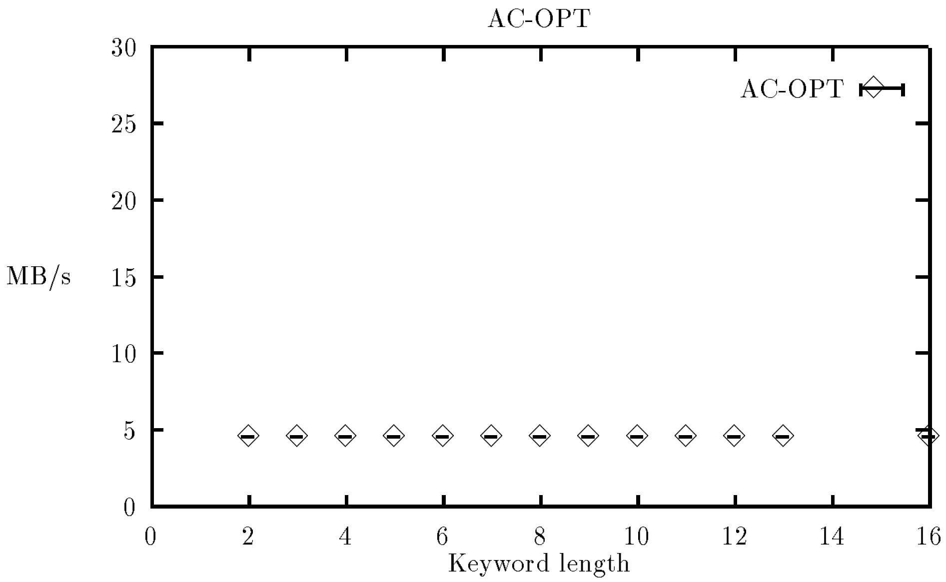


Figure 13.18: Performance variation (in megabytes/second) versus the (single) keyword length for the AC-OPT algorithm.

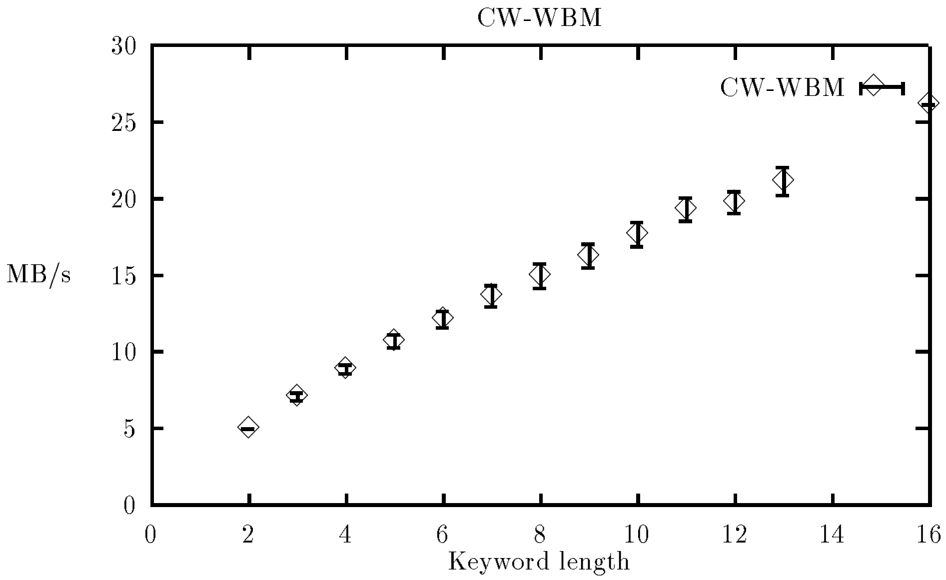


Figure 13.19: Performance variation (in megabytes/second) versus the (single) keyword length for the CW-WBM algorithm.

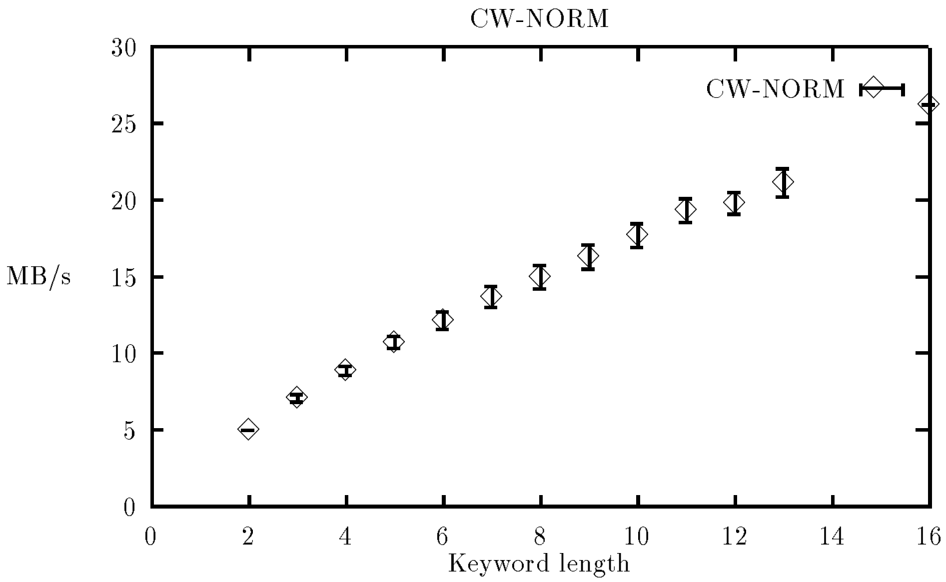


Figure 13.20: Performance variation (in megabytes/second) versus the (single) keyword length for the CW-NORM algorithm.

- Comparing algorithm performance to keyword set size and shortest keyword length proved to be more useful (in selecting an algorithm) than comparing performance to other statistics.

The specific performance conclusions are:

- The performance of the AC-OPT algorithm was independent of the keyword sets. The AC-FAIL and KMP algorithm performance increased slightly with increasing shortest keyword length and decreased with increasing keyword set size.

- The performance of the CW algorithms improved approximately linearly with increasing length of the shortest keyword in the keyword set. The rate of increase was much greater with natural language input than with DNA input. The performance of the CW algorithms declined sharply with increasing keyword set size.

- For a given keyword set size and shortest keyword length, the AC and KMP algorithms displayed little or no variance (in performance). The CW algorithms displayed slightly more variance in performance.

- As predicted in Section 4.4, the CW-NORM algorithm outperforms the CW-WBM algorithm. Under certains conditions, the difference in performance can be substantial. (Furthermore, the cost of precomputation for the two algorithms is approximately the same.)

- The AC-OPT algorithm always outperforms the AC-FAIL algorithm; both require similar precomputation, but the AC-FAIL data structures can be made more space efficient.

- On the single-keyword tests:

  - The single-keyword test results were consistent with those presented by Hume and Sunday [HS91].
  - The AC-FAIL algorithm always outperforms the KMP algorithm.
  - In most cases, the CW algorithms outperform the AC algorithms.

- In [Aho90, p. 281], A.V. Aho states that

  > In practice, with small numbers of keywords, the Boyer-Moore aspects of the Commentz-Walter algorithm can make it faster than the Aho-Corasick algorithm, but with larger numbers of keywords the Aho-Corasick algorithm has a slight edge.

  Although Aho's statement is correct, with the performance data presented in this report, we are able to state more precisely the conditions under which the Commentz-Walter algorithms outperform the Aho-Corasick algorithms.

- On the multiple-keyword natural language tests, the CW algorithms outperformed the AC algorithms when the length of the shortest keyword was long (in general, at least four symbols) and the keyword set size was small (in general, fewer than thirteen keywords). The performance difference between the CW algorithms and the AC algorithms was frequently substantial.

- On the DNA tests, the CW algorithms substantially outperformed the AC algorithms. On these tests the keyword length was at least 100 and the number of keywords in the set was no more than 10. The DNA results show that the CW algorithms can yield much higher performance than the often-used AC-OPT algorithm in areas such as genetic sequence pattern matching.

For applications involving small alphabets and long keywords (such as DNA pattern matching), the performance of the CW-NORM algorithm makes it the algorithm of choice. Only when the keyword set size is much larger than ten keywords should the AC-OPT algorithm be considered.

The following procedure can be used to choose a pattern matching algorithm for a natural language pattern matching application:

if performance independent of keyword set is required **then**
  AC-OPT
else
  if multiple-keyword sets are used **then**
    if fewer than thirteen keywords and the shortest keyword length is at least four **then**
      CW-NORM
    else
      choose an AC algorithm
    fi
  else (single-keyword sets)
    if space is severely constrained **then**
      KMP
    else
      if the keyword length is at least two **then**
        CW-NORM
      else
        choose an AC algorithm
      fi
    fi
  fi
fi

An AC algorithm can be chosen as follows:

if space efficiency is needed **then**
  AC-FAIL

```
else
  AC-OPT
fi
```

Chapter 14

The performance of *FA* construction algorithms

This chapter presents performance data on a number of *FA* (and *DFA*) construction algorithms. The time required to construct an *FA* was measured (for each of the constructions) as well as the time required for a single transition (for each of the types of *FA*s). The implementations given in FIRE Lite and the FIRE Engine were used (see Chapter 10). Additionally, we present recommendations for selecting a construction. The algorithms discussed here are a selection of the ones derived in the taxonomy in Chapter 6.

14.1 Introduction

Most of what is known about the relative performance of the automata construction algorithms is anecdotal. As with the minimization algorithms, most software engineers choose a construction algorithm that is simple or easy to understand. Such choices should, however, be based upon performance data about the algorithms.

In this chapter, we present performance data on eight of the most efficient and easiest to implement constructions.

This chapter is structured as follows:

- In Section 14.2, we list the algorithms used in collecting the performance data.
- The testing methodology used in benchmarking the algorithms is described in Section 14.3.
- The results of the benchmarking are presented in Section 14.4.
- Lastly, the conclusions and recommendations of this chapter are given in Section 14.5.

14.2 The algorithms

The algorithms tested were derived in Chapter 6. They have also been implemented in FIRE Lite. The implementations are discussed in detail in Chapter 10. For convenience, we will put the algorithms in two groups: those producing an *FA*, and those producing a *DFA*. The *FA* constructions are:

- The canonical construction (TH, since it is a variant of Thompson's construction), appearing as Construction 6.15 in this dissertation.
- The Berry-Sethi construction (BS), given here as Construction 6.39 (REM-ε, SYM, A-S).
- The dual of the Berry-Sethi construction (BS-D), appearing as Construction 6.65 (REM-ε-DUAL, SYM, A-S).

The *DFA* constructions are:

- The Aho-Sethi-Ullman construction (ASU) — Construction 6.69 (REM-ε-DUAL, SYM, A-S, E-MARK, SUBSET, USE-S).
- The deterministic item set construction (IC) — Construction (REM-ε, SUBSET, USE-S) on page 156.
- DeRemer's construction (DER) — Construction (REM-ε, SUBSET, USE-S, XFILT) on page 159.
- The filtered item set construction (FIC) — Construction (REM-ε, SUBSET, USE-S, WFILT) on page 158.
- The McNaughton-Yamada-Glushkov construction (MYG), given as Construction 6.44 (REM-ε, SYM, A-S, SUBSET, USE-S).

For specific information on these algorithms, see Chapter 6.

Noticeably absent from this list are the derivatives-based algorithms (Brzozowski's and Antimirov's algorithms). The derivatives in these algorithms are the states. In their pure forms, the derivatives are stored as regular expressions. The space and time required to store and manipulate the regular expressions proved to be extremely costly, when compared to the representations of states used in some of the other constructions. The derivative-based algorithms consistently performed 5 to 10 times slower than the next slowest algorithm (the IC algorithm, in particular). Indeed, the preliminary testing could only be done for the smallest regular expressions without making use of virtual memory (which would further degrade their performance). No doubt the use of clever coding tricks would improve these algorithms greatly — though such coding tricks would yield a new algorithm.

14.3 Testing methodology

This section gives an overview of the methods used in gathering the test data. We begin with the details of the test environment, followed by the details of the methods used to generate the regular expressions for input to the algorithms.

14.3.1 Test environment

All of the tests were performed on an IBM-compatible personal computer running MS-DOS. The machine has an INTEL PENTIUM processor with a 75 Mhz clock, an off-chip cache of 256 kilobytes and main memory of 8 megabytes. During all of the tests, no other programs which could consume processing power were installed.

The test programs were compiled with the WATCOM C++32 compiler (version 9.5a) with optimizations for speed. The WATCOM compiler is bundled with an MS-DOS extender (used to provide virtual memory for applications with large data-structures) known as DOS/4GW. Since the use of virtual memory could affect the performance data, all data-structures were made to fit in physical memory.

Timing was done by reprogramming the computer's built-in counter to count microseconds. This reprogramming was encapsulated within a C++ timer class which provided functionality such as starting and stopping the timer. Member functions of the class also subtracted the overhead of the reprogramming from any particular timing run.

14.3.2 Generating regular expressions

A large number of regular expressions were randomly generated. As in Chapter 13, we used the random number generator appearing in [PTVF92, p. 280]. The regular expressions were generated as follows:

1. A height in the range $[2, 5]$ was randomly chosen for the parse tree of the regular expression. (Larger heights were not chosen for memory reasons; smaller heights were not chosen since the constructions were performing close to the clock resolution.)

2. A regular expression of the desired height was generated, choosing between all of the eligible operators[1] with equal probability.

3. For the leaves, $\emptyset$ and ε nodes were never chosen. The $\emptyset$ was omitted, since such regular expressions prove to be uninteresting (they simply denote the empty language). Similarly, the ε was omitted, since the same effect is obtained by generating ? nodes.

The following table shows the number of nodes in an *RE* and the number of *RE*s with that number of nodes.

[1]Some operators will not be eligible. For example, to generate an *RE* of height 3, only the unary or binary operators can appear at the root.

Number of nodes	*Number of* RE*s*
2	500
4	84
5	118
6	59
7	246
8	346
9	279
10	413
11	251
12	264
13	227
14	196
15	131
16	73
17	73
18	75
19	45
20	40
21	17
22	12
23	6
24	5
25	1
26	1

There are a total of 3462 *RE*s; the mean size is 9.63 nodes and the standard deviation is 4.72. The distribution of the number of nodes reflects the way in which the regular expressions were generated. Note that there are no regular expressions with a single node or with three nodes. These were omitted since the time to construct an *FA* from such small *RE*s was usually below the resolution of the timer. (Two node regular expressions were used since they contain a $*$ or a $+$ node at the root. All of the constructions require more time to construct an automaton corresponding to such an expression.) Furthermore, it is not possible to generate lengthy strings in the language of such regular expressions.

In the following table, we give the number of symbol nodes in an *RE* and the number of *RE*s with that number of nodes.

Number of symbol nodes	*Number of* RE*s*
1	500
2	247
3	657
4	773
5	553
6	367
7	181
8	114
9	45
10	17
11	6
12	2

The mean number of symbol nodes is 4.00 nodes and the standard deviation is 1.99.

Other statistics on the regular expressions were also collected, such as the height of the *RE*s, the star-height of the *RE*s, and some measure of the inherent nondeterminism in the *RE*s[2]. These statistics did not prove to be useful in considering the performance of the algorithms.

14.3.3 Generating input strings

For each type of automaton (*FA*, *ε-free FA*, and *DFA*), we also present data on the time required to make a single transition. In order to measure this, for each *RE* used as input to the constructions we generate a string in the prefix of the language denoted by the *RE*. Strings of length up to 10000 symbols were generated.

The constructed automaton processes the string, making transitions, while the timer is used to measure the elapsed time. The time was divided by the number of symbols processed, yielding the average time for a single transition.

14.4 Results

The performance data will be presented in three sections. First, we present the time required to construct an automaton. Next, we present the size of the constructed automaton. Lastly, we consider the time required to make a single transition.

14.4.1 Construction times

For each of the generated regular expressions and each of the constructions, we measured the number of microseconds to construct the automaton. For many applications, the

[2]One estimate of such nondeterminism is the ratio of alternation (union) nodes and $*$ or $+$ nodes to the total number of nodes in the regular expression.

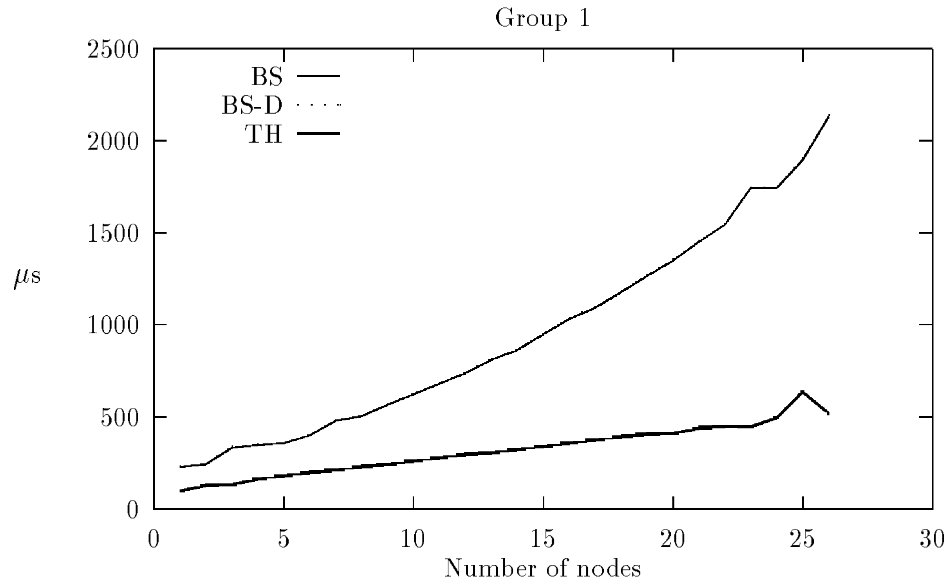


Figure 14.1: Median construction times for *FA* constructions graphed against the number of nodes in the regular expressions. Note that BS and BS-D are superimposed as the higher ascending line.

construction time can be the single biggest factor in determining which construction to use.

First, the constructions are divided into two groups: those producing an *FA* (not a *DFA*), and those producing a *DFA*. The median performance of these two groups of constructions was graphed against the number of nodes in the regular expressions in Figures 14.1 and 14.2 respectively. All three of the algorithms in the first group are predicted to perform linearly in the number of nodes in the input regular expression. In Figure 14.1, the performance of the BS and BS-D constructions was nearly identical, as could be predicted from their duality relationship given in Chapter 6. They both performed somewhat worse than the TH algorithm. The apparent jump in construction time (of the TH algorithm) for 25 node regular expressions is due to the fact that only a single such expression was generated. Had more regular expressions been generated, the median performance would have appeared as a straight line (following the linear performance predicted for the TH algorithm).

The scale on Figure 14.2 shows that the second group of constructions were much slower than the first group. The ASU construction was by far the fastest, with FIC and MYG being the middle performers, and DER and IC being the slowest. In the range of 20 to 26 nodes, all of the constructions displayed peaks in their construction times. In these cases, some of the generated regular expressions have corresponding *DFA*s which are exponentially larger — forcing all of the constructions to take longer.

We now consider the performance of the same two groups of constructions, graphed

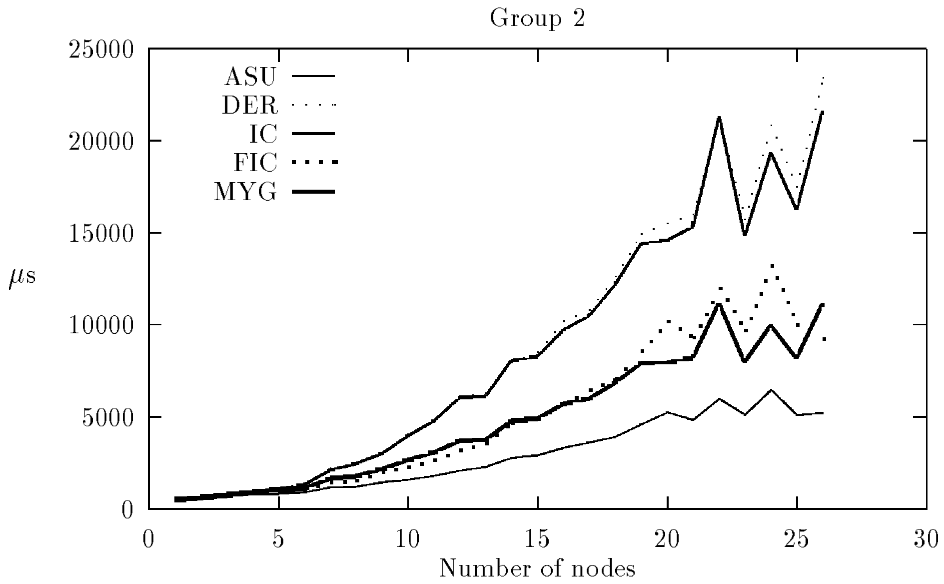


Figure 14.2: Median construction times for *DFA* constructions graphed against the number of nodes in the regular expressions. The lowest line is ASU performance, while the middle pair of lines are MYG and FIC; the highest pair of lines is DER and IC.

against the number of symbol nodes in the regular expressions. The graphs appear in Figures 14.3 and 14.4 respectively. These two graphs present similar information to Figures 14.1 and 14.2.

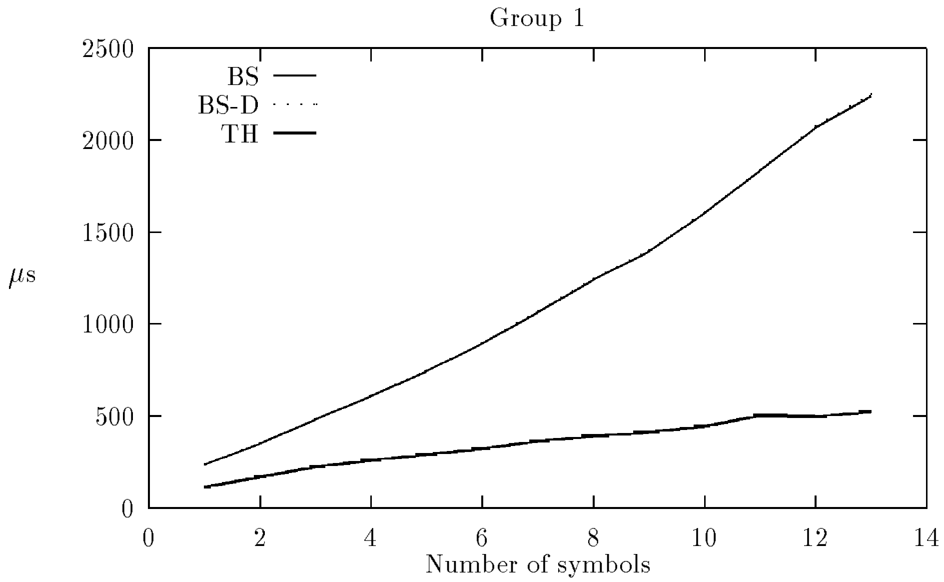


Figure 14.3: Median construction times for *FA* constructions graphed against the number of symbol nodes in the regular expressions. The BS and BS-D performance is identical, graphed as the higher line.

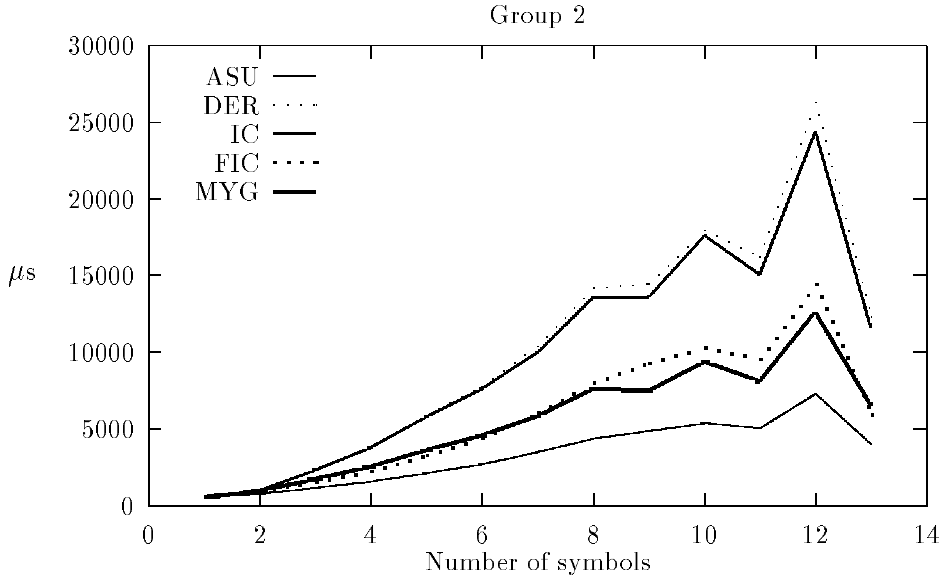


Figure 14.4: Median construction times for *DFA* constructions graphed against the number of symbol nodes in the regular expressions. The best performance was delivered by ASU, followed by FIC and MYG and lastly by DER and IC.

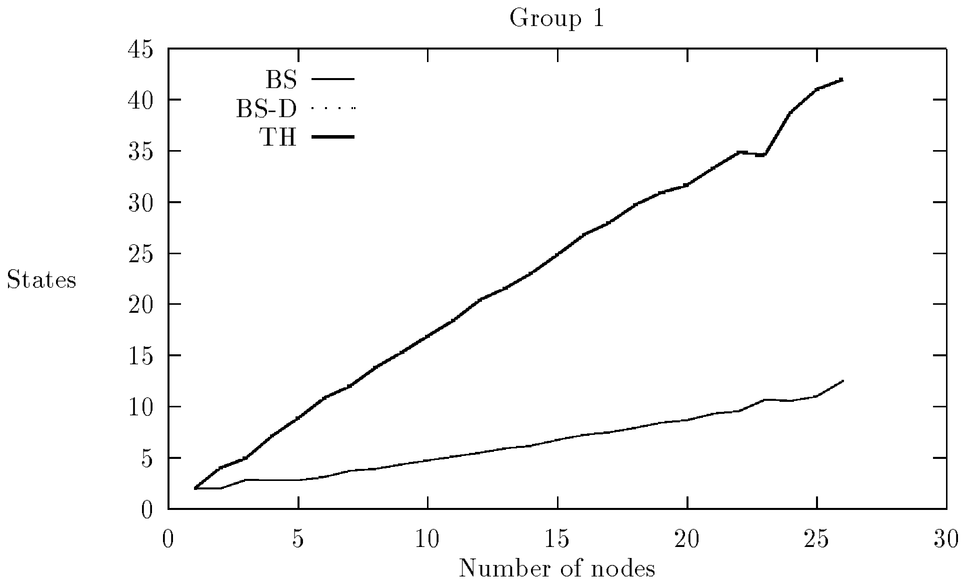


Figure 14.5: The number of states in the *FA* is graphed against the number of nodes in the regular expression used as input for the TH, BS, and BS-D constructions. Note that the BS and BS-D constructions produce automata of identical size.

14.4.2 Constructed automaton sizes

The size of each of the constructed automata was measured. The amount of memory space consumed by an automaton is directly proportional to the number of states in the automaton, and this data can be used to choose a construction based upon some memory constraints. Since the exact amount of memory (in bytes) consumed depends heavily on the compiler being used, we present the data in state terms.

Again, we group the non-*DFA* producing constructions and the *DFA* constructions. Figures 14.5 and 14.6 give the automata sizes versus number of nodes in the regular expressions, for the two groups of constructions. The former figure shows that the size of the TH-generated automata can grow quite rapidly. The BS and BS-D constructions produce automata of identical size (as can be seen from their derivations in Chapter 6). In the second figure (Figure 14.6), we can identify two interesting properties of the constructions: ASU and FIC produce automata of the same size, as do the pair IC and MYG. Given the superior performance of ASU (over FIC), there is little reason to make use of FIC. Similarly, the MYG construction out-performs IC, and there is no reason to use IC since the automata will be the same size.

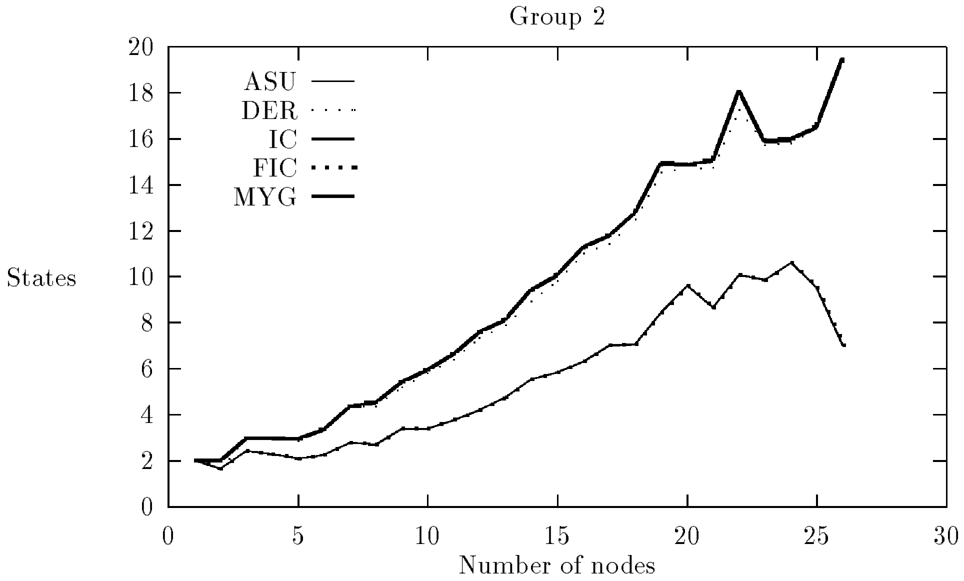


Figure 14.6: The number of states in the *DFA* is graphed against the number of nodes in the regular expression used as input for the ASU, DER, FIC, IC, and MYG constructions. Note that the FIC and ASU constructions produce *DFA*s of identical size (the superimposed lower line), as did the pair IC and MYG (the superimposed higher line).

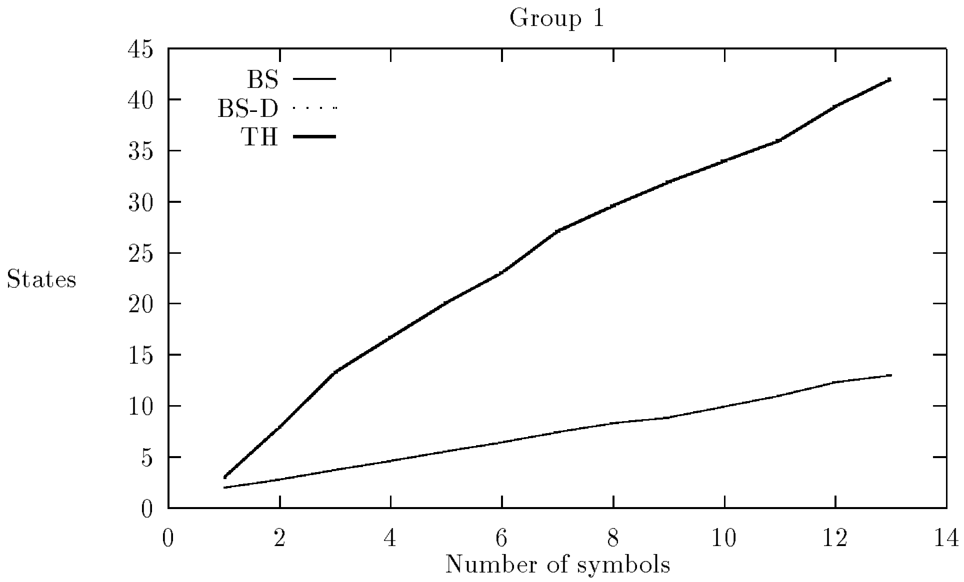


Figure 14.7: The number of states in the *FA* is graphed against the number of symbol nodes in the regular expression used as input for the TH, BS, and BS-D constructions. Note that the BS and BS-D constructions produce automata of identical size (the lower ascending line).

Figures 14.7 and 14.8 give the automata sizes versus number of symbol nodes in the regular expressions, for the two groups of constructions. These two graphs provide similar information to that given in Figures 14.5 and 14.6.

14.4.3 Single transition performance

The time required to make a single transition was measured for *FA*s (using TH), ε*-free FA*s (using BS), and *DFA*s (using FIC). The median time for the transitions has been graphed against the number of states in Figure 14.9. (Note that, for a given number of states, the number of each of the different types of automata varied.) The more general *FA*s displayed the slowest transition times, since the current set of states is stored as a set of integers, and the ε-transition and symbol transition relations are stored in a general manner. The ε*-free FA*s displayed much better performance, largely due to the time required to compute ε-transition closure in an automaton with ε-transitions. With the simple array lookup mechanism used in *DFA*s, it is not surprising that their transitions are by far the fastest and are largely independent of the number of states in the *DFA*.

The individual performance data for *FA*s, ε*-free FA*s, and *DFA*s are shown in Figures 14.10, 14.11, and 14.12. The variance for general *FA*s and ε*-free FA*s is quite large. In both cases, the time for a single transition depends upon the number of states in the current state set. The variance for a *DFA* transition appears to be quite large. This is

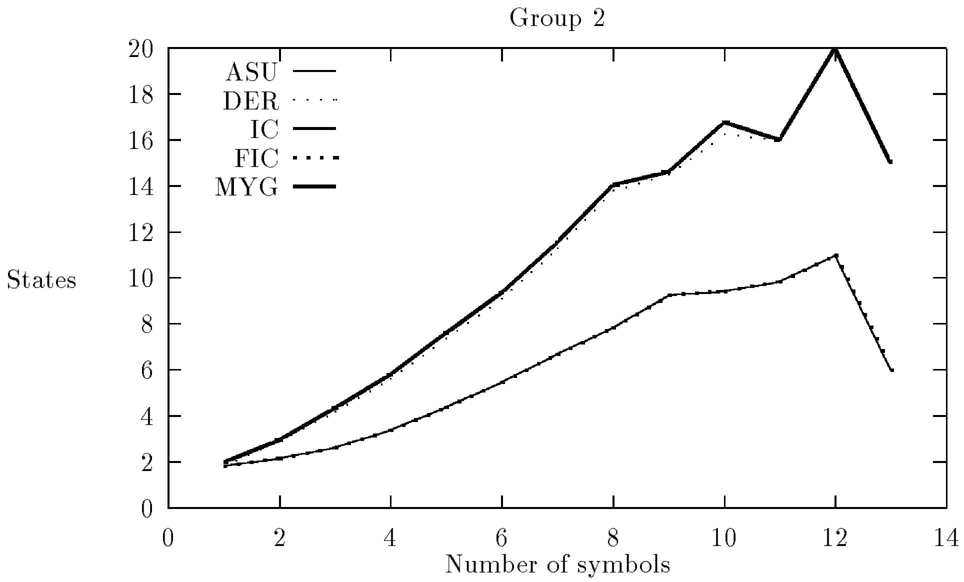


Figure 14.8: The number of states in the *DFA* is graphed against the number of symbol nodes in the regular expression used as input for the ASU, DER, FIC, IC, and MYG constructions. Note that the ASU and FIC constructions produce *DFA*s of identical size, as do the MYG and IC pair.

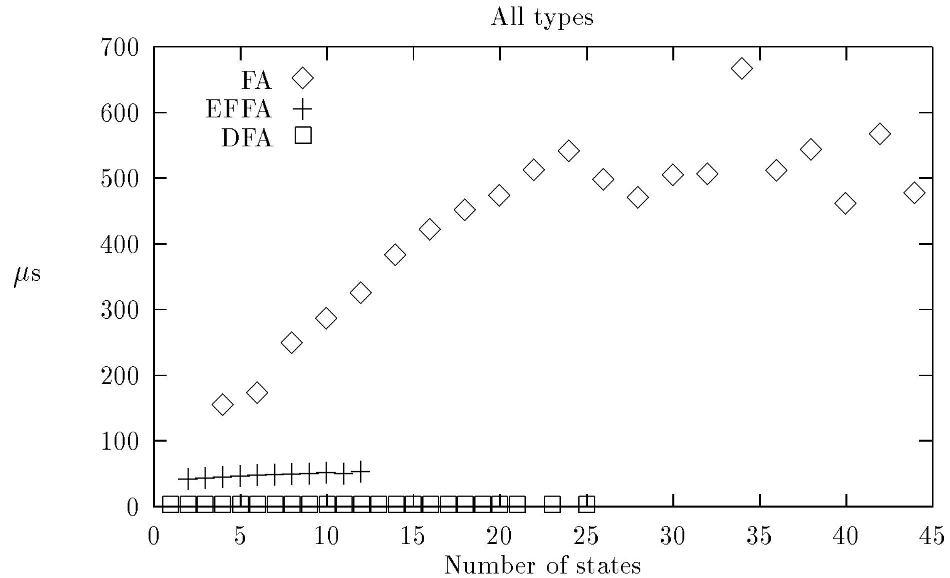


Figure 14.9: Median time to make a single transition in microseconds (μs), for *FA*s, ε*-free FA*s, and *DFA*s, versus the number of states.

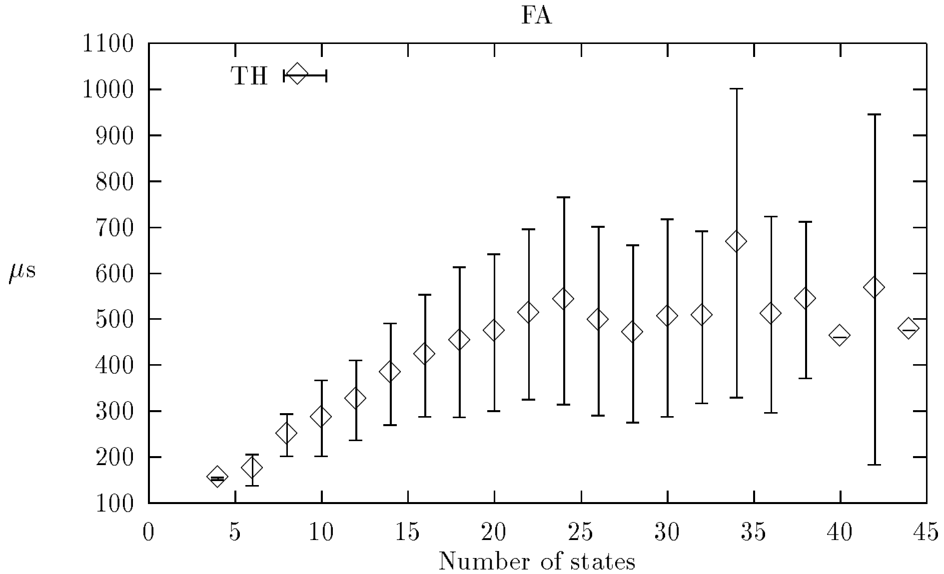


Figure 14.10: Performance variance for a single transition in microseconds (μs) in an *FA*, versus the number of states. Also included are $+1$ and -1 standard deviation bars.

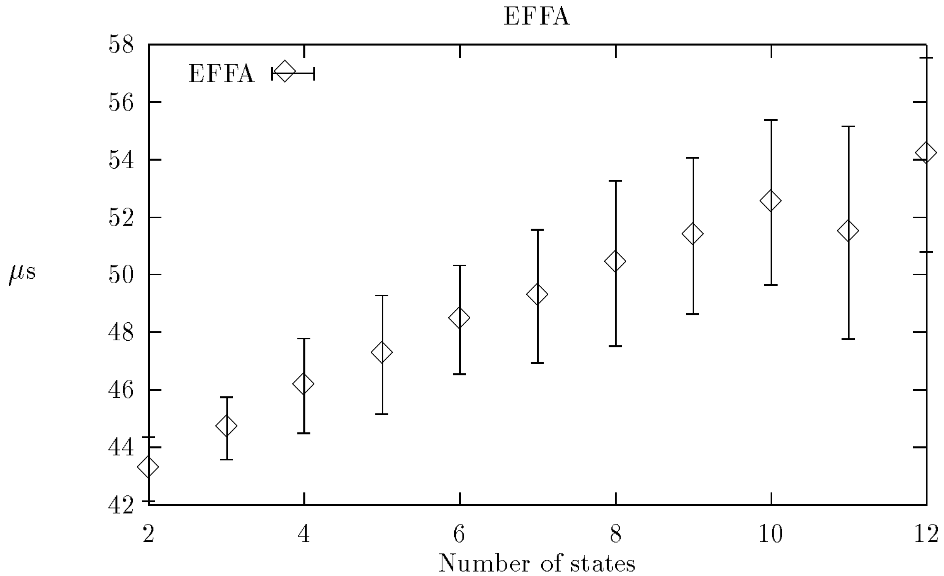


Figure 14.11: Performance variance for a single transition in microseconds (μs) in a *ε-free FA*, versus the number of states. Also included are $+1$ and -1 standard deviation bars.

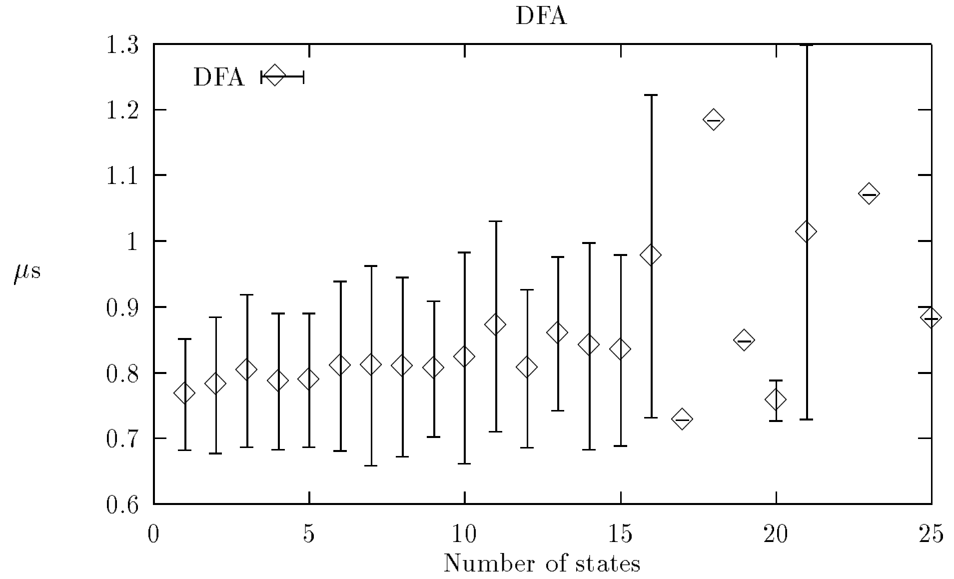


Figure 14.12: Performance variance for a single transition in microseconds (μs) in a *DFA*, versus the number of states. Also included are +1 and −1 standard deviation bars.

largely due to the fact that single transitions are around the resolution of the timer, meaning that other factors play a role. Such factors include artificial ones such as clock jitter and real ones such as instruction cache misses.

14.5 Conclusions and recommendations

The conclusions of this chapter are:

- The fact that some constructions derived in Chapter 6 are encodings of one another (and some others are duals of one another) can be seen in the data.
  - The BS and BS-D algorithms (they are duals of one another) have the same performance characteristics.
  - The ASU and FIC algorithms construct automata of the same size. This can been predicted by comparing the algorithm for Construction (REM-ε, SUBSET, USE-S, WFILT) on page 185 with Algorithm 6.86.
  - Interestingly, IC and MYG produce automata of the same size. Indeed, the resulting *DFA*s will always be isomorphic since their underlying *FA* constructions (Constructions 6.19 and 6.39, respectively) are simply encodings of one another.
- The advantages of the 'filters' introduced in Section 6.4.1 can be seen in the performance of IC, which is worse than either DER or FIC. Furthermore, IC produces

larger automata than either DER or FIC.

- As was shown in Chapter 6, the BS and BS-D constructions produce automata with sizes linear in the number of symbol nodes in the input regular expression.
- Predictably, the subset construction (with start-unreachable state removal) is a costly operation. All of the *DFA* construction algorithms were significantly slower than the general *FA* constructions.
- ASU is the subset construction composed with BS-D, while MYG is the subset construction composed with BS. Given the identical performance of BS and BS-D, it is interesting to note that ASU was significantly faster than MYG. Furthermore, ASU produced smaller *DFA*s than MYG.
- The time to make a transition can vary widely for different types of automata. As such, it can be an important factor in choosing a type of automaton.
  - Both *FA*s with ε-transitions and ε*-free FA*s have transition times that depend upon the number of states in the automaton, however, *FA*s with ε-transitions have significantly longer transition times.
  - *DFA*s have transition times which are largely independent of the size of the automaton[3]. These times were around the resolution of the clock[4].

The following procedure can be used to choose a finite automata construction algorithm:

Choices := {TH, BS, BS-D, ASU};
if construction time is important **then**
  Choices := *Choices* $\cap$ {TH, BS, BS-D}
fi;
if automaton size is important **then**
  Choices := *Choices* $\cap$ {BS, BS-D, ASU}
fi;
if transition time is important **then**
  Choices := *Choices* $\cap$ {BS, BS-D, ASU}
fi

[3]Although, inspecting the implementation reveals that very dense transition graphs will yield more costly transitions than a sparse transition graph.

[4]This does not invalidate the results, since the average transition time is taken over a large number of transitions.

Chapter 15

The performance of *DFA* minimization algorithms

In this chapter, we present data on the performance of five *DFA* minimization algorithms implemented in FIRE Lite (see Chapter 11). The algorithms tested were derived in the taxonomy of *DFA* minimization algorithms in Chapter 7, and are implemented in FIRE Lite.

15.1 Introduction

Very little is known about the performance of *DFA* minimization algorithms in practice. Most software engineers choose an algorithm by reading their favourite text-book, or by attempting to understand Hopcroft's algorithm — the best known algorithm, with $\mathcal{O}(n \log n)$ running time. The data in this chapter will show that somewhat more is involved in choosing the right minimization algorithm. In particular, the algorithms appearing in a popular formal languages text-book [HU79] and in a compiler text-book [ASU86] have relatively poor performance (for the chosen input data). Two of the algorithms which could be expected to have poor performance[1] actually gave impressive results in practice. Recommendations for software engineers will be given in the conclusions of this chapter.

In short, this chapter is structured as follows:

- Section 15.2 gives an outline of the five algorithms tested.
- Section 15.3 explains the methodology used in gathering the test data.
- The performance data for the five algorithms are presented in Section 15.4.
- The conclusions and recommendations of this chapter are given in Section 15.5.

[1] They are Brzozowski's algorithm (which uses the costly subset construction) and a new algorithm which has exponential worst-case running time.

15.2 The algorithms

The five algorithms are derived in Chapter 7 and their corresponding implementations are described in detail in Chapter 11. Here, we give a brief summary of each of the algorithms (and an acronym which will be used in this chapter to refer to the algorithm):

- The Brzozowski algorithm (BRZ), derived in Section 7.2. It is unique in being able to process a *FA* (not necessarily a *DFA*), yielding a minimal *DFA* which accepts the same language as the original *FA*.

- The Aho-Sethi-Ullman algorithm (ASU), appearing as Algorithm 7.21. It computes an equivalence relation on states which indicate which states are indistinguishable (see Chapter 7). The appearance of this algorithm in [ASU86] has made it one of the most popular algorithms among implementors.

- The Hopcroft-Ullman algorithm (HU), appearing as Algorithm 7.24. It computes the complement of the relation computed by the Aho-Sethi-Ullman algorithm. Since this algorithm traverses transitions in the *DFA* in their reverse direction, it is at a speed disadvantage in most *DFA* implementations (including the one used in FIRE Lite and the FIRE Engine).

- The Hopcroft algorithm (HOP), presented as Algorithm 7.26. This is the best known algorithm (in terms of running time analysis) with running time of $\mathcal{O}(n \log n)$ (where n is the number of states in the *DFA*).

- The new algorithm (BW) appearing as Algorithm 7.28 in the taxonomy. It computes the equivalence relation (on states) from below (with respect to the refinement ordering). The practical importance of this is explained in Section 7.4.7.

These algorithms are the only ones implemented in FIRE Lite.

15.3 Testing methodology

This section gives an overview of the methods used in gathering the test data. The test environment is identical to that used in collecting the *FA* construction performance data (Chapter 14). However, we consider the methods used to generate the *DFA*s for input to the algorithms.

One caveat about the distribution of the test data used in collecting the benchmarks is in order. In practice, *DFA*s are usually obtained from one of two sources: they are constructed from regular expressions[2], or they are generated from some other specification[3]. In the first case, the *DFA*s have certain characteristics: they are usually not very large, they have relatively sparse transition graphs, and the alphabet frequently consists of the entire

[2]This is common in pattern matching and compiler lexical analysis applications of *DFA*s.

[3]This is common in modeling digital circuits.

ASCII character set. In the second case, the *DFA*s can potentially consist of thousands of states, with dense transition graphs and relatively small (even binary or ternary) alphabets. The FIRE Lite is best structured for obtaining *DFA*s from regular expressions — as would be done in tools such as `lex` or `grep` — and so we only consider the minimization of the first group of *DFA*s. Due to the memory limitations of the test environment and some space inefficiencies in FIRE Lite, we only consider the minimization of relatively small *DFA*s (fewer than 25 states). Although it is possible to extrapolate the performance data, the performance of the algorithms on much larger *DFA*s is difficult to forecast.

Random regular expressions were generated (using FIRE Lite and the techniques outlined in Chapter 14). The *DFA*s were constructed from the regular expressions, using the 'item set' construction — Construction (REM-ε, SUBSET, USE-S) appearing on page 156. Some data on the constructed *DFA*s is as follows:

Number of states	*Number of* DFA*s*
2	1585
3	395
4	541
5	490
6	454
7	303
8	266
9	232
10	150
11	104
12	85
13	61
14	37
15	25
16	27
17	20
18	15
19	10
20	9
21	8
22	9
23	7

There are a total of 4833 *DFA*s; the mean size is 5.24 states and the standard deviation is 3.65. Clearly, the *DFA* sizes are not evenly distributed. The size distribution of the *DFA*s results from the distribution of the randomly generated *RE*s.

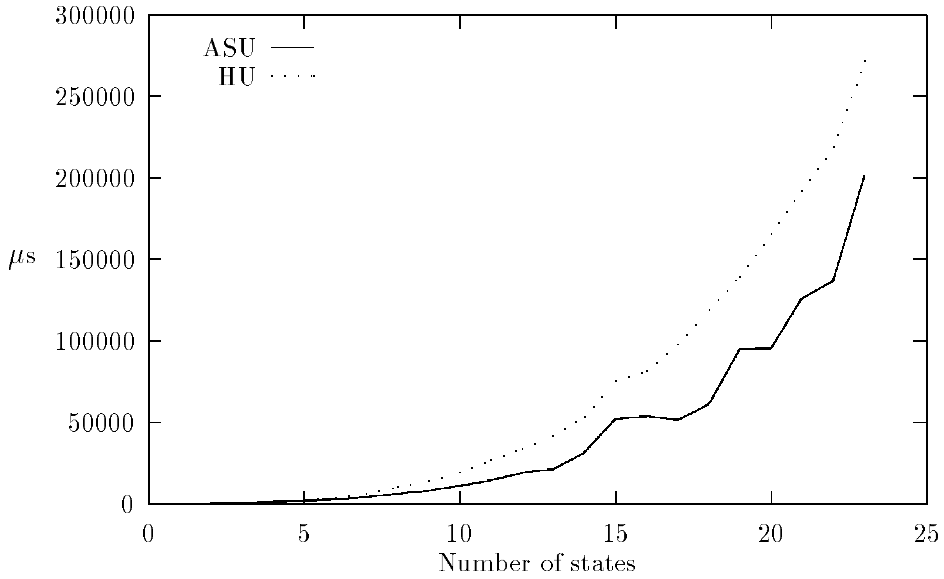


Figure 15.1: Median performance (in microseconds to minimize) versus *DFA* size.

15.4 Results

For each of the random *DFA*s and each of the five algorithms, we measured the time in microseconds (μs) required to construct the equivalent (accepting the same language) minimal *DFA*.

The performance of the algorithms was graphed against the number of states in the original *DFA*. Graphing the performance against the number of edges in the original *DFA* was not found to be useful in evaluating the performance of the algorithms. The algorithms can be placed in two groups, based upon their performance. In order to aid in the comparison of the algorithms, we present graphs for these two groups separately.

The first group (ASU and HU) are the slowest algorithms; the graph appears in Figure 15.1. The HU algorithm was the worst performer of the five algorithms. It traverses the transitions (in the input *DFA*) in the reverse direction. A typical implementation of a *DFA* does not favour this direction of traversal. The ASU algorithm performed slightly better, although its performance is also far slower than any of BRZ, HOP, or BW. The second group (BRZ, HOP, and BW) are significantly faster; the corresponding graph appears in Figure 15.2. Note that this graph uses a different scale from the one in Figure 15.1. The data point (for 17 states) for the BW algorithm was dropped, since (at that point) the algorithm was more than 30 times slower than any of the other algorithms in this group. The complete (unedited, including the 17 state data point) data for the BW algorithm is presented in Figure 15.8.

We now turn to the performance of the individual algorithms. (The graphs to be presented use different scales for the y axis. For direct comparison between the graphs,

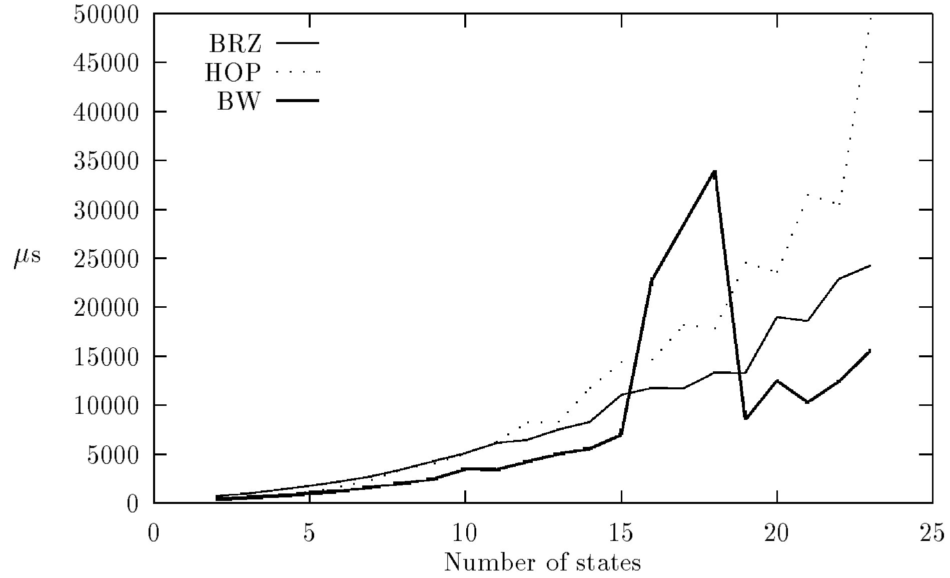


Figure 15.2: Median performance (in microseconds to minimize) versus *DFA* size.

use the superimposed graphs, Figures 15.1 and 15.2.) The performance variance of ASU and HU are shown in Figures 15.3 and 15.3 respectively. Both of these algorithms display quite small variance in performance for a given size of input.

The performance variance of BRZ, HOP, and BW are shown in Figures 15.5, 15.6, and 15.7. BRZ and HOP both display quite small variance in performance, while the performance variance for BW is very large in some cases. Figure 15.7 shows the performance variance for the BW algorithm with the 17 state *DFA* data point removed. Figure 15.8 shows all of the BW performance variance data (including the 17 state data point). Although the BW algorithm usually displays $\mathcal{O}(n \log n)$ performance (in the tested range of *DFA* sizes), the latter graph shows that, for certain input *DFA*s, the BW algorithm can occasionally give exponential performance. In the following paragraph, we briefly describe one type of *DFA* which can cause this behaviour.

Figure 15.9 gives part of a *DFA* which can cause the BW algorithm's exponential running time. (See Chapter 7 for a more in-depth discussion of the algorithm.) Depending upon the numbering (integer encoding in the implementation) of the states, the algorithm may begin by testing states p and q for equivalence. The algorithm considers each alphabet symbol $(a, b, \ldots)$ in turn. Beginning with a, it determines whether p'_0 and q'_0 are equivalent; recursively, this entails determining if the pairs (p''_0, q''_0) and (p''_1, q''_1) are equivalent. Eventually, the algorithm must determine if p'_1 and q'_1 are equivalent, etc. This unfortunate first choice of states is due to the fact that the integer encoding of the states is used to choose the first pair of states for consideration. Had the integer representations of the states been permuted, a different pair of states would have been chosen as the starting point and the

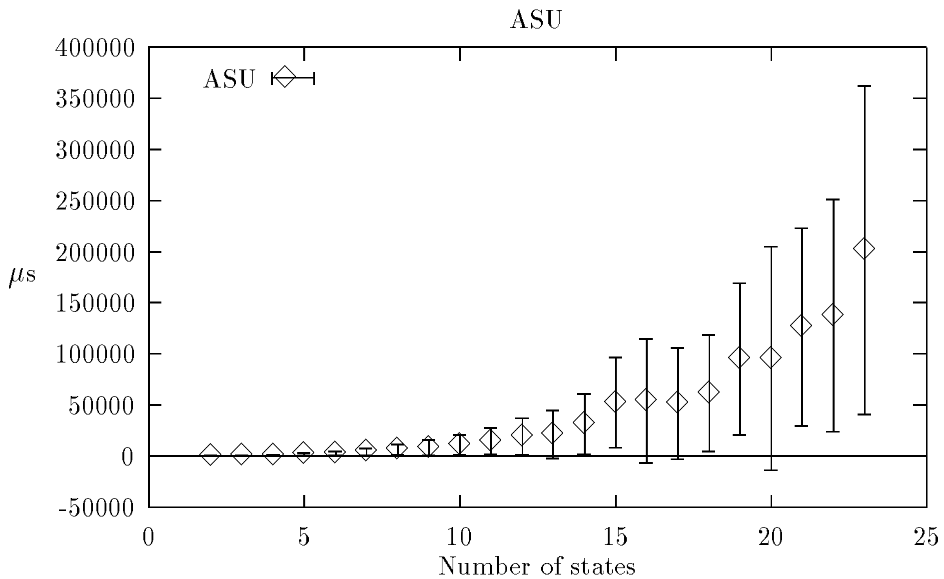


Figure 15.3: Performance variance (in microseconds to minimize) versus *DFA* size for the ASU minimization algorithm, including $+1$ and -1 standard deviation bars.

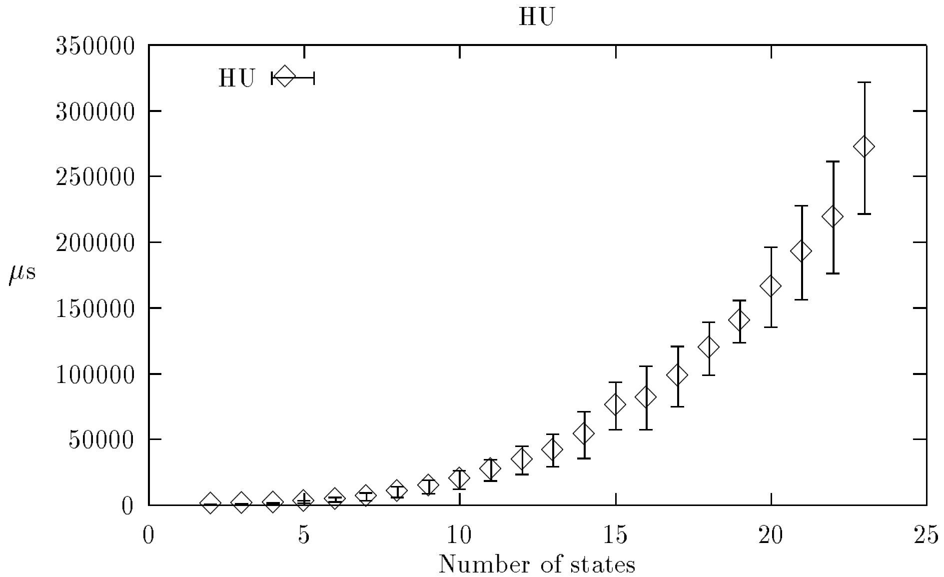


Figure 15.4: Performance variance (in microseconds to minimize) versus *DFA* size for the HU minimization algorithm, including $+1$ and -1 standard deviation bars.

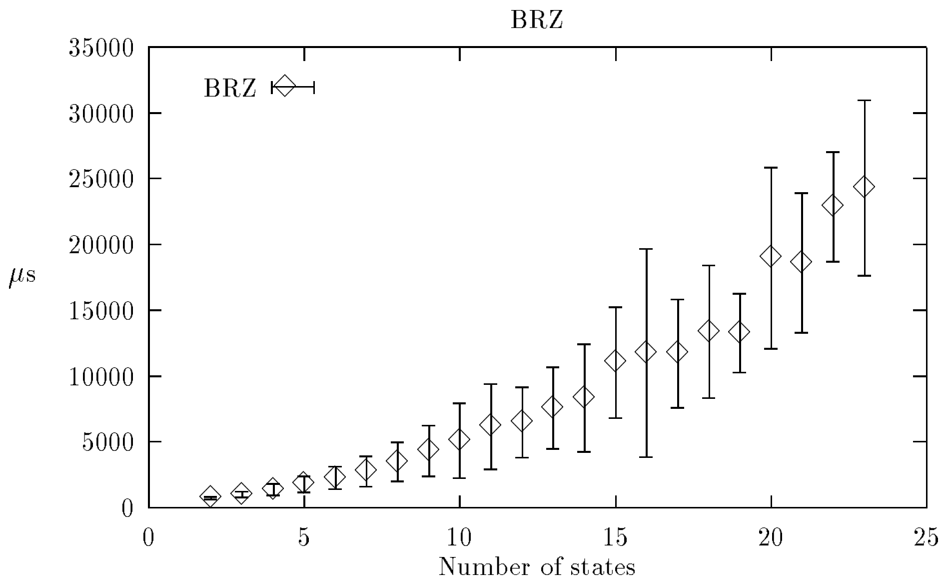


Figure 15.5: Performance variance (in microseconds to minimize) versus *DFA* size for the BRZ minimization algorithm, including +1 and −1 standard deviation bars.

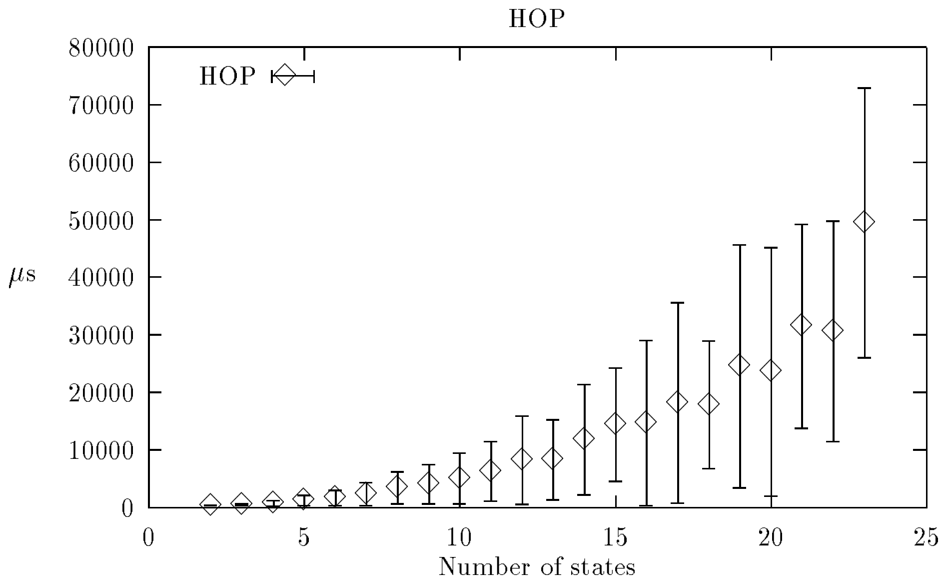


Figure 15.6: Performance variation (in microseconds to minimize) versus *DFA* size for the HOP minimization algorithm, including +1 and −1 standard deviation bars.

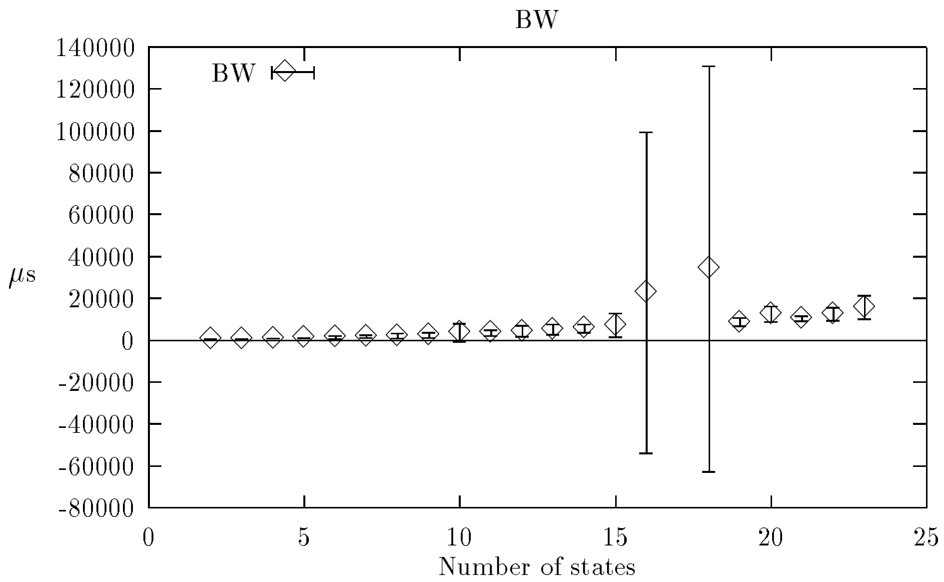


Figure 15.7: Performance variation (in microseconds to minimize) versus *DFA* size for the BW minimization algorithm, including +1 and −1 standard deviation bars.

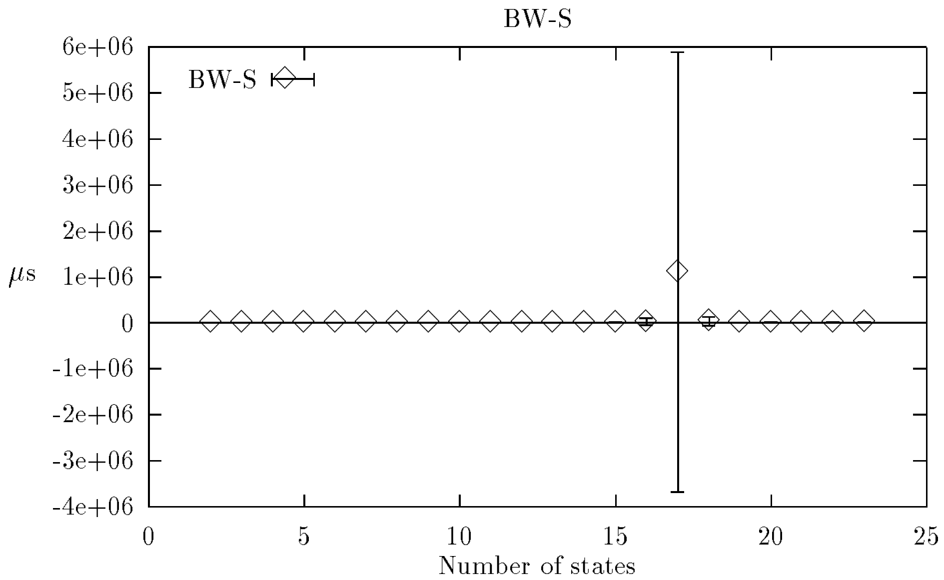


Figure 15.8: Median performance (in microseconds to minimize) versus *DFA* size for the BW minimization algorithm.

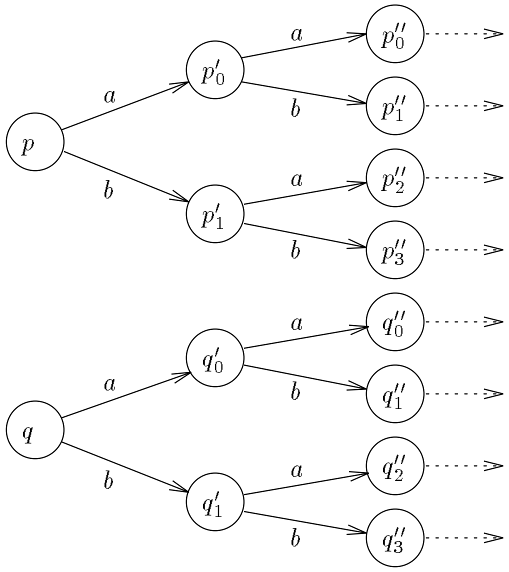


Figure 15.9: Part of a *DFA* which causes exponential running time in the BW algorithm.

exponential behaviour would not have occured.

15.5 Conclusions and recommendations

We can draw the following conclusions from the data presented in this chapter[4]:

- Given their relative performance, the five algorithms can be put into two groups, the first consisting of ASU and HU, and the second consisting of BRZ, HOP, and BW.
- The HU algorithm has the lowest performance of all of the algorithms. This is largely due to the fact that it traverses the transitions of the *DFA* in the reverse direction — a direction not favoured by most practical implementations of *DFA*s.
- The ASU algorithm also displays rather poor performance. Traditionally, the algorithm has been of interest because it is easy to understand, as we saw in Chapter 7. The simplicity of BRZ minimization algorithm makes it even more suitable for teaching purposes.
- The HOP algorithm is the best known algorithm (in term of theoretical running time), with $\mathcal{O}(n \log n)$ running time. Despite this, it is the worst of the second group of algorithms. With its excellent theoretical running time, it will outperform the BRZ and BW algorithms on extremely large *DFA*s. With memory constraints, we were unable to identify where the crossing-point of their performance is.

[4]Keeping in mind the caveats mentioned in Section 15.3

- The BRZ algorithm is extremely fast in practice, consistently outperforming Hopcroft's algorithm (HOP). This result is surprising, given the simplicity of the algorithm. The implementation of this algorithm constructs an intermediate *DFA*; the performance can be further improved by eliminating this intermediate step.

- The new algorithm (BW) displayed excellent performance. The algorithm is frequently faster than even Brzozowski's algorithm. Unfortunately, this algorithm can be erratic at times — not surprising given its exponential running time. The algorithm can be further improved using memoization — see Section 7.4.7.

Given these conclusions, we can make the following recommendations:

1. Use the new algorithm (BW, appearing as Algorithm 7.28 in this dissertation), especially in real-time applications (see Section 7.4.7 for an explanation of why this algorithm is useful for real-time applications). If the performance is still insufficient, modify the algorithm to make greater use of memoization.

2. Use Brzozowski's algorithm (derived in Section 7.2), especially when simplicity of implementation or consistent performance is desired. The algorithm is able to deal with an *FA* as input (instead of only *DFA*s), producing the minimal equivalent *DFA*. When a minimization algorithm is being combined with a *FA* construction algorithm, Brzozowski's minimization algorithm is usually the best choice. The *DFA* construction algorithms are usually significantly slower than the *FA* construction algorithms, as is shown in Chapter 14. For this reason, a *FA* construction algorithm combined with Brzozowski's minimization algorithm will produce the minimal *DFA* faster than a *DFA* construction algorithm combined with any of the other minimization algorithms.

   Brzozowski's algorithm can be further improved by eliminating the *DFA* which is constructed in an intermediate step.

3. Use Hopcroft's algorithm (Algorithm 7.26) for massive *DFA*s. It is not clear from the data in this chapter precisely when this algorithm becomes more attractive than the new one or Brzozowski's.

4. The two most-commonly taught text-book algorithms (the Aho-Sethi-Ullman algorithm, Algorithm 7.21, and the Hopcroft-Ullman algorithm, Algorithm 7.24) do not appear to be good choices for high performance. Even for simplicity of implementation, Brzozowski's algorithm is better.

The following procedure can be used to choose a deterministic finite automata minimization algorithm:

Choices := {BRZ, BW, HOP};
if easy of understanding is important **then**
  Choices := *Choices* $\cap$ {BRZ, BW}

fi;
if the application has real-time deadlines **then**
  $Choices := Choices \cap \{\mathrm{BW}\}$
fi;
if asymptotic performance is important **then**
  $Choices := Choices \cap \{\mathrm{HOP}\}$
fi;
if good average-case performance is important **then**
  $Choices := Choices \cap \{\mathrm{BRZ}\}$
fi;

Part V

Epilogue

Chapter 16

Conclusions

The conclusions of this dissertation are summarized in this chapter. We first present the more general conclusions. Some chapters are accompanied by specific conclusions. A selection of those conclusions will be presented as well.

16.1 General conclusions

The general conclusions are:

- Each of the taxonomies presented in this dissertation covers an area of computer science which has been active for at least thirty-five years. The original presentations of the algorithms differ vastly in their purposes (the intended use of the algorithms) and presentation styles. A great deal of reverse engineering was required to identify the inner workings of the algorithms. Additionally, one of the most powerful techniques in computing science, *abstraction*, was most helpful in finding the common threads in the families of algorithms.

- The main goal of this research was to improve the accessibility of the algorithms. This has been achieved in a number of ways, thanks to the fact that some of the research has already been reported in preliminary papers:
  - The taxonomies have been used at a number of institutes as reference material for teaching.
  - The toolkits have been used in industry for the creation of reliable software components and at universities for teaching the essential concepts of reusable software construction.
  - The performance data has been used extensively by software engineers to select high performance algorithms for applications.

- The research reported in this dissertation ranges from the more theoretical taxonomies and algorithm derivations to the very practical toolkits and algorithm performance benchmarks. Although these areas seem to be at opposite ends of the spectrum, it proved useful as well as easy to consider them together.

- The taxonomies are more than just surveys. They explain and classify the algorithms in such depth that the corresponding literature (the original articles describing the algorithms and subsequent surveys) is no longer required reading. Similarly, some existing implementations of well-known algorithms are made obsolete by the toolkits, since the toolkits contain implementations of many different algorithms solving the same problem.

- A single taxonomy construction methodology and framework was successfully used to develop taxonomies for three totally different problems in computing science. However, successfully using the same methodology does not necessarily mean that all of the algorithms must be derived using the same formalisms. The formalisms used for each of the three problems can be described as follows:

  - *Pattern matching.* The algorithms were derived as imperative programs, with a largely formal approach to the derivation.
  - *Finite automata construction.* The constructions were given as mathematical functions, each being derived as the composition of a number of other functions. Algorithms implementing the functions were also considered, although outside the framework of the taxonomy.
  - *Deterministic finite automata minimization.* The minimization algorithms were derived as a mixture of the two formalisms mentioned above.

- In constructing the taxonomies, we did not adhere strictly to Jonkers' methodology. In his dissertation, at least one of the algorithms was derived in a formal manner. In the taxonomies in this dissertation, a relatively liberal approach to the interpretation of the *details* is adopted.

- It does not appear that taxonomies could be constructed for all types of algorithm families. The algorithms treated in this dissertation have something significant in common: they all have simple specifications of both the input and the output. In particular, it is not clear how easy it would be to taxonomize algorithms such as those based upon heuristics.

- Some of the algorithms treated in the taxonomies are usually considered to be inaccessible in the standard literature. For example, the efficient Commentz-Walter pattern matching algorithm and Hopcroft's deterministic finite automata minimization algorithm are both relatively difficult to understand. The refinement-based derivation of these algorithms, and their corresponding implementations in the toolkits, should make them more accessible than before.

- Truly capitalizing on the organization provided by the taxonomies required the construction of the toolkits. Even with the taxonomies, constructing the toolkits still required a remarkable number of software engineering decisions. Conclusions can be drawn for both algorithm theoreticians and software engineers:

  - Part III points out that the completion of a taxonomy still leaves a number of software engineering decisions. Some examples of these decisions are those that lead to the call-back public interface in the class libraries — and the consequent possibility of multi-threading.
  - The nature of the taxonomies, in which common parts of algorithms are factored, can be used to great effect in reusing parts of class libraries. As outlined in Chapter 8, a number of design issues surround reuse, for example abstract classes and virtual member functions versus templates.

- The toolkits began as a sidetrack from the primary task of developing the taxonomies. With the taxonomies in-hand, the toolkits were implemented extremely rapidly — eliminating much of the development time usually required for such toolkits. Two clearly identifiable aspects of the toolkits were influenced by the taxonomies:
  - Since all of the abstract algorithms were presented (in the taxonomies) in the same formalism, it was possible to create a coherent toolkit with the C++ classes making use of a common set of foundation classes.
  - The inheritance hierarchy is usually one of the most difficult parts of a class library to design. Given a taxonomy family tree, the corresponding class inheritance hierarchy was easy to structure.

- Toolkits which provide more than one algorithm solving the same problem are difficult to use effectively without information on the performance of the algorithms in practice. The information in Part IV provides data (and analysis of the data) and recommendations on the use of algorithms in the toolkits.

- Part IV shows that the algorithms with the best theoretical complexity are not always the fastest in practice, even on non-trivial input.

- Experience with the object-oriented toolkits (and with older C toolkits) show that the object-oriented approach to toolkit design does not affect the algorithm performance significantly.

- At first glance, it may appear that the taxonomies only serve to classify existing algorithms. By combining the taxonomy details in new ways, or by making use of techniques developed during the taxonomization, it is possible to construct new algorithms. In particular, the following interesting algorithms were derived:
  - The use of 'predicate weakening' was used in Chapter 4 to derive interesting new variants of the Commentz-Walter multiple-keyword pattern matching algorithms.
  - A new regular expression pattern matching algorithm was developed in Chapter 5 — answering an open question posed by A.V. Aho.

- A number of new finite automata construction algorithms were derived in Chapter 6. Unfortunately, the performance data presented in Chapter 14 shows that several of these algorithms are not particularly useful in practice. Some do, however, serve the purpose of being easy to understand.
- A new deterministic finite automata minimization algorithm, the only existing algorithm suitable for use in real-time applications, was derived in Chapter 7.

16.2 Chapter-specific conclusions

Some of the chapter-specific conclusions are:

- *Chapter 4.* The Aho-Corasick and the multiple keyword Knuth-Morris-Pratt pattern matching algorithms all share a common skeleton. They differ only in the implementation of a particular Moore machine transition function. Similarly, the Commentz-Walter algorithms also share a common skeleton, differing in the shift function used to skip portions of the input string.

- *Chapter 4.* The technique of 'predicate weakening' was extremely useful in developing new shift functions for both the Commentz-Walter and the Boyer-Moore pattern matching algorithms.

- *Chapter 4.* Another taxonomy of pattern matching algorithms, due to Hume and Sunday, was easily incorporated into the taxonomy in this dissertation.

- *Chapter 5.* There *is* a Boyer-Moore type algorithm for regular expression pattern matching (see Chapter 5 for the algorithm and its precomputation), answering an open question posed by A.V. Aho in [Aho80, p. 342]. The derivation relied heavily on techniques developed for the taxonomy of keyword pattern matching algorithms, and it is doubtful that the algorithm could have been easily invented without the use of these techniques. Preliminary testing of the algorithm shows that it is frequently faster than a generalization of the Aho-Corasick algorithm.

- *Chapter 6.* The earlier taxonomy presented in [Wat93a] contained two taxonomy trees. The derivations presented there seemed to indicate that the two subfamilies of algorithms were related, but could not be derived from one another. The taxonomy presented in this dissertation shows that they can all be derived from a single 'canonical' algorithm.

- *Chapter 6.* The canonical algorithm encoded the 'maximal' amount of information in states. All of the other algorithms were derived by either changing the representation of the states or by omitting some of the information, thereby merging states. Furthermore, all of the algorithms were derived as compositions of mathematical functions.

- *Chapter 6.* Even the most recent algorithms, such as those presented by Antimirov in [Anti94, Anti95] and by Antimirov and Watson in [AW95], were incorporated into the taxonomy.
- *Chapter 7.* Brzozowski's minimization algorithm proved to be extremely easy to derive and to understand. Unfortunately, it does not appear possible to derive it from the other minimization algorithms. In the past, the origins of this algorithm have also been accidentally misattributed.
- *Chapter 7.* A new minimization algorithm, suitable for real-time applications, was derived. The algorithm computes a relation (for use in minimization) as a fixed point, from the safe side. This implies that its intermediate computations are usable in reducing the size of (but perhaps not *minimizing*) the *DFA*.
- *Chapter 9.* Designing and structuring generic software is much more difficult than designing software for a single application. The general structure of the pattern matching taxonomy proved to be helpful in guiding the structure of the SPARE Parts.
- *Chapter 10.* Finite automata toolkits, such as the FIRE Engine and FIRE Lite, have proven to be general enough to find use in the following areas: compiler construction, hardware modeling, and computational biology.
- *Chapter 15.* Despite its exponential worst-case running time, Brzozowski's minimization algorithm has excellent practical performance for realistic input. Additionally, the new minimization algorithm also displays excellent practical running time, even though it also has exponential worst-case running time.

16.3 A personal perspective

The research reported in this dissertation, and my experiences while performing and writing it, show that a delicate balance between the practical and the theoretical is crucial to a computer scientist.

Chapter 17

Challenges and open problems

While the research reported in this dissertation has answered a number of questions, and brought some order to the field of regular language algorithms, it has also suggested some new directions for research. The following problems (which are of various levels of difficult) are suitable for Master's or Ph.D students:

1. Keyword pattern matching:
   (a) Expand the taxonomy to include Commentz-Walter and Boyer-Moore algorithms which retain information about previous matches, for reuse in subsequent match attempts (see Remark 4.156).
   (b) Consider the use of 'match orders' (Section 4.5) for the Commentz-Walter multiple-keyword algorithms.
   (c) Construct taxonomies of: approximate pattern matching algorithms, multi-dimensional pattern matching algorithms, and tree and graph pattern matching algorithms.
2. The new regular expression pattern matching algorithm:
   (a) Explore the use of other weakenings, and use 'strategies' (Section 4.4) to catalogue them.
   (b) Explore the use of a left lookahead symbol.
   (c) Quantify the effects (on the performance of the algorithm) of the type of finite automaton used (deterministic or nondeterministic automata; with or without ε-transitions).
   (d) Consider a version of the algorithm which reuses previous match information.
3. Extend the research to different pattern matching problems, such as multi-dimensional pattern matching and tree pattern matching.
4. The taxonomy of finite automata constructions:

   (a) Modify the algorithms to include the construction of Moore machines and Mealy machines (regular transducers).
   (b) Include the construction of tree automata and graph automata.

5. SPARE Parts:
   (a) Add tree pattern matching to the toolkit.
   (b) Expand the toolkit to deal with strings of (C++) objects, instead of only characters.

6. FIRE Lite:
   (a) Add regular transductions to the toolkit.
   (b) Expand the toolkit to deal with strings of (C++) objects, instead of only characters.
   (c) Implement the minimization algorithms that are not presently included in the toolkit.

7. The performance of *DFA* minimization algorithms:
   (a) Benchmark the minimization algorithms for the types of *DFA* which typically arise in digital circuit applications.

References

[AC75] AHO, A.V. and M.J. CORASICK. Efficient string matching: an aid to bibliographic search, *Comm. ACM,* **18**(6) (1975) 333–340.

[Aho80] AHO, A.V. Pattern matching in strings, in: R.V. Book, ed., *Formal Language Theory: Perspectives and Open Problems.* (Academic Press, New York, 1980) 325–347.

[Aho90] AHO, A.V. Algorithms for finding patterns in strings, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A.* (North-Holland, Amsterdam, 1990) 257–300.

[AHU74] AHO, A.V., J.E. HOPCROFT, and J.D. ULLMAN. *The Design and Analysis of Computer Algorithms.* (Addison-Wesley, Reading, MA, 1974).

[AMS92] AOE, J.-I., K. MORIMOTO, and T. SATO. An efficient implementation of trie structures, *Software — Practice and Experience,* **22**(9), (September 1992) 695–712.

[ANSI95] AMERICAN NATIONAL STANDARDS INSTITUTE. *Programming language C++.* Working Paper for Draft Proposed International Standard for Information Systems, (ANSI Committee X3J16/ISO WG21, 31 January 1995).

[Anti94] ANTIMIROV, V.M. Partial derivatives of regular expressions and finite automata constructions, Technical Report CRIN 94-R-245, CRIN, France, December 1994.

[Anti95] ANTIMIROV, V.M. Partial derivatives of regular expressions and finite automata constructions, in: E.W. Mayr and C. Puech, eds., *Twelfth Annual Symposium on Theoretical Aspects of Computer Science,* Lecture Notes in Computer Science 900 (Springer-Verlag, Berlin, 1995) 455–466.

[Aoe94] AOE, J.-I. *Computer Algorithms: String Pattern Matching Strategies.* (IEEE Computer Society Press, 1994).

[ASU86] AHO, A.V., R. SETHI, and J.D. ULLMAN. *Compilers: Principles, Techniques, and Tools.* (Addison-Wesley, Reading, MA, 1988).

[AU92] Aho, A.V. and J.D. Ullman. *Foundations of Computer Science.* (Computer Science Press, New York, 1992).

[AW95] Antimirov, V.M. and B.W. Watson. From regular expressions to small NFAs through partial derivatives, Unpublished Manuscript, April 1995.

[B-K93a] Brüggemann-Klein, A. Regular expressions into finite automata, *Theoretical Computer Science* **120** (1993) 197–213.

[B-K93b] Brüggemann-Klein, A. *Private communication* (July 1993).

[BL77] Backhouse, R.C. and R.K. Lutz. Factor graphs, failure functions and bi-trees, in: G. Goos and J. Hartmanis, eds., *Fourth Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science 52 (Springer-Verlag, Berlin, 1977) 61–75.

[BM77] Boyer, R.S. and J.S. Moore. A fast string searching algorithm, *Comm. ACM,* **20**(10) (1977) 62–72.

[Booc94] Booch, G. *Object oriented analysis and design, with applications.* (Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994).

[Brau88] Brauer, W. On minimizing finite automata, *EATCS Bulletin* **35** (June 1988).

[Broy83] Broy, M. Program construction by transformations: a family tree of sorting programs, in: A.W. Biermann and G. Guiho, eds., *Computer Program Synthesis Methodologies* (1983) 1–49.

[Brzo62] Brzozowski, J.A. Canonical regular expressions and minimal state graphs for definite events, in: *Mathematical theory of Automata, Vol. 12 of MRI Symposia Series.* (Polytechnic Press, Polytechnic Institute of Brooklyn, NY, 1962) 529–561.

[Brzo64] Brzozowski, J.A. Derivatives of regular expressions, *J. ACM* **11**(4) (1964) 481–494.

[BS86] Berry, G. and R. Sethi. From regular expressions to deterministic automata, *Theoretical Computer Science* **48** (1986) 117–126.

[BS95] Brzozowski, J.A. and C.-J. Seger. *Asynchronous circuits.* (Springer, Berlin, 1995).

[B-YR90] Baeza-Yates, R. and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching, in: J.R. Gilbert and R. Karlsson, eds., *Second Scandinavian Workshop on Algorithm Theory,* Lecture Notes in Computer Science 447 (Springer-Verlag, Berlin, 1990) 332–347.

[Budd91] BUDD, T.A. *An introduction to object-oriented programming.* (Addison-Wesley, Reading, MA, 1991).

[Budd94] BUDD, T.A. *Classic data structures in C++.* (Addison-Wesley, Reading, MA, 1994).

[CE95] CARROLL, M.D. and M.A. ELLIS. *Designing and coding reusable C++.* (Addison-Wesley, Reading, MA, 1995).

[CH91] CHAMPARNAUD, J.M. and G. HANSEL. Automate: A computing package for automata and finite semigroups, *J. Symbolic Computation* **12** (1991) 197–220.

[Cham93] CHAMPARNAUD, J.M. From a regular expression to an automaton, Technical Report, IBP, LITP, Université Paris 7, Paris, France, Working document 23 September 1993.

[Chan92] CHANG, C.-H. From regular expressions to DFAs using compressed NFAs, Ph.D dissertation, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, NY, October 1992.

[Com79a] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer-Verlag, Berlin, 1979) 118–132.

[Com79b] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[Copl92] COPLIEN, J.O. *Advanced C++: programming styles and idioms.* (Addison-Wesley, Reading, MA, 1992).

[CP92] CHANG, C.-H. and R. PAIGE. From regular expressions to DFAs using compressed NFAs, Technical Report, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, NY, 1992.

[CR94] CROCHEMORE, M. and W. RYTTER. *Text Algorithms.* (Oxford University Press, Oxford, England, 1994).

[Darl78] DARLINGTON, J. A synthesis of several sorting algorithms, *Acta Informatica* **11** (1978) 1–30.

[DeRe74] DEREMER, F.L. Lexical analysis, in: F.L. Bauer and J. Eickel, eds., *Compiler Construction: an Advanced Course,* Lecture Notes in Computer Science 21 (Springer-Verlag, Berlin, 1974) 109–120.

[Dijk76] DIJKSTRA, E.W. *A discipline of programming.* (Prentice Hall, Englewood Cliffs, NJ, 1976).

[Earl70] EARLEY, J. An efficient context-free parsing algorithm, *Comm. ACM* **13**(2) (February 1970) 94–102.

[vdEi92] VAN DEN EIJNDE, J.P.H.W. Program derivation in acyclic graphs and related problems, Computing Science Report 92/04, Eindhoven University of Technology, The Netherlands, 1992.

[tEvG93] TEN EIKELDER, H.M.M. and H.P.J. VAN GELDROP. On the correctness of some algorithms to generate finite automata for regular expressions, Computing Science Report 93/32, Eindhoven University of Technology, The Netherlands, 1993.

[t-Ei91] TEN EIKELDER, H.M.M. Some algorithms to decide the equivalence of recursive types, Computing Science Report 91/31, Eindhoven University of Technology, The Netherlands, 1991.

[EM85] EHRIG, E. and B. MAHR. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* (Springer-Verlag, Berlin, 1985).

[Fred60] FREDKIN, E. Trie memory, *Comm. ACM* **3**(9) (1960) 490–499.

[FS93] FAN, J.-J. and K.-Y. SU. An efficient algorithm for matching multiple patterns, *IEEE Trans. on Knowledge and Data Engineering* **5**(2) (April 1993) 339–351. Reprinted in [Aoe94].

[GB-Y91] GONNET, G.H. and R. BAEZA-YATES. *Handbook of Algorithms and Data Structures (In Pascal and C).* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[GHJV95] GAMMA, E., R. HELM, R. JOHNSON, and J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, Reading, MA, 1995).

[GJ90] GRUNE, D. and C.J.H. JACOBS. *Parsing Techniques: A Practical Guide.* (Ellis Horwood, West Sussex, England, 1990).

[Glus61] GLUSHKOV, V.M. The abstract theory of automata, *Russian Mathematical Surveys* **16** (1961) 1–53.

[Gold93] GOLDBERG, R.R. Finite state automata from regular expression trees, *The Computer Journal* **36**(7) (1993) 623–630.

[Grie73] GRIES, D. Describing an algorithm by Hopcroft, *Acta Informatica* **2** (1973) 97–109.

[HKR94] HEERING, J., P. KLINT, and J. REKERS. Lazy and incremental program generation, *ACM Trans. on Programming Languages and Systems* **16**(3) (1994) 1010–1023.

[HN92] HENRICSON, M. and E. NYQUIST. Programming in C++: Rules and recommendations, Technical Report M90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, Alvsjo, Sweden, 1992.

[Hopc71] HOPCROFT, J.E. An $n \log n$ algorithm for minimizing the states in a finite automaton, in: Z. Kohavi, ed., *The Theory of Machines and Computations.* (Academic Press, New York, 1971) 189–196.

[HS91] HUME, S.C. and D. SUNDAY. Fast string searching, *Software—Practice and Experience* **21**(11) (1991) 1221–1248.

[HU79] HOPCROFT, J.E. and J.D. ULLMAN. *Introduction to Automata, Theory, Languages, and Computation.* (Addison-Wesley, Reading, MA, 1979).

[Huff54] HUFFMAN, D.A. The synthesis of sequential switching circuits, *J. Franklin Institute* **257**(3) (1954) 161–191 and **257**(4) (1954) 275–303.

[ISO90] ISO/IEC. *Programming languages — C.* International Standard 9899:1990, (ISO, 1st edition, 15 December 1990).

[John86] JOHNSON, J.H. INR: A program for computing finite automata, Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1986.

[Jonk82] JONKERS, H.B.M. Abstraction, specification and implementation techniques, Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1982. Also appears as MC-Tract 166, Mathematical Center, Amsterdam, The Netherlands, 1983.

[JPTW90] JANSEN, V., A. POTHOFF, W. THOMAS, and U. WERMUTH. A short guide to the Amore system, *Aachener Informatik-Berichte* **90**(02), Lehrstuhl für Informatik II, (Universität Aachen, January 1990).

[KMP77] KNUTH, D.E., J.H. MORRIS and V.R. PRATT. Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[Knut65] KNUTH, D.E. On the translation of languages from left to right, *Inform. Control* **8** (1965) 607–639.

[KP95] KELLER, J.P. and R. PAIGE. Program derivation with verified transformations — a case study, *Comm. on Pure and Applied Mathematics* **48** (1995) In press.

[KR88] KERNIGHAN, B.W. and D.M. RITCHIE. *The C Programming Language.* (Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988).

[KW70] KAMEDA, T. and P. WEINER. On the state minimization of nondeterministic finite automata, *IEEE Trans. on Computers* **C-19**(7) (1970) 617–627.

[Leis77] LEISS, E. Regpack: An interactive package for regular languages and finite automata, Research Report CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada, October 1977.

[Lipp91] LIPPMAN, S.B. *C++ primer.* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[Marc90] MARCELIS, A.J.J.M. On the classification of attribute evaluation algorithms, *Science of Computer Programming* **14** (1990) 1–24.

[MeyB88] MEYER, B. *Object oriented software construction.* (Prentice Hall, Englewood Cliffs, NJ, 1988).

[MeyB92] MEYER, B. *Eiffel: The Language.* (Prentice Hall, Englewood Cliffs, NJ, 1992).

[MeyB94] MEYER, B. *Reusable Software: The Base Object-Oriented Component Libraries.* (Prentice Hall, Englewood Cliffs, NJ, 1994).

[MeyS92] MEYERS, S. *Effective C++: 50 specific ways to improve your programs.* (Addison-Wesley, Reading, MA, 1992).

[Mirk65] MIRKIN, B.G. On dual automata, *Kibernetika* **2**(1) (1966) 7–10.

[Moor56] MOORE, E.F. Gedanken-experiments on sequential machines, in: C.E. Shannon and J. McCarthy, eds., *Automata Studies.* (Princeton University Press, Princeton, NJ, 1956) 129–153.

[Murr93] MURRAY, R.B. *C++ strategies and tactics.* (Addison-Wesley, Reading, MA, 1993).

[MY60] MCNAUGHTON, R. and H. YAMADA. Regular expressions and state graphs for automata, *IEEE Trans. on Electronic Computers* **9**(1) (1960) 39–47.

[Myhi57] MYHILL, J. Finite automata and the representation of events, Technical Report WADD TR-57-624, Wright Patterson AFB, Ohio, 1957, 112–137.

[Nero58] NERODE, A. Linear automaton transformations, *Proc. AMS* **9** (1958) 541–544.

[Park89] PARKER, S.P., ed., *Dictionary of scientific and technical terms.* (McGraw-Hill, New York, 4th edition, 1989)

[Perr90] PERRIN, D. Finite Automata, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B.* (North-Holland, Amsterdam, 1990) 1–57.

[Pirk92] PIRKLBAUER, K. A study of pattern-matching algorithms, *Structured Programming* **13** (1992) 89–98.

[Plau95] PLAUGER, P.J. *The Draft Standard C++ Library.* (Prentice Hall, New Jersey, 1995).

[PTB85] PAIGE, R., R.E. TARJAN and R. BONIC. A linear time solution to the single function coarsest partition problem, *Theoretical Computer Science* **40** (1985) 67–84.

[PTVF92] PRESS, W.H., S.A. TEUKOLSKY, W.T. VETTERLING and B.P. FLANNERY. *Numerical Recipes in C: The Art of Scientific Computing.* (Cambridge University Press, Cambridge, England, 2nd edition, 1992).

[RS59] RABIN, M.O. and D. SCOTT. Finite automata and their decision problems, *IBM J. Research* **3**(2) (1959) 115–125.

[RW93] RAYMOND, D.R. and D. WOOD. The Grail papers: Version 2.0, Department of Computer Science, University of Waterloo, Canada, January 1994. Available by ftp from `CS-archive.uwaterloo.ca`.

[Ryt80] RYTTER, W. A correct preprocessing algorithm for Boyer-Moore string-searching, *SIAM J. Comput.* **9**(2) (1980) 509–512.

[SE90] STROUSTRUP, B. and M. ELLIS. *The annotated C++ reference manual.* (Addison-Wesley, Reading, MA, 1990).

[SL94] STEPANOV, A. and M. LEE. Standard Template Library, Computer Science Report, Hewlett-Packard Laboratories, 1994.

[Smit82] SMIT, G. DE V. A comparison of three string matching algorithms, *Software — Practice and Experience* **12** (1982) 57–66.

[vdSn85] VAN DE SNEPSCHEUT, J.L.A. Trace theory and VLSI design, Ph.D dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1985. Also available as Lecture Notes in Computer Science 200 (Springer-Verlag, Berlin, 1985).

[vdSn93] VAN DE SNEPSCHEUT, J.L.A. *What computing is all about.* (Springer-Verlag, New York, 1993).

[Souk94] SOUKUP, J. *Taming C++: Pattern Classes and Persistence for Large Projects.* (Addison-Wesley, Reading, MA, 1994).

[SS-S88] SIPPU, S. and E. SOISALON-SOININEN. *Parsing Theory: Languages and Parsing, Vol. 1.* (Springer-Verlag, Berlin, 1988).

[Step94] STEPHEN, G.A. *String searching algorithms.* (World Scientific, Singapore, 1994).

[Stro91] STROUSTRUP, B. *The C++ programming language.* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[Stro94] STROUSTRUP, B. *The Design and Evolution of C++.* (Addison-Wesley, Reading, MA, 1994).

[Tali94] TALIGENT. *Taligent's Guide to Designing Programs: Well-Mannered Object Oriented Design in C++.* (Addison-Wesley, Reading, MA, 1994).

[Teal93] TEALE, S. *C++ IOStreams Handbook.* (Addison-Wesley, Reading, MA, 1993).

[Thom68] THOMPSON, K. Regular expression search algorithms, *Comm. ACM* **11**(6) (1968) 419–422.

[Urba89] URBANEK, F. On minimizing finite automata, *EATCS Bulletin* **39** (October 1989).

[Wat93a] WATSON, B.W. A taxonomy of finite automata construction algorithms, Computing Science Report 93/43, Eindhoven University of Technology, The Netherlands, 1993. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat93b] WATSON, B.W. A taxonomy of finite automata minimization algorithms, Computing Science Report 93/44, Eindhoven University of Technology, The Netherlands, 1993. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat94a] WATSON, B.W. The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/pattm/`.

[Wat94b] WATSON, B.W. An introduction to the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions, Computing Science Report 94/21, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat94c] WATSON, B.W. The design and implementation of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions, Computing Science Report 94/22, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wein95] WEINER, R. *Software Development using Eiffel: There can be life other than C++.* (Prentice Hall, Englewood Cliffs, NJ, 1995).

[WM94] WU, S. and U. MANBER. A fast algorithm for multi-pattern searching, Technical Report TR94-17, University of Arizona, Tucson, Arizona, May 1994. Available by e-mail from `udi@cs.arizona.edu`.

[Wood87] WOOD, D. *Theory of Computation.* (Harper & Row, New York, 1987).

[WW94] WATSON, B.W. and R.E. WATSON. A Boyer-Moore type algorithm for regular expression pattern matching, Computing Science Report 94/31, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `watson@win.tue.nl`.

[WZ92] WATSON, B.W. and G. ZWAAN. A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992. Available by e-mail from `watson@win.tue.nl` or `wsinswan@win.tue.nl`.

[WZ95] WATSON, B.W. and G. ZWAAN. A taxonomy of sublinear multiple keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `wsinswan@win.tue.nl`.

Index

Summary

A number of fundamental computing science problems have been studied since the 1950s and 1960s. For each of these problems, numerous solutions (in the form of algorithms) have been developed over the years. In these collections of solutions, we can identify the following three deficiencies:

1. Algorithms solving the same problem are difficult to compare to one another. This could be due to the use of different programming languages, paradigms, styles of presentation — or simply the addition of unnecessary details.

2. Collections of algorithm implementations solving a problem are difficult to find. Some of the algorithms are presented in a relatively obsolete manner, either using a now-defunct notation or obsolete programming language, making it difficult to either implement the algorithm or find an existing implementation.

3. Little is known about the comparative practical running time performance of the algorithms. The lack of existing implementations in one and the same framework, especially of some of the older algorithms, has made it difficult to determine the running time characteristics of the algorithms. Selection of an algorithm must then be made on the basis of the theoretical running time, or simply by guessing.

In this dissertation, a solution to each of these deficiencies is presented. To present the solutions, we use the following three fundamental computing science problems:

1. Keyword pattern matching in strings. Given a finite non-empty set of keywords (the patterns) and an input string, find the set of all occurrences of a keyword as a substring of the input string.

2. Finite automata (*FA*) construction. Given a regular expression, construct a finite automaton which accepts the language denoted by the regular expression.

3. Deterministic finite automata (*DFA*) minimization. Given a *DFA*, construct the unique minimal *DFA* accepting the same language.

In the following paragraphs, we will outline the solutions presented for each of the deficiencies.

The difficulty in comparing algorithms is overcome by creating a *taxonomy* of algorithms for a given problem. Each of the algorithms is rewritten in a common notation and is examined to determine the essential ingredients that distinguish it from any other algorithm. These ingredients (known as *details*) can take the form of problem details (a restriction on the class of problems solved), or algorithm details (some correctness-preserving change to the algorithm to improve efficiency). Once each algorithm has been reduced in such a way, it can be characterized by its set of details. In presenting the taxonomy, the common details of several algorithms can be factored and presented together. In this fashion, a 'family tree' of the algorithms is constructed, showing clearly what any two algorithms have in common and where they differ. Because the root of the family tree is a naïve, and correct, algorithm and the details are applied in a correctness-preserving manner, the correctness argument for each of the algorithms is implicit in the taxonomy.

The common notation and presentations in the taxonomies enable us to implement the algorithms uniformly, in the form of a *class library* (also known as a *toolkit*). The factoring of essential details, inherent in the taxonomies, leads to factoring of common components in the inheritance hierarchy of the class library. Object-oriented concepts, such as virtual (or deferred) member functions, inheritance and template classes, prove to be useful in presenting a coherent class library, the structure of which reflects the taxonomy from which it was created. For the first time, most (if not all) solutions are presented in a single class library, giving clients of the library a large choice of objects and functions.

With a class library that contains most of the known solutions, we are finally able to gather data on the performance of the algorithms in practice. Since the algorithms are taken from a single class library which was implemented by one person, and the quality of implementation of the class library is homogeneous, the relative performance data gathered (comparing algorithms) is not biased by the implementation. This performance data allows software engineers to make more informed decisions (based upon their needs, and the characteristics of their input data) concerning which algorithm and therefore which objects and functions to use in their applications.

The development of the taxonomies has not been without its spin-offs. In each of the three taxonomies presented, significant new algorithms have been developed. These algorithms are also implemented in the corresponding class libraries. The techniques developed in the taxonomy of pattern matching algorithms proved to be particularly useful in deriving an algorithm for regular expression pattern matching, and in doing so, answering an open question posed by A.V. Aho in [Aho80, p. 342].

Samenvatting (Dutch summary)

Er bestaan een aantal fundamentele problemen in de informatica die al sinds begin jaren 50 en 60 bestudeerd worden. Voor elk van deze problemen zijn er in de loop der tijd een groot aantal oplossingen (in de vorm van algoritmen) ontwikkeld. Bij deze oplossingen kunnen we drie verschillende tekortkomingen onderscheiden:

1. Algoritmen die één en hetzelfde probleem oplossen zijn moeilijk met elkaar te vergelijken. Dit kan worden veroorzaakt door het gebruik van verschillende programmeertalen, paradigmata, presentatiestijlen of eenvoudigweg door toevoeging van onnodige details.

2. Collecties van implementaties van algoritmen die hetzelfde probleem oplossen, zijn moeilijk te vinden. Sommige algoritmen worden op een vrij obsolete manier gepresenteerd (hetzij door het gebruik van een achterhaalde notatie dan wel door toepassing van verouderde programmeertalen). Dit maakt het moeilijk de algoritme te implementeren of een bestaande implementatie te vinden.

3. Er is weinig bekend over de relatieve snelheid van de algoritmen in de praktijk. Omdat implementaties in één en hetzelfde raamwerk ontbreken (vooral daar waar het oudere algoritmen betreft), is het moeilijk om de snelheidskarakteristieken te bepalen. De keuze van de algoritme moet dan gemaakt worden op basis van de theoretische snelheid of gewoon door gissen.

In dit proefschrift wordt voor ieder van deze tekortkomingen een oplossing gepresenteerd aan de hand van de volgende drie fundamentele informatica-problemen:

1. Patroonherkenning in symboolrijen. Uitgaande van een eindige niet-lege verzameling van sleutelwoorden (de patronen) en een invoerrij, vind de verzameling voorkomens van een sleutelwoord als een subrij van de invoerrij.

2. Constructie van eindige automaten (*FA*). Gegeven een reguliere expressie, construeer een eindige automaat die de taal van de reguliere expressie accepteert.

3. Minimalisatie van deterministische eindige automaten (*DFA*). Gegeven een *DFA*, construeer de unieke minimale *DFA* die dezelfde taal accepteert.

In de volgende alinea's zullen de oplossingen voor de drie verschillende tekortkomingen kort uiteengezet worden.

De moeilijkheid van het vergelijken van algoritmen wordt opgelost door een *taxonomie* van algoritmen voor een gegeven probleem te creëren. Iedere algoritme wordt herschreven in een gemeenschappelijke notatie. Daarnaast wordt bekeken wat de essentiële ingrediënten zijn die de verschillende algoritmen van elkaar onderscheiden. Deze ingrediënten of 'details' kunnen ingedeeld worden in probleem-details (een restrictie op het soort problemen dat opgelost wordt) of in algoritme-details (een correctheid-behoudende transformatie van de algoritme ter verbetering van de efficiency). Zodra een algoritme op een dergelijke manier ontleed is, kan deze gekarakteriseerd worden op basis van de verzameling details. In de taxonomie kunnen de gemeenschappelijke details van twee algoritmen gefactoriseerd en gemeenschappelijk gepresenteerd worden. Op deze manier wordt er een stamboom van algoritmen gecreeërd, die duidelijk aangeeft in welke opzichten verschillende algoritmen met elkaar overeenkomen dan wel van elkaar verschillen. Omdat de wortel van de stamboom een naïeve (en correcte) algoritme is, en de details in een correctheidbehoudende manier toegepast worden, is het correctheidsargument voor iedere algoritme impliciet aanwezig in de taxonomie.

De algemene notatie en presentaties in de taxonomieën stellen ons in staat om de algoritmen op uniforme wijze, in de vorm van een *class library* (ofwel *toolkit*), te implementeren. De factorisering van essentiële details, inherent aan de taxonomieën, leidt tot factorisering van algemene componenten in de inheritance hierarchy van de class library. Object-georiënteerde concepten, zoals virtual (of deferred) member functions, inheritance en template classes, blijken nuttig voor het presenteren van een coherente class library waarvan de structuur de corresponderende taxonomie weergeeft. Voor het eerst worden vrijwel alle oplossingen gepresenteerd in één enkele class library, waardoor de gebruikers van de bibliotheek een grote keuze hebben uit objecten en functies.

Met een class library die bijna alle bekende oplossingen bevat, zijn we eindelijk in staat om gegevens ten aanzien van de praktijkprestaties van de algoritmen te verzamelen. Aangezien de algoritmen afkomstig zijn uit één enkele class library welke geïmplementeerd is door één en dezelfde persoon en de kwaliteit van de implementatie van de class library homogeen is, zijn de relatieve gegevens ten aanzien van de prestaties van de algoritmen niet beïnvloed door de implementatie. Deze gegevens stellen software-ingenieurs in staat om (afhankelijk van hun behoeften en de aard van hun input data) beter gefundeerde keuzen te maken met betrekking tot de in hun applicaties te gebruiken algoritmen en de daarbij behorende objecten en functies.

De ontwikkeling van de taxonomieën heeft bovendien enkele bijprodukten opgeleverd. In ieder van de drie taxonomieën die gepresenteerd worden, zijn belangrijke nieuwe algoritmen ontwikkeld. Deze algoritmen zijn ook geïmplementeerd in de corresponderende class libraries. De technieken ontwikkeld in de taxonomie van de patroonherkenning-algoritmen bleken vooral nuttig te zijn voor de ontwikkeling van een algoritme voor reguliere expressie patroonherkenning, waarmee een oplossing wordt gegeven voor een open probleem gesteld door A.V. Aho in 1980 [Aho80, p. 342].

Curriculum Vitae

I was born on 10 October 1967 in the town of Mutare (Nyika for "the river of ore"), in Eastern Zimbabwe. Until recently, the town was known as Umtali (the Shangane form of Mutare) and the country was known as Rhodesia. After living in Pretoria and Durban (South Africa), and attending school in England for a year, my family and I immigrated to Canada.

From September 1980 to June 1985, I attended Charles Bloom Secondary School (junior high-school) followed by Vernon Secondary School (senior high-school), graduating with Honours in Mathematics and receiving a scholarship to the University of Waterloo on 30 June 1985. From September 1980 until September 1985, I served as a Royal Canadian Air Cadet, rising to the rank of Flt.Sgt., receiving the Duke of Edinburgh Award and the Sir William Bishop Award, and qualifying as a pilot on 25 August 1985.

In September 1985, I became a student of the Faculty of Mathematics at the University of Waterloo in Ontario, Canada. On 26 October 1991, I received the *Bachelor of Mathematics — Honours Joint Combinatorics and Optimization/Computer Science Co-operative Program*, having specialized in compilers and computer architecture.

From January 1986 until August 1990, I held a number of jobs which were relevant to my education and career:

1986 *Software Engineer* at the Canada Land Data System, Environment Canada, Ottawa.

1986 *Programmer-Analyst* at the Canadian Imperial Bank of Commerce, Toronto.

1987-1988 *Software Engineer* at Waterloo Microsystems, Waterloo.

1988 *Software Engineer* at Rockwell, California.

1988-1989 *Team Software Engineer* at the Computer Systems Group, University of Waterloo, Waterloo.

1989-1990 *Compiler Engineer* at Microsoft Corporation, Redmond, Washington.

1990 *Instruction Set Architect* at the Digital Engineering Department, Eindhoven University, Eindhoven.

After a practical-oriented education at Waterloo and sufficient work experience, I applied and was accepted to work towards a Ph.D in one of the best-known computing science faculties: at the Eindhoven University of Technology. From September 1991 until August 1995, I worked with Prof.Dr. F.E.J. Kruseman Aretz, Dr.Ir. Kees Hemerik and Dr.Ir. Gerard Zwaan on developing the taxonomies that would become the cornerstones of this dissertation.

While the development and benchmarking of algorithm implementations began as late night hobbies, they would later become the second and third parts of this thesis research. The toolkits have also become highly popular amongst software engineers and researchers worldwide.

During my four years of study for a Ph.D, I was fortunate enough to travel to a number of other universities and institutes to report on various aspects of my research:

Sept. 1991 The EuroMicro Symposium in Vienna, Austria — *Compiling for a high-level computer architecture.*

Nov. 1993 The Computing Science Netherlands Symposium, Utrecht, The Netherlands — *A taxonomy of keyword pattern matching algorithms* (Best Paper Award).

May 1993 The University of Waterloo, Canada — *A taxonomy of keyword pattern matching algorithms.*

Sept. 1994 The University of Waterloo, Canada — *Algorithms for minimizing deterministic finite automata.*

Oct. 1994 The University of Pretoria, South Africa — *Object-oriented compiler construction* and *Taxonomies and the mathematics of program construction.*

Feb. 1995 Simon Fraser University, Canada — *The minimization of deterministic finite automata.*

Feb. 1995 The University of Victoria, Canada — *A new string pattern matching algorithm.*

May 1995 The SAICSIT 95 Symposium on Research and Development, Pretoria, South Africa — (Invited Keynote Speaker) *Trends in compiler construction* and (Tutorial) *Taxonomies and toolkits: uses for the mathematics of program construction.*

May 1995 The University of Cape Town, South Africa — *A new regular expression pattern matching algorithm.*

Sept. 1995 Universität München, Germany — *Class library implementation methods.*

Bruce W. Watson, 27 July 1995
e-mail: `watson@win.tue.nl`