Towards SPARE time

a new taxonomy and toolkit of keyword pattern matching algorithms

Cleophas, L.G.W.A.

Award date:
2003

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Towards SPARE TIME

A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms

by
L.G.W.A. Cleophas

Supervisors:	prof. dr. B.W. Watson
	dr. ir. G. Zwaan
Advisor:	drs. W. A. A. Nuij

Eindhoven, August 2003

Abstract

We present a new taxonomy and toolkit of keyword pattern matching algorithms. The new taxonomy is an extension of a prior taxonomy of such algorithms. It includes a number of algorithms (including factor- and factor oracle-based and bit-parallel prefix-based pattern matching algorithms) that have been published or received a lot of attention in the last decade.

Based on the new taxonomy, we developed a pattern matching toolkit. This toolkit is a revision and extension of the SPARE PARTS toolkit that had been developed based on the original taxonomy. We present the architecture of the new toolkit, which is named SPARE TIME.

Samenvatting

We presenteren een nieuwe taxonomie en toolkit van algorithmen voor *keyword pattern matching*. De nieuwe taxonomie vormt een uitbreiding van een eerdere taxonomie van zulke algorithmen. Ze bevat een aantal algorithmen (waaronder algorithmen gebaseerd op factoren en factor oracles en bit-parallelle algorithmen gebaseerd op prefixen) die in de afgelopen tien jaar gepubliceerd zijn of veel aandacht gekregen hebben.

Op basis van de nieuwe taxonomie hebben we een *pattern matching toolkit* ontwikkeld. Deze toolkit is een herziene en uitgebreide versie van de SPARE PARTS toolkit die was ontwikkeld op basis van de originele taxonomie. We presenteren de architectuur van deze nieuwe toolkit, genaamd SPARE TIME.

'The man who graduates today
and stops learning tomorrow
is uneducated the day after.'
– Newton D. Baker

Contents

List of Figures

Preface

This document presents the master's thesis for my study *Technische Informatica* at the Technische Universiteit Eindhoven. The research and practical work that is part of this thesis took place from October 2002 to August 2003 within the *Software Construction (SoC)* group of the Department of Mathematics and Computing Science's Division of Computing Science. During this period, I was supervised by Prof. Dr. Bruce W. Watson, head of the group, and Dr. Gerard Zwaan.

Acknowledgements

The research in this thesis started out based on an earlier taxonomy of keyword pattern matching algorithms by Bruce Watson and Gerard Zwaan. As a result, this thesis heavily builds on that work, and I thank them for allowing me to use it as a starting point. The practical part of this thesis, in the form of the SPARE TIME toolkit, is based on the earlier SPARE PARTS toolkit developed by Bruce Watson. I have often used or built on ideas from both the taxonomy and the toolkit, and sometimes reused parts of their papers and thesis on the subject. Wherever I did so, I have indicated this as such.

Bruce Watson allowed me to freely choose the direction in which I wanted to take my work, yet kept coming up with interesting new directions to choose from all the time.

I thank Gerard Zwaan in particular for his meticulous attention to details: although it took me some time to get used to, I am confident that it greatly improved the quality of this thesis and will contribute to the quality of any work I may do in the future as well.

I thank Wim Nuij for joining my thesis committee, especially since my thesis colloquium and defence were planned during the summer period.

Finally, I would like to thank my friends and family for their support during these ten months. In particular, I thank Michiel Frishert for his constructive criticism, our brainstorm sessions—whether in person or over the internet—and reading parts of this thesis, and Jeroen Heijmans for his comments on parts of it as well.

Loek Cleophas
Eindhoven, August 2003

Part I

Preliminaries

Chapter 1

Introduction

In this chapter, we provide the problem statement and its context, as well as an overview of the structure of this thesis.

1.1 Problem statement

One of the oldest and most frequently studied problems in computing science is the *keyword pattern matching* problem. Informally, this is "the problem of finding all occurrences of keywords from a given set as substrings in a given string" ([WZ96]). Among the best known and most used solutions for the problem are such algorithms as Aho-Corasick, Boyer-Moore, Knuth-Morris-Pratt, Commentz-Walter.

Watson and Zwaan (in [WZ96], [Wat95, Chapter 4]) constructed a taxonomy of these algorithms. They showed that the algorithms could all be derived by adding *algorithm details* and *problem details* in some order, starting from a simple high-level algorithm. In addition, Watson constructed a toolkit of C++ implementations of these algorithms ([Wat95, Chapter 9]). The aim of constructing the taxonomy and toolkit was to overcome three deficiencies:

1. The difficulty of comparing the algorithms, due to differences in programming language or style, or due to the addition of unnecessary details.
2. The non-existence of large collections of implementations of such algorithms.
3. The lack of information about practical running time performance of such algorithms.

Their taxonomy and toolkit proved a significant improvement over the situation as it existed before.

In the last ten years or so however, a number of new algorithms has been developed, while other algorithms have gained more and more attention. These include bit-parallel algorithms such as Shift-And, and factor- or factor oracle-based algorithms such as Backward DAWG Matching and Backward Oracle Matching. In this thesis, we extend both the taxonomy and the toolkit so that the first two deficiencies that we described apply even less to the field of keyword pattern matching. Future benchmarking using the extended toolkit can do the same for the third deficiency.

1.2 Thesis structure

This thesis is divided into four parts. The first part contains this chapter and Chapter 2, an overview of the notation and definitions used.

The second part includes the theoretical parts of this thesis. In Chapter 3 the new taxonomy of keyword pattern matching algorithms and the new algorithms in that taxonomy are presented. Each of the algorithms is presented by deriving it from a simple high-level algorithm. This is done by introducing algorithm details—transformations applied to an algorithm—and problem details—restrictions of the problem—in a particular order. Chapter 4 discusses *factor oracles*, particular data structures used in some of the previous chapter's algorithms, in more detail. Two new construction algorithms and some properties regarding the language recognized by factor oracles give more insight into such automata.

Part III discusses the more practical part of the thesis. Chapter 5 discusses general issues in toolkit (re)design and extension, and how these were used to bring the original toolkit SPARE PARTS up to date with respect to present day C++ and Standard Template Library standards and implementations. The extension of this 2003 version of SPARE PARTS to SPARE TIME is then discussed.

The final part of this thesis contains the epilogue, consisting of the conclusions (Chapter 6) and ideas for future work (Chapter 7).

Chapter 2

Notation and definitions

In this chapter we introduce most of the notation and definitions used in this thesis. Some definitions that are only used in small parts of the text, are introduced as needed in the main text instead.

We advise the reader to initially skip this chapter, and refer back to it whenever a notation or definition is unclear or unknown to him or her.

2.1 Notation

Since a large part of this thesis consists of derivations of existing algorithms, we will often use notations corresponding to their use in existing literature on those algorithms. Nevertheless, we tried to adopt the following standard conventions for naming variables, functions and sets whenever possible.

Convention 2.1. The following general naming conventions are used:

- A, B, C for arbitrary sets.
- V for the alphabet.
- a, b, c, d, e for alphabet symbols.
- p, r, s, t, u, v, w, x, y, z for words over alphabet V.
- L, P for languages.
- P, Q for predicates.
- h, i, j, k, l, m, n for integer variables.
- M for finite automata.
- q, r for states, and Q for state sets. Note that states will often also be identified by integer variables.
- δ and γ for automata transition functions.

Sometimes functions, relations or predicates are used that have longer names than just a single character. Subscripts, superscripts, prime symbols etc. are sometimes used as well. In cases where more than one of the above conventions uses the same symbol, the meaning of such a symbol will be clear from the context in which it is used. □

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

2.2 Basic definitions

Notation 2.3 (Quantifications). A basic understanding of the meaning of *quantifications* is assumed. We use the following notation:

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

where $\oplus$ is the associative and commutative *quantification operator* (with unit $e_\oplus$), a is the *dummy variable* introduced, R is the *range predicate* on the dummy, and f is the *quantified expression.* By definition, we have:

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

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

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

□

Notation 2.4 (Natural numbers). We use the symbol $\mathbb{N}$ to denote the set of all natural numbers. For notational convenience, we assume $+\infty, -\infty \in \mathbb{N}$. □

Definition 2.5 (Minimum and maximum). Define **min** and **max** to be infix binary functions on integers such that

$$\begin{aligned} i \,\mathbf{min}\, j &= \mathbf{if}\ i \leq j\ \mathbf{then}\ i\ \mathbf{else}\ j\ \mathbf{fi} \\ i \,\mathbf{max}\, j &= \mathbf{if}\ i \geq j\ \mathbf{then}\ i\ \mathbf{else}\ j\ \mathbf{fi} \end{aligned}$$

Recall from Notation 2.3 that **min** and **max** have $+\infty$ and $-\infty$ as units respectively. □

Property 2.6 (Conjunction and disjunction in MIN quantifications). For predicates P, Q and integer function f we have

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

□

Notation 2.7 (Function signatures). For any two sets A and B, we use $f \in A \to B$ to indicate that f is a total function from A to B. Set A is the *domain* of f while B is the *codomain* or *range* of f. □

Definition 2.8 (Powerset). For any set A we use $\mathcal{P}(A)$ (the *powerset* of A) to denote the set of all subsets of A. □

Definition 2.9 (Precedence of operators). The set operators in order of decreasing precedence are: $\times$, $\cap$, $\cup$. □

Definition 2.10 (Dual of a function). Let R_A and R_B be the reversal operator for sets A and B respectively. Two functions $f \in A \to B$ and $f_d \in A \to B$ are eachother's *dual* if and only if

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

□

Definition 2.11 (Symmetrical function). A *symmetrical function* is one that is its own dual. □

Definition 2.12 (Nondeterministic algorithm). An algorithm is called *nondeterministic* if the order in which (some of) its statements can be executed is not fixed, or if the guards in a selection statement are not mutually exclusive. □

Notation 2.13 (Conditional conjunction/disjunction). We use **cand** and **cor** for *conditional conjunction* and *conditional disjunction* respectively. A conditional conjunction (disjunction) is one in which the second operand is evaluated if and only if this is necessary to determine the value of the conjunction (disjunction). □

2.3 Strings, Languages and Automata

Definition 2.14 (Alphabet). An *alphabet* is a finite non-empty set of *symbols* or *characters*. □

Definition 2.15 (Set of all strings). Given alphabet V, we define V^* to be the set of all strings over V. □

Notation 2.16 (Empty string). We use ε to denote the string of length 0 (the empty string). □

Definition 2.17 (String reversal function R). Assuming alphabet V, we define string reversal function R recursively by $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$ (for $a \in V$, $w \in V^*$). We will use R on sets of strings as well. □

Definition 2.18 (String operators $\uparrow, \downarrow, \lceil, \lfloor$). Assuming alphabet V, we define four infix operators $\uparrow, \downarrow, \lceil, \lfloor \in V^* \times \mathbb{N} \to V^*$ as follows:

- $w\uparrow k$ is the k **min** $|w|$ leftmost symbols of w
- $w\downarrow k$ is the $(|w| - k)$ **max** 0 rightmost symbols of w
- $w\lceil k$ is the k **min** $|w|$ rightmost symbols of w
- $w\lfloor k$ is the $(|w| - k)$ **max** 0 leftmost symbols of w

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

Property 2.19 (String operators $\uparrow, \downarrow, \lceil, \lfloor$). For string operator $\uparrow, \downarrow, \lceil$ and $\lfloor$,

$$(w\uparrow k)(w\downarrow k) = w$$
$$(w\lfloor k)(w\lceil k) = w$$

□

Example 2.20 (String operators $\uparrow, \downarrow, \lceil, \lfloor$). $(hers)\uparrow 3 = her$, $(hers)\downarrow 1 = ers$, $(hers)\lceil 5 = hers$ and $(hers)\lfloor 10 = \varepsilon$. □

Definition 2.21 (Language). Given alphabet V, any subset of V^* is a *language* over V. □

Definition 2.22 (Concatenation of languages). Language concatenation is an infix operator $\cdot \in \mathcal{P}(V^*) \times \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ (the dot) defined as

$$L_1 \cdot L_2 = (\cup x, y : x \in L_1 \wedge y \in L_2 : \{xy\})$$

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

Notation 2.23 (Concatenation of languages). We often use juxtaposition instead of writing operator $\cdot$, i.e. we use L_1L_2 instead of $L_1 \cdot L_2$. For language L and string w, we take Lw to mean $L\{w\}$. □

Definition 2.24 (Functions pref, suff and fact). For any given alphabet V, define **pref** $\in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$, **suff** $\in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ and **fact** $\in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ as

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

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

Property 2.25 (Idempotence of pref, suff and fact). **pref**, **suff** and **fact** are idempotent. □

Property 2.26 (Relationship between fact and suff, pref). Function **fact** can also be defined in terms of the functions **suff** and **pref**:

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

and

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

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

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

□

Property 2.27 (Duality of pref and suff). Functions **pref** and **suff** are each other's duals. This can be seen as follows:

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

□

Property 2.28 (Symmetry of fact). Function **fact** is symmetrical. This can be seen as follows:

$$
\begin{aligned}
& \mathbf{fact}(L^R) \\
\equiv & \quad \{ \text{ Property 2.26 } \} \\
& \mathbf{pref}(\mathbf{suff}(L^R)) \\
\equiv & \quad \{ \text{ Property 2.27 } \} \\
& \mathbf{pref}(\mathbf{pref}(L)^R) \\
\equiv & \quad \{ \text{ Property 2.27 } \} \\
& \mathbf{suff}(\mathbf{pref}(L))^R \\
\equiv & \quad \{ \text{ Property 2.26 } \} \\
& \mathbf{fact}(L)^R
\end{aligned}
$$

□

Notation 2.29 (String arguments to functions pref, suff and fact). For string $w \in V^*$, we will write $\mathbf{pref}(w)$ ($\mathbf{suff}(w)$, $\mathbf{fact}(w)$) instead of $\mathbf{pref}(\{w\})$ ($\mathbf{suff}(\{w\})$, $\mathbf{fact}(\{w\})$). □

Property 2.30 (Languages and pref, suff, fact). For any L, $L \subseteq \mathbf{pref}(L)$, $L \subseteq \mathbf{suff}(L)$ and $L \subseteq \mathbf{fact}(L)$. □

Property 2.31 (Non-empty languages and pref, suff, fact). For any $L \neq \varnothing$, $\varepsilon \in \mathbf{pref}(L)$, $\varepsilon \in \mathbf{suff}(L)$ and $\varepsilon \in \mathbf{fact}(L)$. □

Definition 2.32 (Prefix and suffix partial orderings). Partial orders $\leq_p$, $<_p$, $\leq_s$ and $<_s$ over $V^* \times V^*$ are defined as

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

□

Property 2.33 (Language intersection). If A and B are languages over alphabet V and $a \in V$, then

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

and

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

□

Definition 2.34 ((Deterministic) Finite Automaton). A (deterministic) finite automaton is a 5-tuple $M = \langle Q, V, \delta, q_0, F \rangle$ where

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

□

Definition 2.35 (Extending transition relation δ). We extend transition relation $\delta \in Q \times V \to Q$ to $\delta^* \in Q \times V^* \to Q$ defined by

$$\delta^*(q, \varepsilon) = q$$

and

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

□

Part II

The taxonomy

Chapter 3

A new taxonomy of keyword pattern matching algorithms

3.1 Introduction

The (exact) keyword pattern matching problem can be described as "the problem of finding all occurrences of keywords from a given set as substrings in a given string" ([WZ96]). This problem has been frequently studied in the past, and many different algorithms have been suggested for solving it. Watson and Zwaan (in [WZ96], [Wat95, Chapter 4]) derived a set of well-known solutions to the problem from a common starting point, factoring out their commonalities and presenting them in a common setting to better comprehend and compare them. This leads to a taxonomy of such algorithms as the single-keyword Knuth-Morris-Pratt ([KMP77]) and Boyer-Moore algorithms ([BM77]), as well as the multiple-keyword Aho-Corasick ([AC75]) and Commentz-Walter algorithms ([CW79a, CW79b]).

Although the taxonomy contained a large number of variations on these four basic algorithms, some efficient variants were not included. Among these are the single and multiple keyword Boyer-Moore-Horspool algorithms ([Hor80, NR02]). In addition, bit-parallel algorithms related to the four basic algorithms (such as (Multiple) Shift-And and Shift-Or ([WM92, BYG89])) were not considered. Lastly, a new category of algorithms—based on factors instead of prefixes or suffixes of keywords—has emerged in the last decade. This category includes algorithms such as (Set) Backward DAWG Matching ([CCG+94, NR00]) and (Set) Backward Oracle Matching ([ACR01, AR99]). In this chapter we systematically and formally derive all those algorithms and show how they fit in an extended version of the original taxonomy as presented in [WZ96] and [Wat95].

3.2 The taxonomy

Figure 3.1 shows the new taxonomy, combining the results of [WZ96] and [Wat95, Chapter 4] with the additions and changes discussed in this chapter. This graph can be viewed as an alternative table of contents to this chapter, leading the reader to the algorithms or algorithm families he or she is interested in.

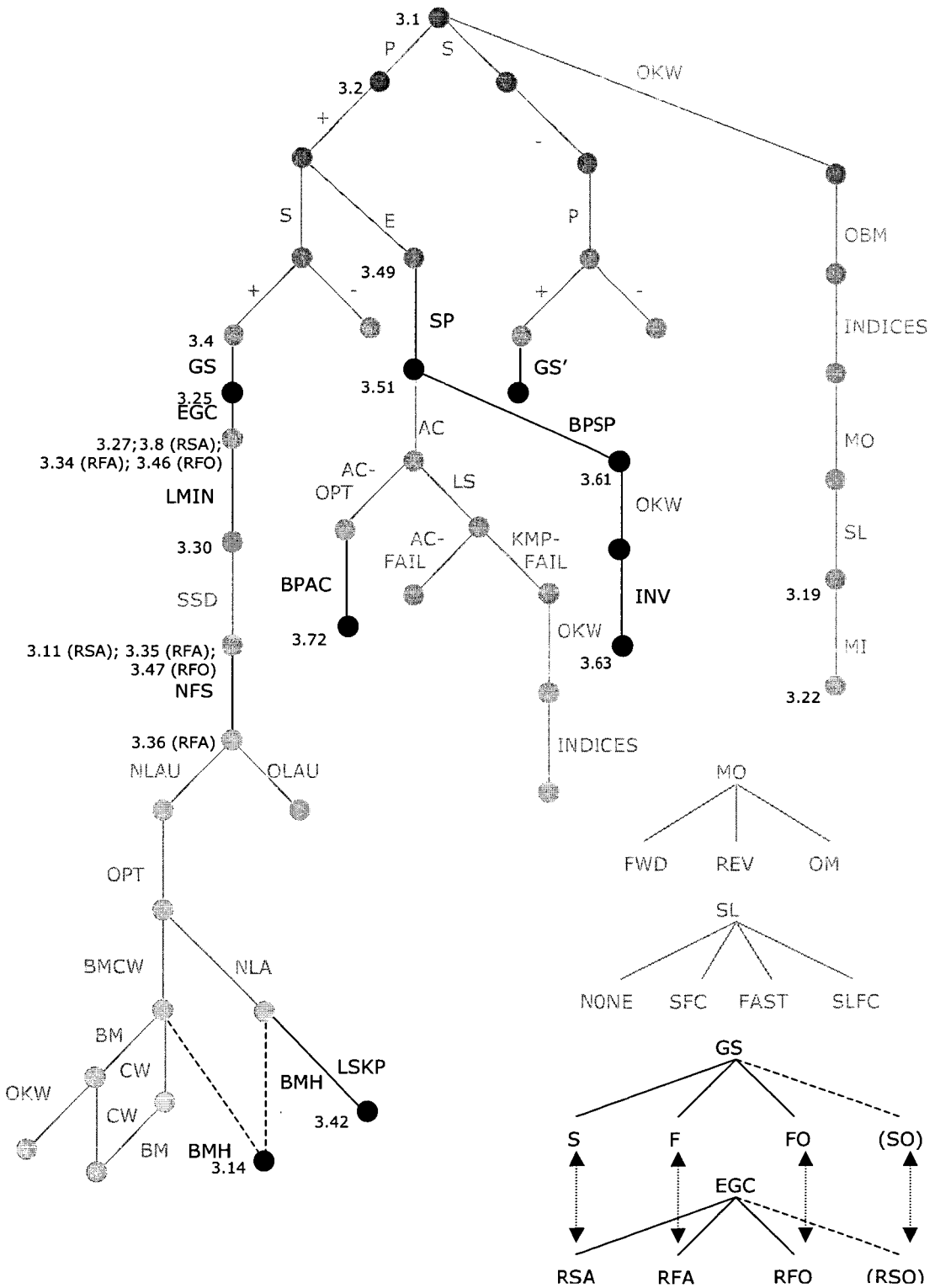

Figure 3.1: A new taxonomy of pattern matching algorithms. Branches and detail names in grey are from the original taxonomy in [WZ96] and [Wat95, Chapter 4]. A list of the algorithm and problem details used plus a short description of each detail is given in Appendix A.

The various algorithms are derived from a common starting point by adding algorithm and problem details. The addition of such a detail results in a new algorithm solving the same original problem or a restriction of that problem. To indicate a particular algorithm and form a taxonomy graph, we use the sequence of details in order of introduction. This method was used in [WZ96, Wat95, Jon83, Mar90] before. We use predicate calculus in our derivations ([DF88, DS90, Kal90]) and present our algorithms in an extended version of Dijkstra's guarded command language ([Dij76]). The extensions that we use are:

- **as** $b \rightarrow S$ **sa** as a shortcut for **if** $b \rightarrow S \; [] \; \neg b \rightarrow$ skip **fi**
- **for** $x : P \rightarrow S$ **rof** for executing statement list S once for each value of x initially satisfying P (assuming that there is a finite number of such values for x), where the order in which values of x are chosen is arbitrary. The **for-rof** statement is taken from [vdE92].

The notation and definitions used are introduced in Chapter 2, unless their use is very local, in which case they are introduced when needed. Most algorithm and problem details will be defined in the course of the text, but a list of the details plus a short description is available in Appendix A as well.

3.3 The problem and some naive solutions

In this section, we start out with a naive solution to the problem and derive more detailed solutions from there. The text in this section is based on—and partially taken from— [WZ96, section 2].

Formally the keyword pattern matching problem, given an alphabet V (a non-empty finite set of symbols), an input string $S \in V^*$, and a finite non-empty pattern set $P = \{p_0, p_1, \ldots p_{|P|-1}\} \subseteq V^*$, is to establish[1]

$$R: \; O = (\bigcup \; l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\}) \, .$$

that is to let O be the set of triples (l, v, r) such that l, v and r form a splitting of S in three parts and the middle part is a keyword in P.

A trivial (but unrealistic) solution to the problem is

Algorithm 3.1()

$O := (\bigcup \; l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\})$
$\{ \; R \; \}$

[1]Note that the problem definition is slightly different but equivalent to that used in [WZ96, Wat95], where $R: \; O = \left(\bigcup \; l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\}\right)$ is used. As a result, the algorithms given in this text will be slightly different in structure but equivalent in meaning to the algorithms of the same name in those texts.

The sequence of details describing this algorithm is the empty sequence (sequences of details are introduced in Section 3.2 and Figure 3.1).

Two basic directions in which to proceed while developing naive algorithms to solve this problem are, informally, to consider a substring of S as "suffix of a prefix of S" or as "prefix of a suffix of S". Only the first possibility is considered here, since the second possibility only leads to algorithms that are the mirror images of algorithms obtained by following the first possibility (basically, it amounts to reversing all strings in the problem). Moreover, this is the way that the algorithms we consider treat substrings of input string S.

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

$$
\begin{aligned}
& (\bigcup\ l,v,r : lvr = S \wedge v \in P : \{(l,v,r)\}) \\
= & \quad \{\text{introduce } u : u = lv\} \\
& (\bigcup\ l,v,r,u : ur = S \wedge lv = u \wedge v \in P : \{(l,v,r)\}) \\
= & \quad \{\text{nesting}\} \\
& (\bigcup\ u,r : ur = S : (\bigcup\ l,v : lv = u \wedge v \in P : \{(l,v,r)\}))
\end{aligned}
$$

A simple non-deterministic algorithm is obtained by applying "examine prefixes of a given string in any order" (algorithm detail (P)) to input string S. It results in

Algorithm 3.2(P)

$O := \varnothing$;
for $(u,r) : ur = S \rightarrow$
    $O := O \cup (\bigcup\ l,v : lv = u \wedge v \in P : \{(l,v,r)\})$
rof{ R }

This algorithm is used as a starting point in Section 3.9 to derive the shift algorithms as well as a bit-parallel version of the Aho-Corasick algorithm. Here, we consider how to update O in the repetition of the preceding algorithm. The update can be computed with another non-deterministic repetition. This inner repetition would consider suffixes of u. Thus by applying "examine suffixes of a given string in any order" (algorithm detail (S)) to string u we obtain algorithm

Algorithm 3.3(P, S)

$O := \varnothing$;
for $(u,r) : ur = S \rightarrow$
    for $(l,v) : lv = u \rightarrow$
        as $v \in P \rightarrow O := O \cup \{(l,v,r)\}$ **sa**
    rof
rof{ R }

Algorithm (P, S) consists of two nested non-deterministic 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. This gives two binary choices. Since the Boyer-Moore, Commentz-Walter and Boyer-Moore-Horspool algorithms examine string S from left to right, and the patterns in P from right to left we focus our attention on the following algorithm:

Algorithm 3.4(P$_+$, S$_+$)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;\ l, v := u, \varepsilon;$
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv$ }
  do $l \neq \varepsilon \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v;$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
od{ R }

This algorithm has running time $\Theta(|S|^2)$, assuming that computing membership of P is a $\Theta(1)$ operation.

3.4 Suffix-based pattern matching

We will now improve the running time of Algorithm 3.4 (P$_+$, S$_+$) by considering the set of suffixes of keywords, **suff**(P). We know that $w \in$ **suff**$(P) \equiv (\exists x : x \in V^* : xw \in P)$. It follows that if $w \notin$ **suff**(P) any extension of w on the left is not an element of **suff**(P) either. Consequently, the inner repetition in Algorithm 3.4 can terminate as soon as $(l\restriction 1)v \notin$ **suff**(P) holds, since then all suffixes of u that are equal to or longer than $(l\restriction 1)v$ are not in **suff**(P) and hence not in P. The inner repetition guard can therefore be strengthened to

$l \neq \varepsilon$ **cand** $(l\restriction 1)v \in$ **suff**(P).

Observe that $v \in$ **suff**(P) is now an invariant of the inner repetition. This invariant is initially established by the assignment $v := \varepsilon$ since $P \neq \varnothing$ and thus $\varepsilon \in$ **suff**(P). Direct evaluation of $(l\restriction 1)v \in$ **suff**(P) is expensive. Therefore, it is done using the transition function $\delta_{R,\mathbf{suff},P}$ of a finite automaton recognizing **suff**$(P)^R$, where $\delta_{R,\mathbf{f},P}$ has the following properties:

Property 3.5 (Transition function of automaton recognizing f$(P)^R$). The transition function $\delta_{R,\mathbf{f},P}$ of a (weakly deterministic) finite automaton $M = \langle Q, V, \delta_{R,\mathbf{f},P}, q_0, F\rangle$ recognizing $\mathbf{f}(P)^R$ has the property that

$\delta^*_{R,\mathbf{f},P}(q_0, w^R) \neq \perp \equiv w^R \in \mathbf{f}(P)^R$

and we assume

$$\delta_{R,\mathbf{f},P}(q,\varepsilon) = q$$

□

Note that Property 3.5 is possible only if $\mathbf{pref}(\mathbf{f}(P)^R) \subseteq \mathbf{f}(P)^R$, i.e. if $\mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$. Also note that $w^R \in \mathbf{f}(P)^R \equiv w \in \mathbf{f}(P)$.

Since we will always refer to the same set P in the remainder of this document, we will use $\delta_{R,\mathbf{f}}$ instead of $\delta_{R,\mathbf{f},P}$.

Transition function $\delta_{R,\mathbf{suff}}$ can be computed beforehand, as in [WZ96, subsection 4.1][2].

Algorithm detail 3.6. (GS=S). (Guard Strengthening = Suffix). Strengthen the guard of the inner repetition by adding conjunct $(l\restriction 1)v \in \mathbf{suff}(P)$. □

By making $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$ an invariant of the inner repetition of the algorithm, we can use the following algorithm detail:

Algorithm detail 3.7. (EGC=RSA). (Efficient Guard Computation = Reverse Suffix Automaton). Given a finite automaton recognizing $\mathbf{suff}(P)^R$ and satisfying Property 3.5, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{suff}(P)$ then becomes $q \neq \bot$. □

These two algorithm details are equivalent to the introduction of algorithm detail (RT) in [WZ96, Wat95], where the first detail (the strengthening of the guard) is introduced implicitly. We replace its name by two separate algorithm detail names here in anticipation of the generalization of guard strengthening and efficient guard computation in Section 3.6.

Algorithm 3.8(P$_+$, S$_+$, GS=S, EGC=RSA)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\lfloor 1;\ l, v := u, \varepsilon;\ q := \delta_{R,\mathbf{suff}}(q_0, l\restriction 1);$
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv \wedge v \in \mathbf{suff}(P) \wedge q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \bot \rightarrow$
    $l, v := l\lfloor 1, (l\restriction 1)v;$
    $q := \delta_{R,\mathbf{suff}}(q, l\restriction 1);$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
  { $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{suff}(P)$ }
od{ R }

[2] In [WZ96], the transition function is called τ_P. It is called $\tau_{P,r}$ in [Wat95] to distinguish it from the *forward trie* function. We generalize the function by means of a parameter $\mathbf{f}$ and make it into a transition function on automata in anticipation of its use in Section 3.6.

Assuming P is constant, $(\mathbf{MAX}\, p : p \in P : |p|)$ is constant and this algorithm has $\Theta(|S|)$ running time[3]. It will serve as a starting point for the derivation of the algorithms in the following two sections.

3.4.1 Suffix-based sublinear pattern matching

In section 3 of [WZ96], a family of sublinear keyword pattern matching algorithms is derived starting from Algorithm 3.8 (P$_+$, S$_+$, GS=S, EGC=RSA). The basic idea is to make shifts of more than one symbol. This is accomplished by replacing $u, r := u(r\uparrow 1), r\downarrow 1$ by $u, r := u(r\uparrow k), r\downarrow k$ for some k satisfying $1 \leq k \leq (\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing : n)$. The upperbound is the distance to the next match, the maximal safe shift distance. Any smaller number k satisfying the equation is safe as well, and we thus define a safe shift distance as:

Definition 3.9 (Safe shift distance). A shift distance k satisfying

$$1 \leq k \leq (\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing : n)$$

is called a *safe shift distance.* □

Algorithm detail 3.10. (SSD). Replace assignment

$$u, r := u(r\uparrow 1), r\downarrow 1$$

in Algorithm 3.8(P$_+$, S$_+$, GS=S, EGC=RSA) by assignment

$$u, r := u(r\uparrow k), r\downarrow k$$

using a safe shift distance k. □

In [WZ96], various approximations from below of the maximal safe shift distance are derived by weakening the predicate $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$. This results in safe shift distances that are easier to compute than the maximal safe shift distance. In these derivations, the $u = lv \wedge v \in \mathbf{suff}(P)$ part of the invariant of the inner repetition in Algorithm 3.8 is used. By adding $l, v := \varepsilon, \varepsilon$ to the initial assignments of the algorithm, we turn this into an invariant of the outer repetition. This also turns $l = \varepsilon$ **cor** $(l\uparrow 1)v \notin \mathbf{suff}(P)$—the negation of the guard of the inner repetition—into an invariant of the outer repetition. Since shift functions may depend on l, v and r, we will write $k(l, v, r)$. Hence, we arrive at the following algorithm skeleton:

Algorithm 3.11(P$_+$, S$_+$, GS=S, EGC=RSA, SSD)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
$l, v := \varepsilon, \varepsilon;$
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$
$\wedge\ u = lv \wedge v \in \mathbf{suff}(P)$
$\wedge\ (l = \varepsilon$ **cor** $(l\uparrow 1)v \notin \mathbf{suff}(P))$ }

[3]We will from now on use this assumption on P when discussing running time of algorithms.

do $r \neq \varepsilon \rightarrow$
$u, r := u(r\restriction k(l,v,r)), r\downarrow k(l,v,r);\ l, v := u, \varepsilon;\ q := \delta_{R,\mathbf{suff}}(q_0, l\restriction 1);$
as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
{ invariant: $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$ }
do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
$l, v := l\downarrow 1, (l\restriction 1)v;$
$q := \delta_{R,\mathbf{suff}}(q, l\restriction 1);$
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
od{ R }

Based on this algorithm skeleton, various shift functions are derived in [WZ96][4], leading among others to the Commentz-Walter, Fu-San and multiple keyword Boyer-Moore algorithms.

3.4.2 The multiple-keyword Horspool algorithm

In this section, we consider two particular weakenings of the range predicate $\mathbf{suff}(u(r\restriction n))\cap P \neq \varnothing$:

$$V^*vV^n \cap V^*P \neq \varnothing$$

(used in algorithm detail (NLA) given in [WZ96, section 3.7]) and

$$V^*(l\restriction 1)vV^n \cap V^*P \neq \varnothing$$

(used in [WZ96, section 3.2, page 13]). Note that the new predicates only refer to l and v, but not to r. Informally, this amounts to discarding any right lookahead.

Convention 3.12 (Shift function signatures). Whenever r is not used in a shift function, we will use $k(l, v)$ instead of $k(l, v, r)$. □

We now further weaken the first predicate, assuming $v \neq \varepsilon$:

$$\begin{array}{ll} & V^*vV^n \cap V^*P \neq \varnothing \\ \Rightarrow & \quad \{ v = (v\downarrow 1)(v\restriction 1) \} \\ & V^*(v\downarrow 1)(v\restriction 1)V^n \cap V^*P \neq \varnothing \\ \Rightarrow & \quad \{ v\downarrow 1 \in V^* \} \\ & V^*(v\restriction 1)V^n \cap V^*P \neq \varnothing \end{array}$$

We now further weaken the second predicate, assuming $v = \varepsilon$:

[4]The algorithm skeleton is called (P_+, S_+, RT, SSD) there.

$$
\begin{array}{ll}
& V^*(l\restriction 1)vV^n \cap V^*P \neq \varnothing \\
\Rightarrow & \quad \{ v = \varepsilon \} \\
& V^*(l\restriction 1)V^n \cap V^*P \neq \varnothing
\end{array}
$$

Note the close resemblance between the two weakened predicates: the only difference is that the first refers to $(v\restriction 1)$ (assuming $v \neq \varepsilon$) whereas the second refers to $(l\restriction 1)$ (assuming $v = \varepsilon$). Using these predicates (depending on whether $v = \varepsilon$ or $v \neq \varepsilon$) we get a practical safe shift distance. We show the case $v \neq \varepsilon$ (i.e. $(v\restriction 1)$ occurs in the predicate) here:

$$
\begin{array}{ll}
& (\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing : n) \\
\geq & \quad \{ \text{ weakening steps above } \} \\
& (\mathbf{MIN}\, n : 1 \leq n \wedge (V^*(v\restriction 1)V^n \cap V^*P \neq \varnothing) : n) \\
= & \quad \{ \text{ definition } char_{bm} \text{ ([WZ96, section 3.4] } \} \\
& char_{bm}(v\restriction 1)
\end{array}
$$

To get to a safe shift function, we need $l \neq \varepsilon$ to hold in case $v = \varepsilon$. Using the above derivation, we could then use $char_{bm}(l\restriction 1)$ as a safe shift distance. Note that in Algorithm 3.11 (P+, S+, GS=S, EGC=RSA, SSD), $l \neq \varepsilon$ does not hold initially. Assuming $\varepsilon \notin P$, we can solve this by changing the initialization to $u, r := S\restriction lmin_P, S\downarrow lmin_P$ (where $lmin_P = (\mathbf{MIN}\, p : p \in P : |p|)$). This then allows the use of

Definition 3.13 (Shift function k_{bmh}). Shift function k_{bmh} is defined as:

$$
k_{bmh}(l, v) = \begin{cases} char_{bm}(v\restriction 1) & \text{if } v \neq \varepsilon, \\ char_{bm}(l\restriction 1) & \text{if } v = \varepsilon. \end{cases}
$$

□

Algorithm detail 3.14. (BMH). (Boyer-Moore-Horspool). Calculating the shift distance using k_{bmh} is algorithm detail (BMH). □

The use of shift function k_{bmh} yields algorithm (P+, S+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BMH)[5], the Set Horspool algorithm ([NR02, subsection 3.3.2]). Adding problem detail (OKW) leads to the single-keyword Horspool algorithm ([NR02, subsection 2.3.2], [Hor80]).

Remark 3.15. In Section 3.5, we will derive the single-keyword Horspool algorithm in a different way. □

[5]The characterization of the algorithm is debatable, as it can be seen as a member of the algorithm family (P+, S+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW) in case $v = \varepsilon$ and as a further development of algorithm (P+, S+, GS=S, EGC=RSA, SSD, NLAU, OPT, NLA) in case $v \neq \varepsilon$.

3.5 The single-keyword Horspool algorithm

In Section 3.4.2, we derived the Boyer-Moore-Horspool algorithm from algorithm (P$_+$, S$_+$, GS=S, EGC=RSA, SSD), the same algorithm that gives rise to the Commentz-Walter multiple-keyword pattern matching algorithm family (see [WZ96, section 3]). Horspool originaly came up with his algorithm as a simplification of the single-keyword Boyer-Moore algorithm, and we therefore show that it is possible to describe a single-keyword version of the algorithm based on the Boyer-Moore algorithm (OKW, OBM, INDICES, MO, SL=FAST) (see [Wat95, section 4.5]). Before we give this algorithm, we first give a number of definitions (from [Wat95]) in order to make it more readable:

Definition 3.16 (Perfect match predicate *PerfMatch*). [6] 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*. □

Function *shift* shifts the position of v in S to the right by k symbols, extending l to the right by k symbols and removing the k leftmost characters from r:

Definition 3.17 (Shift function *shift*). Define right shift function $shift \in (V^*)^3 \times \mathbb{N} \rightarrow (V^*)^3$ by

$$shift(l, v, r, k) = (l(vr{\uparrow}k), (v(r{\uparrow}k)){\downarrow}k, r{\downarrow}k)$$

□

The single-keyword Boyer-Moore algorithm variants in Section 4.5 of [Wat95] and in this section are quite different from the multi-keyword versions derived in Section 4.4 of that text and Section 3.4.2 of this thesis.

The algorithms in [Wat95, Section 4.5] use a skip loop in order to make a shift before a match attempt is made, and a shift after the match attempt. In addition, the order in which characters of v and p are matched is not necessarily right-to-left.

As a result, the following algorithm not only uses details (OKW), (OBM) and (INDICES) to arrive at a general single-keyword Boyer-Moore algorithm, but adds (MO) and (SL=FAST) for the match order (*mo*) and skip loop details respectively. We do not discuss these algorithm details any further here, but refer the reader to Appendix A for a brief description of the details, and to [Wat95] for a more detailed discussion.

The actual safe shift function used in the following algorithm is function sl_1 as in [Wat95, Definition 4.163]:

Definition 3.18 (Function sl_1). Given $j : 1 \leq j \leq |p|$, we can define function $sl_1 \in V \rightarrow \mathbb{N}$ by

$$sl_1(a) = (\mathbf{MIN}\, k : 1 \leq k \wedge (1 + k \leq j \Rightarrow a = p_{j-k}) : k)$$

Note that sl_1 depends implicitly on j. □

[6] Although a name like *Match* or *CompleteMatch* might have been more clear, we use the original name here.

Algorithm 3.19(OKW, OBM, INDICES, MO, SL=FAST)

$l, v, r := \varepsilon, S1|p|, S\lfloor|p|;\ O := \varnothing;$
{ invariant: $lvr = S \wedge O = (\bigcup x, y, z : PerfMatch((x, y, z)) \wedge (xy <_p lv) : \{(x, y, z)\})$
$\wedge\ (|v| \leq |p|) \wedge (|v| < |p| \Rightarrow r = \varepsilon)$ }
do $|v| = |p| \rightarrow$
{ $|v| = |p|$ }
do $1 \leq |r| \wedge \neg(v_{|p|} = p_{|p|}) \rightarrow$
$(l, v, r) := shift(l, v, r, sl_1(v_{|p|})$ **min** $|r|)$
od;
{ $|v| = |p| \wedge (v_{|p|} = p_{|p|} \vee r = \varepsilon)$ }
$i := match(v, p, mo)$;
{ $(1 \leq i \leq |p| + 1)$
$\wedge\ (i \leq |p| \Rightarrow v_{mo(i)} \neq p_{mo(i)} \wedge \big(\forall j : 1 \leq j < i : v_{mo(j)} = p_{mo(j)}\big))$
$\wedge\ (i = |p| + 1 \Rightarrow v = p)$ }
as $i = |p| + 1 \rightarrow O := O \cup \{(l, v, r)\}$ **sa**;
$(l, v, r) := shift(l, v, r, 1)$;
od
{ $O = (\bigcup l, v, r : PerfMatch((l, v, r)) : \{(x, y, z)\})$ }

Remark 3.20. Note that we do not give an implementation for function *match*, but that it is sufficiently specified by the annotation of the above algorithm. See [Wat95, p. 103] for a possible implementation. □

In the above algorithm, the shift distance at the end of the outer loop is still equal to 1, that is, no information from the previous match attempt is used to compute a greater shift. In [Wat95, Subsection 4.5.2], algorithm detail (MI) (for Match Information) is introduced to make use of this information to get larger shifts. As is done there, and similar to what we saw in Subsection 3.4.1 for the multiple keyword Boyer-Moore-Horspool algorithm, we now want to compute a safe shift distance based on the maximal safe shift distance

$$(\mathbf{MIN}\, k : 1 \leq k \leq |r| \wedge PerfMatch(shift(l, v, r, k)) : k)$$

We weaken the range predicate:

$$\begin{array}{ll} & PerfMatch(shift(l, v, r, k)) \\ \Rightarrow & \quad \{ \text{ derivation of [Wat95, page 105] } \} \\ & (\forall h' : 1 + k \leq h' \leq |p| : v_{h'} = p_{h'-k}) \\ \Rightarrow & \quad \{ \text{ domain split, one-point rule} — h' = |p| \} \\ & 1 + k \leq |p| \Rightarrow v_{|p|} = p_{|p|-k} \end{array}$$

Remark 3.21. The last predicate in the preceding derivation is related to predicate I_3'' : $(i \leq |p|$ **cand** $1 + k \leq mo(i) \Rightarrow v_{mo(i)} = p_{mo(i)-k})$ given on page 109 of [Wat95]. This makes

sense, since this predicate leads to auxiliary function $char_1(i)$ of the Boyer-Moore algorithm, on which Horspool based his algorithm.[7] □

Based on the above derivation, a safe shift distance is:

$$\Big(\mathbf{MIN}\, k : 1 \leq k \leq |p| \wedge (1 + k \leq |p| \Rightarrow v_{|p|} = p_{|p|-k}) : k\Big)$$

This safe shift distance equals $sl_1(v_{|p|})$ which we previously defined.

Using function sl_1 can be seen as another instance of algorithm detail (MI) (Match Information) (see Appendix A). We have thus shown that the single-keyword Horspool algorithm can be viewed as a variation on the Boyer-Moore algorithm skeleton (OKW, OBM, INDICES, MO, SL, MI):

Algorithm 3.22(OKW, OBM, INDICES, MO, SL=FAST, MI=BMH)

$l, v, r := \varepsilon, S\uparrow|p|, S\downarrow|p|;\ O := \varnothing;$
{ invariant: $lvr = S \wedge O = (\bigcup x, y, z : PerfMatch((x, y, z)) \wedge (xy <_p lv) : \{(x, y, z)\})$
$\wedge\ (|v| \leq |p|) \wedge (|v| < |p| \Rightarrow r = \varepsilon)$ }
do $|v| = |p| \rightarrow$
{ $|v| = |p|$ }
do $1 \leq |r| \wedge \neg(v_{|p|} = p_{|p|}) \rightarrow$
$(l, v, r) := shift(l, v, r, sl_1(v_{|p|}))$ **min** $|r|$;
od;
{ $|v| = |p| \wedge (v_{|p|} = p_{|p|} \vee r = \varepsilon)$ }
$i := match(v, p, mo)$;
{ $(1 \leq i \leq |p| + 1)$
$\wedge\ (i \leq |p| \Rightarrow v_{mo(i)} \neq p_{mo(i)} \wedge \Big(\forall j : 1 \leq j < i : v_{mo(j)} = p_{mo(j)}\Big))$
$\wedge\ (i = |p| + 1 \Rightarrow v = p)$ }
as $i = |p| + 1 \rightarrow O := O \cup \{(l, v, r)\}$ **sa**;
$(l, v, r) := shift(l, v, r, sl_1(v_{|p|}))$;
od
{ $O = (\bigcup l, v, r : PerfMatch((l, v, r)) : \{(x, y, z)\})$ }

[7]Note that the predicate is also related to predicate $1 + k \leq j \Rightarrow v_j = p_{j-k}$ given at the top of page 106 of [Wat95]. In fact it is the equivalent of that predicate when using $J_2((l, v, r)) \equiv v_{|p|} = p_{|p|}$ instead of $J_3((l, v, r)) \equiv (v_j = p_j)$ (for some $j : 1 \leq j \leq |p|$) as a weakening of *PerfMatch* (see [Wat95, pages 105,107]). This is why we have used predicate J_2 (i.e. choice (FAST) for algorithm detail (SL)) in the skiploop of the algorithms in this section.

3.6 A generalization of suffix-based algorithms

We observed in Section 3.3 that it is possible to strengthen the guard of the inner loop ($l \neq \varepsilon$) to $l \neq \varepsilon$ **cand** $(l\restriction 1)v \in \mathbf{suff}(P)$. This then led us to introduce the automaton transition function $\delta_{R,\mathbf{suff}}$ to easily compute $(l\restriction 1)v \in \mathbf{suff}(P)$. The use of this function leads to Algorithm 3.8 (P$_+$, S$_+$, GS=S, EGC=RSA), which we repeat here for easy reference:

Algorithm 3.23(P$_+$, S$_+$, GS=S, EGC=RSA)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;\ l, v := u, \varepsilon;\ q := \delta_{R,\mathbf{suff}}(q_0, l\restriction 1);$
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv \wedge v \in \mathbf{suff}(P) \wedge q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v;$
    $q := \delta_{R,\mathbf{suff}}(q, l\restriction 1);$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
  { $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{suff}(P)$ }
od{ R }

Since $w \in P \Rightarrow \mathbf{suff}(w) \subseteq \mathbf{suff}(P)$, $\mathbf{suff}(w) \not\subseteq \mathbf{suff}(P) \Rightarrow w \notin P$. The essential properties of **suff** that we use here are that $P \subseteq \mathbf{suff}(P)$ and $\mathbf{suff}(\mathbf{suff}(P)) \subseteq \mathbf{suff}(P)$ (*suffix-closedness*). Hence, we can use other functions **f** satisfying $P \subseteq \mathbf{f}(P)$ and $\mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$ to strengthen guard $l \neq \varepsilon$. Since $(l\restriction 1)v \notin \mathbf{f}(P) \Rightarrow (l\restriction 1)v \notin P$, $(l\restriction 1)v \notin \mathbf{f}(P) \Rightarrow w(l\restriction 1)v \notin \mathbf{f}(P) \Rightarrow w(l\restriction 1)v \notin P$ (for every w), the guard can be strengthened to $l \neq \varepsilon \wedge (l\restriction 1)v \in \mathbf{f}(P)$[8]. We thus introduce algorithm detail (GS):

Algorithm detail 3.24. (GS). (Guard Strengthening). Strengthening the inner repetition guard $l \neq \varepsilon$ to $l \neq \varepsilon$ **cand** $(l\restriction 1)v \in \mathbf{f}(P)$ for function $\mathbf{f} \in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ satisfying $P \subseteq \mathbf{f}(P)$ and $\mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$. □

This algorithm detail leads to the following algorithm skeleton:

Algorithm 3.25(P$_+$, S$_+$, GS)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;\ l, v := u, \varepsilon;$

[8]This is the reason why we replaced detail (RT) by detail sequence (GS=S, EGC=RSA) in Section 3.4.

as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
{ invariant: $u = lv \wedge v \in \mathbf{f}(P)$ }
do $l \neq \varepsilon$ **cand** $(l\restriction 1)v \in \mathbf{f}(P) \rightarrow$
$l, v := l\downarrow 1, (l\restriction 1)v$;
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
{ $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{f}(P)$ }
od{ R }

Several choices for function **f** are possible, of which we mention the following:

- **suff**. This choice was discussed in Section 3.3 (and more extensively in [WZ96, Wat95]).
- **fact**. We discuss this choice in Section 3.7.
- A function that returns a superset of **fact**. An implementation of this choice using factor oracles is discussed in Section 3.8.
- A function that returns a superset of **suff**. This could be implemented using **sufforacle**, i.e. the function defining the language recognized by a *suffix oracle*([ACR01, AR99]) on a set of keywords. We will not explore this option in this thesis.[9]

In order to easily compute the conjunct $(l\restriction 1)v \in \mathbf{f}(P)$ introduced by algorithm detail (GS), an automaton is often introduced recognizing the language $\mathbf{f}(P)^R$. This leads to

Algorithm detail 3.26. (EGC). (Efficient Guard Computation). Given a finite automaton recognizing $\mathbf{f}(P)^R$ and satisfying Property 3.5, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{f}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{f}(P)$ then becomes $q \neq \perp$. □

This algorithm detail leads to the following algorithm skeleton:

Algorithm 3.27(P$_+$, S$_+$, GS, EGC)

$u, r := \varepsilon, S$;
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \,[\!]\, \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
$u, r := u(r\restriction 1), r\downarrow 1$; $l, v := u, \varepsilon$; $q := \delta_{R,\mathbf{f}}(q_0, l\restriction 1)$;
as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon; r)\}$ **sa**;
{ invariant: $u = lv \wedge v \in \mathbf{f}(P) \wedge q = \delta^*_{R,\mathbf{f}}(q_0, ((l\restriction 1)v)^R)$ }
do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
$l, v := l\downarrow 1, (l\restriction 1)v$;
$q := \delta_{R,\mathbf{f}}(q, l\restriction 1)$;

[9]Since the language of a suffix oracle—like the language of a factor oracle—has not yet been defined independent of the automaton, it would be necessary to construct the suffix oracle to get a working algorithm. The construction of a suffix oracle is more complicated and less memory efficient than that of a factor oracle however. See [ACR01, AR99] for details.

as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
{ $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{f}(P)$ }
od{ R }

We have seen one particular choice of this detail, (EGC=RSA), in Section 3.3. The other choices will be discussed together with the corresponding choices for detail (GS), i.e. in Sections 3.7 and 3.8. Note that guard $v \in P$ can be efficiently computed, i.e. computed in $\Theta(1)$, in this and following algorithms by providing a map from states of the automaton to a boolean.[10]

3.6.1 A change leading to smaller automata

In practice, the multiple-keyword algorithms using automata (among them the algorithm family (P_+, S_+, GS, EGC)) often use automata recognizing $\mathbf{f}(P')^R$ where $P' = \{v : v \in \mathbf{pref}(P) \wedge |v| = lmin_P\}$ instead of $\mathbf{f}(P)^R$. Informally, an automaton is built on the prefixes of length $lmin_P$, in order to obtain smaller automata.

Algorithm detail 3.28. (LMIN). The automaton used in algorithm detail (EGC) is built on $\mathbf{f}(P')$ where $P' = \{w : w \in \mathbf{pref}(P) \wedge |w| = lmin_P\}$ instead of on $\mathbf{f}(P)$. □

Remark 3.29. Note that this algorithm detail could be applied to any of the pattern matching algorithms in the taxonomy shown in figure 3.1. □

As a result of using algorithm detail (LMIN) with Algorithm 3.27 (P_+, S_+, GS, EGC), after assignment $l, v := l\lfloor 1, (l\restriction 1)v$ in the inner loop, $v \in \mathbf{f}(P')$ holds (instead of $v \in \mathbf{f}(P)$ as before). Due to Property 3.5, in case $|v| = lmin_P$ (i.e. $v \in P'$) we need to verify any matches $v(r\restriction i) \in P$ for $i \leq lmax_P - lmin_P$ (where $lmax_P = (\mathbf{MAX}\, p : p \in P : |p|)$). Since there is a longest keyword, we do not need to increase i past the mentioned maximum value. This leads to the following algorithm skeleton (where details (GS) and (EGC) still need to be instantiated):

Algorithm 3.30(P_+, S_+, GS, EGC, LMIN)

$u, r := \varepsilon, S$;
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \,[\!]\, \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\lfloor 1$; $l, v := u, \varepsilon$; $q := \delta_{R,\mathbf{f}}(q_0, l\restriction 1)$;
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv \wedge v \in \mathbf{f}(P') \wedge q = \delta^*_{R,\mathbf{f}}(q_0, ((l\restriction 1)v)^R)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
    $l, v := l\lfloor 1, (l\restriction 1)v$;
    $q := \delta_{R,\mathbf{f}}(q, l\restriction 1)$;

[10]The construction of such a map may require quite some precomputation time. We do not consider the relative precomputation times of the various algorithms, since they are both relatively hard to compare in terms of $\mathcal{O}$ notation and are assumed to be relatively small compared to the time taken to perform the actual pattern matching (i.e. the text on which the matching is performed is assumed to be relatively long).

```
    od;
    { l = ε cor (l↑1)v ∉ f(P') }
    as |v| = lmin_P →
        w, s := v, r;
        as w ∈ P → O := O ∪ {(l, w, s)} sa;
        do |w| ≠ lmax_P ∧ s ≠ ε →
            w, s := w(s↑1), s↓1;
            as w ∈ P → O := O ∪ {(l, w, s)} sa
        od
    sa
od{ R }
```

This algorithm also has $\Theta(|S|)$ running time, assuming P (and thus $(\mathbf{MAX}p : p \in P : |p|)$) to be constant.

Remark 3.31. It is possible to improve the last part of this algorithm by introducing a forward trie function (as in [Wat95, subsection 4.2.2]), assuming that the initialization to $\delta^*(q_0, v)$ in the forward trie is a $\Theta(1)$ operation. This can be achieved for example by having a mapping between each element of $\mathbf{f}(P')$ of length $lmin$ and the corresponding state of the forward trie. Since $|v| = lmin_P$ always holds when the forward trie is used, it might be possible to only construct the parts of the trie of depth $lmin_P$ or greater. We do not further discuss this option. □

The use of algorithm detail (LMIN) has the following effects:

- Reduced size of automaton: Since $lmin_P$ might be less than $|p_i|$ for some i, the automaton might have less states and less transitions. This gain may (partially) be offset by the time spent executing the new **as** $|v| = lmin_P \rightarrow \ldots$ **sa** statement, or by the space and time spent when introducing the forward trie as in Remark 3.31.

- Reduced maximal shift distances with detail (SSD): Since $|v|$ might be less than $|p_i|$ (for i such that $|p_i| \geq lmin_P$), there is less information from v that can be used. Hence shift distances might be smaller, leading to a larger total number of shifts.

- Reduced number of character comparisons: Let there be keywords $p_i, p_j \in P$, such that $p_j \neq p_i$ and $p_j \in \mathbf{pref}(p_i)$. In this case, using detail (LMIN) it will take less comparisons to verify a match of both keywords. Originally, for an occurrence of p_i in S, $|p_i| + |p_j|$ comparisons are needed to detect both matches. Using detail (LMIN), $lmin + (|p_i| - lmin) = |p_i|$ comparisons are needed.

- Increased number of character comparisons: Let there be an occurrence of $u \in \mathbf{f}(P')$ in S such that $u \notin \mathbf{f}(P)$, $|u| + 1$ character comparisons will be made (assuming that $au \notin \mathbf{f}(P')$, with a the next character in S following the occurrence of u). When not using detail (LMIN), less than $|u| + 1$ comparisons will be made.

The effects thus depend on the set of keywords P and the text S.

3.7 Factor-based pattern matching

We now derive a family of algorithms by using the set of factors of P, $\mathbf{fact}(P)$. We introduce

Algorithm detail 3.32. (GS=F). (Guard strengthening = Factor). Strengthen the guard of the inner repetition by adding conjunct $(l\restriction 1)v \in \mathbf{fact}(P)$. □

The above guard strengthening may be used because $P \subseteq \mathbf{fact}(P)$ and $\mathbf{suff}(\mathbf{fact}(P)) \subseteq \mathbf{fact}(P)$. The inner repetition guard then becomes

$$l \neq \varepsilon \textbf{ cand } (l\restriction 1)v \in \mathbf{fact}(P)$$

As with $(l\restriction 1)v \in \mathbf{suff}(P)$ before, direct evaluation of $(l\restriction 1)v \in \mathbf{fact}(P)$ is expensive. Instead, we will use the transition function of an automaton recognizing the set $\mathbf{fact}(P)^R$. Using function $\delta_{R,\mathbf{fact}}$ introduced in Section 3.4 and making $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\restriction 1)v)^R)$ an invariant of the inner repetition, the guard becomes

$$l \neq \varepsilon \textbf{ cand } q \neq \bot$$

Algorithm detail 3.33. (EGC=RFA). (Efficient Guard Computation = Reverse Factor Automaton). Given a finite automaton recognizing $\mathbf{fact}(P)^R$ and satisfying Property 3.5, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{fact}(P)$ then becomes $q \neq \bot$. □

Note that various automata exist whose transition functions can be used for $\delta_{R,\mathbf{fact}}$. One is the trie built on $\mathbf{fact}(P)^R$, another is the *suffix automaton* or *dawg* (for directed acyclic word graph) on $\mathbf{fact}(P)^R$ ([CR94, CH97]).

The use of algorithm detail (EGC=RFA) leads to

Algorithm 3.34(P_+, S_+, GS=F, EGC=RFA)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;\ l, v := u, \varepsilon;\ q := \delta_{R,\mathbf{fact}}(q_0, l\restriction 1);$
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv \wedge v \in \mathbf{fact}(P) \wedge q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\restriction 1)v)^R)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \bot \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v;$
    $q := \delta_{R,\mathbf{fact}}(q, l\restriction 1);$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
  { $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{fact}(P)$ }
od{ R }

This algorithm has $\Theta(|S|)$ running time, just like Algorithm 3.8 (P_+, S_+, GS=S, EGC=RSA). The use of detail sequence (GS=F, EGC=RFA) instead of (GS=S, EGC=RSA) has the following effects:

- More character comparisons: In cases where $(l\restriction 1)v \notin \mathbf{suff}(P)$ yet $(l\restriction 1)v \in \mathbf{fact}(P)$, the guard of the inner loop will still be true, and hence the algorithm will go on extending v to the left more than strictly necessary.
- Larger shift distances with detail (SSD): When the guard of the inner loop becomes false, $(l\restriction 1)v \notin \mathbf{fact}(P)$, which gives potentially more information to use in the shift function than $(l\restriction 1)v \notin \mathbf{suff}(P)$. This aspect will be used in the derivations leading to algorithm detail (NFS) in Subsection 3.7.1.

3.7.1 Factor-based sublinear pattern matching

We can introduce the notion of a safe shift distance, as was done for suffix-based algorithms with Algorithm 3.11 (P_+, S_+, GS=S, EGC=RSA, SSD) in Subsection 3.4.1 (and in more detail in [WZ96]). This leads to:

Algorithm 3.35(P_+, S_+, GS=F, EGC=RFA, SSD)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \to O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \to O := \varnothing$ **fi**;
$l, v := \varepsilon, \varepsilon;$
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$
$\wedge\ u = lv \wedge v \in \mathbf{fact}(P)$
$\wedge\ (l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{fact}(P))$ }
do $r \neq \varepsilon \to$
$u, r := u(r\restriction k(l, v, r)), r\downarrow k(l, v, r);\ l, v := u, \varepsilon;\ q := \delta_{R,\mathbf{fact}}(q_0, l\restriction 1);$
as $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
{ invariant: $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\restriction 1)v)^R)$ }
do $l \neq \varepsilon$ **cand** $q \neq \perp \to$
$l, v := l\downarrow 1, (l\restriction 1)v;$
$q := \delta_{R,\mathbf{fact}}(q, l\restriction 1);$
as $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
od
od{ R }

From this algorithm skeleton, we can once again derive various algorithms by using the safe shift functions derived before (since $(l\restriction 1)v \notin \mathbf{fact}(P) \Rightarrow (l\restriction 1)v \notin \mathbf{suff}(P)$). We can however do better, since $(l\restriction 1)v \notin \mathbf{fact}(P)$ is stronger than $(l\restriction 1)v \notin \mathbf{suff}(P)$. We derive:

$\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$
$\equiv$ { $(\star)$ }

$$
\begin{array}{ll}
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \\
\Rightarrow & \quad \{\ \mathbf{suff}(x) \cap P \neq \varnothing \Rightarrow |x| \geq lmin_P,\ |v(r\restriction n)| = |v| + n\ \} \\
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P \ (\star\star)
\end{array}
$$

We now show that the step marked $(\star)$ is valid:

- case $l = \varepsilon$:

$$
\begin{array}{ll}
& \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ u = lv,\ l = \varepsilon\ \} \\
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing
\end{array}
$$

- case $l \neq \varepsilon$ (hence $(l\restriction 1)v \notin \mathbf{fact}(P)$):

$$
\begin{array}{ll}
& \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ u = lv\ \} \\
& \mathbf{suff}(lv(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ l \neq \varepsilon\ \} \\
& \mathbf{suff}((l\downarrow 1)(l\restriction 1)v(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ \text{property of } \mathbf{suff}\text{: } \mathbf{suff}(xay) = \mathbf{suff}(x)ay \cup \mathbf{suff}(y)\ \} \\
& \Big(\mathbf{suff}(l\downarrow 1)\ (l\restriction 1)v(r\restriction n) \cup \mathbf{suff}(v(r\restriction n))\Big) \cap P \neq \varnothing \\
\equiv & \quad \{\ \cap \text{ distributes over } \cup\ \} \\
& \Big(\mathbf{suff}(l\downarrow 1)\ (l\restriction 1)v(r\restriction n) \cap P\Big) \cup \Big(\mathbf{suff}(v(r\restriction n)) \cap P\Big) \neq \varnothing \\
\equiv & \quad \{\ (l\restriction 1)v \notin \mathbf{fact}(P) \equiv V^*(l\restriction 1)vV^* \cap P = \varnothing, \text{ hence } \mathbf{suff}(l\downarrow 1)\ (l\restriction 1)v(r\restriction n) \cap P = \varnothing\ \} \\
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing
\end{array}
$$

Using $(\star)$, we observe that the left conjunct of $(\star\star)$ is equivalent to the predicate $\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing$ used to derive safe shift distances in [WZ96, Wat95]. Let $Weakening(\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing)$ be any weakening of that predicate. Then:

$$
\begin{array}{ll}
& (\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P : n) \\
\geq & \quad \{\ \} \\
& (\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing) \wedge |v| + n \geq lmin_P : n) \\
\geq & \quad \{\ \text{Property 2.6}\ \} \\
& (\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing) : n) \\
& \quad \mathbf{max}\, (\mathbf{MIN}\, n : 1 \leq n \wedge |v| + n \geq lmin_P : n) \\
= & \quad \{\ \} \\
& (\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing) : n)\, \mathbf{max}(1\, \mathbf{max}(lmin_P - |v|))
\end{array}
$$

We may thus use any shift function

$$(\mathbf{MIN}\, n : 1 \leq n \wedge \mathit{Weakening}(\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing) : n)\, \mathbf{max}(1\, \mathbf{max}(\mathit{lmin}_P - |v|)) \ .$$

It is clear that the left operand of the outer **max** corresponds to any safe shift function from [WZ96], represented by the various detail sequences given there. The right operand corresponds to the shift in case $(l\restriction 1)v$ is not a factor of a keyword.

Algorithm detail 3.36. (NFS). (No Factor Shift). The use of

$$k_{ssd,nfs}(l, v, r) = k_{ssd}(l, v, r)\, \mathbf{max}(1\, \mathbf{max}(\mathit{lmin}_P - |v|))$$

in a shift, for any safe shift function k_{ssd} given in [WZ96]. (It is called no-factor shift since it uses $(l\restriction 1)v \notin \mathbf{fact}(P)$.) □

The use of such shift distances results in algorithm (P_+, S_+, GS=F, EGC=RFA, SSD, NFS) and variants where detail (SSD) is further weakened using other details derived from it. One such weakening is that to *true*, leading to:

$$\begin{array}{ll} & (\mathbf{MIN}\, n : 1 \leq n \wedge \mathit{true} : n) \\ = & \quad \{\ \} \\ & 1 \end{array}$$

Algorithm detail 3.37. (ONE). The use of shift distance $k_{one} = 1$. □

Remark 3.38. In [WZ96], this shift distance corresponds to the one in algorithm (P_+, S_+, RT) (our Algorithm 3.8 (P_+, S_+, GS=S, EGC=RSA)), i.e. to not using larger shift distances by introducing detail (SSD). □

From this we derive a new shift distance:

$$\begin{array}{ll} & 1\, \mathbf{max}(1\, \mathbf{max}(\mathit{lmin}_P - |v|)) \\ = & \quad \{\ \} \\ & 1\, \mathbf{max}(\mathit{lmin}_P - |v|) \end{array}$$

This equals the shift distance used in the basic ideas for backward DAWG matching ([NR02, page 27]) and—combined with algorithm detail (LMIN) discussed in Subsection 3.6.1—set backward DAWG matching ([NR02, page 68]). The actual Backward DAWG Matching ([CCG+94], [NR02, page 28-29]) and Set Backward DAWG Matching ([CCG+94], [NR02, page 68]) algorithms use an improvement based on a property of DAWGs. We discuss this in Subsection 3.7.2.

3.7.2 Cheap computation of a particular shift function

Earlier in this section, we discussed the weakening of $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$ to *true*, leading to algorithm detail (ONE) and a constant shift function equaling 1 (or $1\,\mathbf{max}\,k_{nfs}$ in case detail (NFS) is introduced as well). Here, we consider a different weakening of $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$:

$$
\begin{array}{ll}
& \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ u = lv, \text{ if } l \neq \varepsilon \text{ then } (l\restriction 1)v \notin \mathbf{fact}(P), \text{ if } l = \varepsilon \text{ then } u = v\ \} \\
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{\ \text{introduce } last = (\mathbf{MAX}\, m : 0 \leq m \leq |v| \wedge v\restriction m \in \mathbf{pref}(P) : m)\ \} \\
& \mathbf{suff}((v\restriction last)(r\restriction n)) \cap P \neq \varnothing \\
\Rightarrow & \quad \{\ n \leq |r|,\ r\restriction n \in V^n, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap\ \} \\
& \mathbf{suff}((v\restriction last)V^n) \cap P \neq \varnothing
\end{array}
$$

We now derive

$$
\begin{array}{ll}
& (\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}((v\restriction last)V^n) \cap P \neq \varnothing : n) \\
= & \quad \{\ \text{property of } \mathbf{suff}\text{: } \mathbf{suff}(A) \cap B \neq \varnothing \equiv A \cap V^*B \neq \varnothing\ \} \\
& (\mathbf{MIN}\, n : 1 \leq n \wedge (v\restriction last)V^n \cap V^*P \neq \varnothing : n) \\
\geq & \quad \{\ last \leq |v|,\ v\restriction last \in V^{last}, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap\ \} \\
& \left(\mathbf{MIN}\, n : 1 \leq n \wedge V^{last+n} \cap V^*P \neq \varnothing : n\right) \\
\geq & \quad \{\ \text{Property 2.6}\ \} \\
& (\mathbf{MIN}\, n : 1 \leq n : n)\ \mathbf{max}\ \left(\mathbf{MIN}\, n : V^{last+n} \cap V^*P \neq \varnothing : n\right) \\
= & \quad \{\ lmin_P = (\mathbf{MIN}\, p : p \in P : |p|)\ \} \\
& 1\,\mathbf{max}(lmin_P - last)
\end{array}
$$

The last quantification depends on $last = (\mathbf{MAX}\, m : 0 \leq m \leq |v| \wedge v\restriction m \in \mathbf{pref}(P) : m)$, which seems to be rather difficult to compute. When using a DAWG to implement the transition function $\delta_{R,\mathbf{fact}}$ of algorithm detail (EGC=RFA) however, we may use a property of this automaton to compute *last* 'on the fly': the final states of the DAWG correspond to suffixes of some $p^R \in P^R$, i.e. to prefixes of some $p \in P$. Thus, *last* equals the length of v at the moment the most recent final state was visited.

We can thus use the following shift function without any need for precomputation:

Definition 3.39 (Shift function k_{lskp}). Shift function k_{lskp} is defined as:

$$k_{lskp} = 1\,\mathbf{max}(lmin_P - last)$$

□

Remark 3.40. Note that this shift function does not depend on l. It can therefore be seen as a variant of algorithm detail (NLA). The shift function does not directly depend on v either, but it indirectly depends on v due to its dependence on *last*.

Algorithm detail 3.41. (LSKP). (Longest Suffix that is Keyword Prefix). Calculating the shift distance using k_{lskp} is algorithm detail (LSKP). □

Using variable *last* and shift function k_{lskp}, the algorithm becomes:

Algorithm 3.42(P$_+$, S$_+$, GS=F, EGC=RFA, SSD, NLAU, OPT, NLA, LSKP)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
$l, v := \varepsilon, \varepsilon;$
$last := 0;$
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$
$\wedge\ u = lv \wedge v \in \mathbf{fact}(P)$
$\wedge\ (l = \varepsilon\ \mathbf{cor}\ (l \lceil 1)v \notin \mathbf{fact}(P))$ }
do $r \neq \varepsilon \rightarrow$
$k := 1\ \mathbf{max}(lmin_P - last);$
$u, r := u(r \lceil k), r \lfloor k;\ l, v := u, \varepsilon;$
$q, last := \delta_{R,\mathbf{fact}}(q_0, l \lceil 1), 0;$
as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
{ invariant: $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l \lceil 1)v)^R)$
$\wedge\ last = (\mathbf{MAX}\, m : m \leq |v| \wedge v \lceil m \in \mathbf{pref}(P) : m)$ }
do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
$l, v := l \lfloor 1, (l \lceil 1)v;$
$q := \delta_{R,\mathbf{fact}}(q, l \lceil 1)$
as $q \in F \rightarrow last := |v|$ **sa**;
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
od{ R }

This algorithm is a variant of the actual Set Backward DAWG Matching ([CCG+94], [NR02, page 68]) algorithm, which is the same except for the addition of algorithm detail LMIN: it can be described as (P$_+$, S$_+$, GS=F, EGC=RFA, LMIN, SSD, NLAU, OPT, NLA, LSKP), while (P$_+$, S$_+$, GS=F, EGC=RFA, SSD, NLAU, OPT, NLA, LSKP, OKW) describes single-keyword Backward DAWG Matching.

The reader may have noticed that algorithm detail (NFS) (introduced in the previous subsection) is not included in either of the two detail sequences. This is done because the no-factor shift can never be larger than the shift according to k_{lskp}:

$lmin_P - |v|$
$\leq$ { $last \leq |v|$ }

$$
\begin{aligned}
& lmin_P - last \\
= & \quad \{ \text{ definition } k_{lskp} \ \} \\
& k_{lskp}
\end{aligned}
$$

In addition, we note that the quantification $(\mathbf{MIN}\, n : 1 \leq n \wedge (v \lceil last) V^n \cap V^* P \neq \varnothing : n)$ in the second line of the last derivation above equals $d_{sp}(v \lceil last)$ ([WZ96, page 98]) resp. $d_2(v \lceil last)$ ([Wat95, page 89]). It follows that shift function k_{lskp} gives an approximation from below of that function.

Note that we do not include algorithm detail LMIN in the detail sequence of the single-keyword Backward DAWG algorithm either: Although this would make sense since it is the single-keyword version of Algorithm (P_+, S_+, GS=F, EGC=RFA, LMIN, SSD, NLAU, OPT, NLA, LSKP), the addition of algorihm detail LMIN does not influence the algorithm when combined with problem detail OKW. We therefore opt to use the shortest possible detail sequence that describes the algorithm.

3.8 Factor oracle-based pattern matching

We now derive a family of algorithms by using the language of a factor oracle on P^R, **factoracle**(P^R). Although the exact definition of this language is not yet known, it has been proven to be a superset of **fact**(P^R) and to be suffix-closed[11]. We introduce

Algorithm detail 3.43. (GS=FO). (Guard Strengthening = Factor Oracle). Strengthen the guard of the inner repetition by adding conjunct $(l \lceil 1)v \in$ **factoracle**$(P^R)^R$. □

Remark 3.44. Note that **factoracle**(P^R) and **factoracle**$(P)^R$ are not in general the same; see Remark 4.14 for an example. □

Since $P \subseteq$ **factoracle**$(P^R)^R$ and **suff**(**factoracle**$(P^R)^R$) $\subseteq$ **factoracle**$(P^R)^R$ both hold, this guard strengthening may be used. The inner repetition guard then becomes

$$l \neq \varepsilon \ \mathbf{cand}\ (l \lceil 1)v \in \mathbf{factoracle}(P^R)^R$$

Since the exact definition of **factoracle** independent of the factor oracle automaton is currently unknown, direct evaluation of $(l \lceil 1)v \in$ **factoracle**$(P^R)^R$ is not possible. Instead, we will use the transition function of the factor oracle (see Chapter 4, as well as [ACR01, AR99]) recognizing the set **factoracle**(P^R). Using function $\delta_{\mathbf{factoracle}(P^R)}$[12] and making $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l \lceil 1)v)^R)$ an invariant of the inner repetition, the guard becomes

$$l \neq \varepsilon \ \mathbf{cand}\ q \neq \perp$$

[11] We prove this for the single keyword factor oracle in Chapter 4, and it is proven for the multiple keyword version in [AR99]. Proofs for the single keyword version can be found in [ACR01] as well.

[12] Since in general **factoracle**$(P)^R \neq$ **factoracle**(P^R), we cannot use $\delta_{R,\mathbf{factoracle}}$ to describe the transition function of the automaton used. We therefore introduce the notation $\delta_{\mathbf{factoracle}(P^R)}$, the transition function of the automaton recognizing **factoracle**(P^R).

Algorithm detail 3.45. (EGC=RFO). (Efficient Guard Computation = Reverse Factor Oracle). Given a finite automaton recognizing $\mathbf{factoracle}(P^R)$ and satisfying Property 3.5, update a state variable q to uphold invariant $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{factoracle}(P^R)^R$ then becomes $q \neq \perp$. □

The use of algorithm detail (EGC=RFO) leads to

Algorithm 3.46(P_+, S_+, GS=F, EGC=RFO)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\restriction 1), r\downarrow 1;\ l, v := u, \varepsilon;\ q := \delta_{\mathbf{factoracle}(P^R)}(q_0, l\restriction 1);$
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $u = lv \wedge v \in \mathbf{factoracle}(P^R)^R \wedge q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l\restriction 1)v)^R)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v;$
    $q := \delta_{\mathbf{factoracle}(P^R)}(q, l\restriction 1);$
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
  { $l = \varepsilon$ **cor** $(l\restriction 1)v \notin \mathbf{factoracle}(P^R)^R$ }
od{ R }

This algorithm has $\Theta(|S|)$ running time, just like Algorithm 3.8 (P_+, S_+, GS=S, EGC=RSA) and Algorithm 3.34 (P_+, S_+, GS=F, EGC=RFA).

The use of detail sequence (GS=FO, EGC=RFO) instead of (GS=F, EGC=RFA) has the following effects:

- Easier construction of and more compact automata: The factor oracle recognizing $\mathbf{factoracle}(P^R)$ is easier to construct and may have less states and transitions than an automaton recognizing $\mathbf{fact}(P^R)$ (see Chapter 4).
- More character comparisons: When $(l\restriction 1)v \notin \mathbf{fact}(P)^R$ yet $(l\restriction 1)v \in \mathbf{factoracle}(P^R)^R$, the guard of the inner loop will still be true, and hence the algorithm will go on extending v to the left more than strictly necessary.

Note that the effects of using (GS=FO, EGC=RFO) instead of (GS=S, EGC=RSA) are a combination of the effects mentioned here and those described in Section 3.7 when comparing (GS=F, EGC=RFA) and (GS=S, EGC=RSA).

3.8.1 Factor oracle-based sublinear pattern matching

We can introduce the notion of a safe shift distance, as was done for suffix-based algorithms in Subsection 3.4.1, and for factor-based algorithms in Subsection 3.7.1. This leads to:

Algorithm 3.47(P_+, S_+, GS=FO, EGC=RFO, SSD)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**;
$l, v := \varepsilon, \varepsilon;$
{ invariant: $ur = S \wedge O = (\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\})$
$\wedge\ u = lv \wedge v \in \mathbf{factoracle}(P^R)^R$
$\wedge\ (l = \varepsilon\ \mathbf{cor}\ (l \uparrow 1)v \notin \mathbf{factoracle}(P^R)^R)$ }
do $r \neq \varepsilon \rightarrow$
$u, r := u(r \uparrow k(l, v, r)), r \downarrow k(l, v, r);\ l, v := u, \varepsilon;\ q := \delta_{\mathbf{factoracle}(P^R)}(q_0, l \uparrow 1);$
as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
{ invariant: $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l \uparrow 1)v)^R)$ }
do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
$l, v := l \downarrow 1, (l \uparrow 1)v;$
$q := \delta_{\mathbf{factoracle}(P^R)}(q, l \uparrow 1);$
as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
od
od{ R }

We derive

$$
\begin{array}{ll}
& (l \uparrow 1)v \notin \mathbf{factoracle}(P^R)^R \\
\Rightarrow & \quad \{\ \mathbf{factoracle}(P^R)^R \supseteq \mathbf{fact}(P^R)^R = \mathbf{fact}(P)\ \} \\
& (l \uparrow 1)v \notin \mathbf{fact}(P)
\end{array}
$$

Therefore any shift function may be used satisfying

$$(\mathbf{MIN}\, n : 1 \leq n \wedge \mathit{Weakening}(\mathbf{suff}(v(r \uparrow n)) \cap P \neq \varnothing) : n)\, \mathbf{max}(1\, \mathbf{max}(\mathit{lmin}_P - |v|))$$

as derived for factor-based algorithms in Subsection 3.7.1.

The Set Backward Oracle Matching algorithm ([AR99], [NR02, pages 69-72]) equals our algorithm (P_+, S_+, GS=FO, EGC=RFO, LMIN, SSD, NFS, ONE), while the single keyword Backward Oracle Matching algorithm ([ACR01], [NR02, pages 34-36] and Chapter 4 of this text) corresponds to (P_+, S_+, GS=FO, EGC=RFO, SSD, NFS, ONE, OKW).

3.9 Prefix-based pattern matching

In this section we show how to derive the Aho-Corasick and Knuth-Morris-Pratt algorithms and variants thereof, as well as the Shift algorithms. The first part of this section is based on [Wat95, section 4.3].

In Section 3.3, a triple format was used for set O. We remove the redundancy of that format by registering matches by their end-points only; i.e. by dropping the first component of the triples in O.

Problem detail 3.48. (E). (Endpoints). Matches are registered by their end-points. □

Dropping the first component of the triples allows some efficiency improvements to the algorithms. The postcondition R can be rewritten as in [Wat95, page 59], leading to the new postcondition

$$R_e : \; O_e = \Big(\bigcup \; u, r : ur = S \wedge v \in \mathbf{suff}(u) \cap P : \{(v, r)\} \Big) .$$

Adding details (+) and (E) to Algorithm 3.2 (P) of Section 3.3 and using the modified postcondition gives

Algorithm 3.49(P+, E)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O_e := \{(\varepsilon, S)\} \; [\!] \; \varepsilon \notin P \rightarrow O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$ }
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r\!\uparrow\!1), r\!\downarrow\!1;$
  $O_e := O_e \cup (\bigcup \; v : v \in \mathbf{suff}(u) \cap P : \{(v, r)\})$
od
{ R_e }

3.9.1 Towards the Aho-Corasick and Knuth-Morris-Pratt algorithms

In Algorithm 3.49, new matches are registered whenever $\mathbf{suff}(u) \cap P \neq \varnothing$. The essential idea of both the Aho-Corasick and Knuth-Morris-Pratt algorithms is to introduce an easily updateable state variable that gives information about (partial) matches in $\mathbf{suff}(u)$ and allows easy computation of the set $\mathbf{suff}(u) \cap P$.

To facilitate the update to O_e, a variable U is introduced, related to u by the invariant $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$. Since $P \subseteq \mathbf{pref}(P)$, $U \cap P = \mathbf{suff}(u) \cap P$ and we may use $U \cap P$ in the update of variable O_e.

Algorithm detail 3.50. (SP). (Set of Prefixes of P). Introduction of $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ to facilitate updating O_e. □

The introduction of this detail leads to the following algorithm:[13]

[13]The introduction of variable U is discussed in more detail in [Wat95, subsection 4.3.1], although it is not treated as an explicit algorithm detail. Algorithm 3.51 (P+, E, SP) corresponds to the nameless Algorithm [Wat95, 4.38].

Algorithm 3.51(P$_+$, E, SP)

$u, r := \varepsilon, S$; $U := \{\varepsilon\}$;
if $\varepsilon \in P \rightarrow O_e := \{(\varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$
$\wedge\ U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ }
do $r \neq \varepsilon \rightarrow$
$U := (U(r\restriction 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$;
$u, r := u(r\restriction 1), r\downarrow 1$;
$O_e := O_e \cup (U \cap P) \times \{r\}$
od
{ R_e }

Variable u is now superfluous, but will be kept to help formulate invariants.

In [Wat95, page 61] the remark is made that "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 $U := (U(r\restriction 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$." Although U indeed is a language, its size will often be fairly small, since it is limited by $|\mathbf{pref}(P)|$. We will use this observation in Subsection 3.9.2 to derive the well-known Shift-And, Shift-Or and Multiple Shift-And algorithms.

Algorithm 3.51 (P$_+$, E, SP) is further developed into the optimal and failure function Aho-Corasick algorithms and the Knuth-Morris-Pratt algorithm in [Wat95, subsections 4.3.1-4.3.2, 4.3.4-4.3.6]. A bit-parallel version of the optimal Aho-Corasick algorithm (which also uses the above observation) will be discussed in Subsection 3.9.3.

3.9.2 Bit-parallel ancestors of AC and KMP: Shift-And and Shift-Or algorithms

In [BYG89], Baeza-Yates and Gonnet use the idea of bit-parallel updating of variable $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$. In this section, we show that (Multiple) Shift-And and Shift-Or ([BYG89, WM92]) can be derived as part of our taxonomy as well.

The set $U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$ is of course bounded from above in size by $|\mathbf{pref}(P)|$, but it is also bounded from above in size by the weaker $(\sum_{n=0}^{|P|-1} |p_n|) + 1$ (note that P was defined in Section 3.3 as the finite non-empty set $\{p_0, p_1, \ldots p_{|P|-1}\} \subseteq V^*$), where the term 1 is necessary to include the empty prefix ε. It is not hard to see that U always contains ε however, and we therefore choose not to represent it explicitly. We therefore introduce a *bitvector* to store set $U - \{\varepsilon\}$:

Definition 3.52 (Bitvector D_P). Bitvector D_P is defined for $0 \leq i(k,l) < (\sum_{n=0}^{|P|-1} |p_n|)$ by

$$D_P[i(k,l)] \equiv p_k\restriction l \in U$$

where $0 \leq k \leq |P| - 1$ and $1 \leq l \leq |p_k|$ and $i = (\sum_{m=0}^{k-1} |p_m|) + l - 1$. We use $i(k,l)$ to indicate that i is dependent on both k and l. □

Remark 3.53. Somewhat counterintuitive, we will start the numbering of bit-positions in a bitvector from the right in our examples. This is exactly opposite to the normal positioning of elements in vectors from left to right, but it is the way bitvectors are normally used in computing science literature. □

Example 3.54 (Bitvector $D_{\{he,she,hers\}}$). For $P = \{he, she, hers\}$, $D_P[i(k,l)]$ will be defined as follows

Position	8	7	6	5	4	3	2	1	0
D_P	*hers*	*her*	*he*	*h*	*she*	*sh*	*s*	*he*	*h*
	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$	$\in$
	U	U	U	U	U	U	U	U	U

Note that for this particular set of keywords (and in general for any set of keywords including keywords with one ore more equal prefixes not equaling ε), there is duplication of information going on. In section 3.9.3, we will use a bit-parallel encoding of the optimal Aho-Corasick automaton instead, in order to remove this redundancy. □

We now derive for the update to $D_P[i(k,l)]$:

$$
\begin{aligned}
& D_P[i(k,l)]\big(U := (U(r\restriction 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}\big) \\
\equiv & \quad \{ \text{ definition of } D_P \ \} \\
& p_k\restriction l \in (U(r\restriction 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\} \\
\equiv & \quad \{ \ p_k\restriction l \neq \varepsilon \ \} \\
& p_k\restriction l \in U(r\restriction 1) \cap \mathbf{pref}(P) \\
\equiv & \quad \{ \text{ property of } \cap \ \} \\
& p_k\restriction l \in U(r\restriction 1) \wedge p_k\restriction l \in \mathbf{pref}(P) \\
\equiv & \quad \{ \ p_k\restriction l \in \mathbf{pref}(P) \text{ by definition, due to values } k \text{ and } l \ \} \\
& p_k\restriction l \in U(r\restriction 1)
\end{aligned}
$$

We distinguish two cases:

- case $l = 1$:

$$
\begin{aligned}
& p_k\restriction l \in U(r\restriction 1) \\
\equiv & \quad \{ \ \varepsilon \in U, r \neq \varepsilon \ \} \\
& p_k[l] = r\restriction 1 \\
\equiv & \quad \{ \text{ introduction of } B_P \text{ as below } \} \\
& B_P[r\restriction 1][i(k,l)]
\end{aligned}
$$

- case $1 < l \leq |p_k|$:

$$
\begin{aligned}
& p_k\rceil l \in U(r\rceil 1) \\
\equiv \quad & \{ \text{ definition } \rceil/\lfloor,\ r \neq \varepsilon \ \} \\
& p_k\rceil(l-1) \in U \wedge p_k[l] = r\rceil 1 \\
\equiv \quad & \{ \text{ definition of } D_P, \text{ introduction of } B_P \text{ as below } \} \\
& D_P[i(k,l)-1] \wedge B_P[r\rceil 1][i(k,l)]
\end{aligned}
$$

As noted in these derivations, we introduce a *bitmatrix* B_P:

Definition 3.55 (Bitmatrix B_P). The bitmatrix B_P is defined for $v \in V$ and for $0 \leq i(k,l) < (\sum_{n=0}^{|P|-1} |p_n|)$ by

$$B_P[v][i(k,l)] \equiv p_k[l] = v$$

where $0 \leq k \leq |P| - 1$ and $1 \leq l \leq |p_k|$ and $i(k,l) = (\sum_{m=0}^{k-1} |p_m|) + l - 1$. □

Example 3.56 (Bitmatrix $B_{\{he,she,hers\}}$). For $P = \{he, she, hers\}$ and $V = \{e, h, i, r, s\}$, B_P will be defined as follows

	s	r	e	h	e	h	s	e	h
Position	8	7	6	5	4	3	2	1	0
$B_P[e]$	0	0	1	0	1	0	0	1	0
$B_P[h]$	0	0	0	1	0	1	0	0	1
$B_P[i]$	0	0	0	0	0	0	0	0	0
$B_P[r]$	0	1	0	0	0	0	0	0	0
$B_P[s]$	1	0	0	0	0	0	1	0	0

□

Based on the above derivations, the update of $D_P[i]$ should be $D_P[i] := B_P[r\rceil 1][i]$ in case $l = 1$, and $D_P[i] := D_P[i-1] \wedge B_P[r\rceil 1][i]$ in case $1 < l \leq |p_k|$.

We would like to update D_P as a whole using the *left shift operator* $<<$. This operator shifts each bit to the left by one position, dropping the highest bit and inserting a 0 bit at position 0 (note that we assume bits to be numbered in increasing order from right to left). The update would then become $D_P := (D_P << 1)\ \&\ B_P[r\rceil 1]$.

That update is incorrect however for positions $i(k,l)$ of D_P for which $l = 1$. The update would be correct if, after shifting D_P to the left, we mark such positions i by 1, before applying &. Thus, we introduce a bitvector I_P:

Definition 3.57 (Bitvector I_P). Bitvector I_P is defined for $0 \leq i(k,l) < (\sum_{n=0}^{|P|-1} |p_n|)$ by

$$I_P[i(k,l)] \equiv l = 1$$

where $0 \leq k \leq |P| - 1$ and $1 \leq l \leq |p_k|$ and $i(k,l) = (\sum_{m=0}^{k-1} |p_m|) + l - 1$. □

Using I_P, the single update statement becomes

$$D_P := ((D_P << 1)\ |\ I_P)\ \&\ B_P[r\rceil 1].$$

Example 3.58 (Bitvector $I_{\{he,she,hers\}}$). For $P = \{he, she, hers\}$, I_P will be

	s	r	e	h	e	h	s	e	h
Position	8	7	6	5	4	3	2	1	0
I_P	0	0	0	1	0	0	1	0	1

□

Using the bitvectors D_P and I_P and bitmatrix B_P, we get the following algorithm:

Algorithm 3.59()

$u, r := \varepsilon, S$; $U := \{\varepsilon\}$;
InitializeBitVectors;
if $\varepsilon \in P \rightarrow O_e := \{(\varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$
$\wedge\ U = \mathbf{suff}(u) \cap \mathbf{pref}(P)$
$\wedge\ D_P$ represents $\mathbf{suff}(u) \cap \mathbf{pref}(P) - \{\varepsilon\}$ }
do $r \neq \varepsilon \rightarrow$
$U := (U(r\uparrow 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}$;
$D_P := ((D_P << 1) | I_P) \& B_P[r\uparrow 1]$;
$u, r := u(r\uparrow 1), r\downarrow 1$;
$O_e := O_e \cup (U \cap P) \times \{r\}$
od
{ R_e }

We assume that *InitializeBitVectors* initializes D_P, I_P and B_P correctly. A possible implementation of *InitializeBitVectors* is briefly discussed at the end of this subsection.

To completely replace U, the update to O_e needs to be replaced by

as $\varepsilon \in P \rightarrow O_e := O_e \cup \{(\varepsilon, r)\}$ **sa**
$O_e := O_e \cup (\bigcup\ k : D_P[i(k, |p_k|)] : \{(p_k, r)\})$

Algorithm detail 3.60. (BPSP). (Bit parallel set of prefixes of P). Use bitvectors D_P and I_P and bitmatrix B_P to maintain the set U introduced by algorithm detail (SP). □

The set U has now become redundant and may be removed. This brings us to the Multiple Shift-And algorithm ([BYG89], [NR02, p. 45-47]):

Algorithm 3.61(P_+, E, SP, BPSP)

$u, r := \varepsilon, S$;
InitializeBitVectors;
if $\varepsilon \in P \rightarrow O_e := \{(\varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$

$\wedge\ D_P$ represents $\mathbf{suff}(u) \cap \mathbf{pref}(P) - \{\varepsilon\}$ }
do $r \neq \varepsilon \rightarrow$
$D_P := ((D_P << 1) | I_P) \& B_P[r\restriction 1];$
$u, r := u(r\restriction 1), r\downarrow 1;$
as $\varepsilon \in P \rightarrow O_e := O_e \cup \{(\varepsilon, r)\}$ **sa**;
$O_e := O_e \cup (\bigcup\ k : D_P[i(k, |p_k|)] : \{(p_k, r)\})$
od
{ R_e }

Thus, the Multiple Shift-And algorithm can be described by detail sequence (P+, E, SP, BPSP), while (P+, E, SP, BPSP, OKW) describes the single-keyword Shift-And algorithm ([BYG89], [NR02, p. 19-21]).

In the latter case, i.e. for single keyword pattern matching, we may use a trick to remove I_P from the algorithm altogether: we invert the meaning of the bits of D_P, B_P and I_P and derive:

$$\begin{aligned}
& \overline{((D_P << 1) | I_P) \& B_P[r\restriction 1]} \\
\equiv\ & \quad \{\ \overline{\ } = \&, \overline{\&} = |\ \} \\
& (\overline{(D_P << 1)} \& \overline{I_P}) | \overline{B_P[r\restriction 1]} \\
\equiv\ & \quad \{\ \overline{I_P} = 1^{|P|-1}0\ \} \\
& (\overline{D_P} << 1) | \overline{B_P[r\restriction 1]}
\end{aligned}$$

We can thus completely remove I_P by inverting the meaning of the bits, and may then use the update statement

$$D_P := (D_P << 1)\ |\ B_P[r\restriction 1].$$

Algorithm detail 3.62. (INV). (INVert). Invert the bits in bitvector D_P and bitmatrix B_P, leading to the removal of bitvector I_P from the algorithm, if detail (OKW) has been applied. □

The addition of algorithm detail (INV) leads to the Shift-Or algorithm:

Algorithm 3.63(P+, E, SP, BPSP, OKW, INV)

$u, r := \varepsilon, S;$
InitializeBitVectors;
if $\varepsilon \in P \rightarrow O_e := \{(\varepsilon, S)\}\ [\!]\ \varepsilon \notin P \rightarrow O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup\ x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$
$\wedge\ (\neg D_P)$ represents $\mathbf{suff}(u) \cap \mathbf{pref}(P) - \{\varepsilon\}$ }
do $r \neq \varepsilon \rightarrow$
$D_P := (D_P << 1) | B_P[r\restriction 1];$
$u, r := u(r\restriction 1), r\downarrow 1;$

as $\varepsilon \in P \rightarrow O_e := O_e \cup \{(\varepsilon, r)\}$ **sa**;
$O_e := O_e \cup (\bigcup\ k, l : \neg D_P[i(k, |p_k|)] : \{(p_k, r)\})$
od
$\{\ R_e\ \}$

Initialization of I_P, D_P and B_P

The bitvectors I_P and D_P and bitmatrix B_P that are used in the preceding algorithms may be initialized as follows.[14] Note that these initializations describe the situation *without* use of detail (INV); for the initialization when using that detail, all 0 bits should be replaced by 1 bits and vice versa. In the following text, we use $s = \sum_{i=0}^{|P|-1} |p_i|$ and $s(n) = \sum_{i=0}^{n-1} |p_i|$.

Initially, $U = \{\varepsilon\}$ hence $U - \{\varepsilon\} = \varnothing$. D_P can be initialized by

$$D_P := 0^s$$

Based on the definition of I_P, I_P may be initialized by

$I_P := 0^s$;
for $n : 0 \leq n < |P| \rightarrow$
    $I_P := I_P \mid 0^{s-s(n)-1}10^{s(n)}$
rof

For B_P, we use the following initialization:

for $v : v \in V \rightarrow$
    $B_P[v] := 0^s$;
rof;
for $n : 0 \leq n < |P| \rightarrow$
    for $j : 0 \leq j < |p_n| \rightarrow$
        $B_P[p_n[j]][s(n) + j] := 1$
    rof
rof

Remark 3.64. Note that the initializations in this section are intended to give an idea of how such an initialization could be performed, but have not been refined to be translatable statement-by-statement into C++ code for example. Based on the above GCL code however, such a translation is relatively easy: variables s and $s(n)$ should be introduced to update the value of s and $s(n)$ on the fly, and the data type used to represent bitvectors should allow for individual bits to be set. The `bitset` in C++'s STL supports this for example. □

[14] These initializations are based on those described in [NR02, p.46-47].

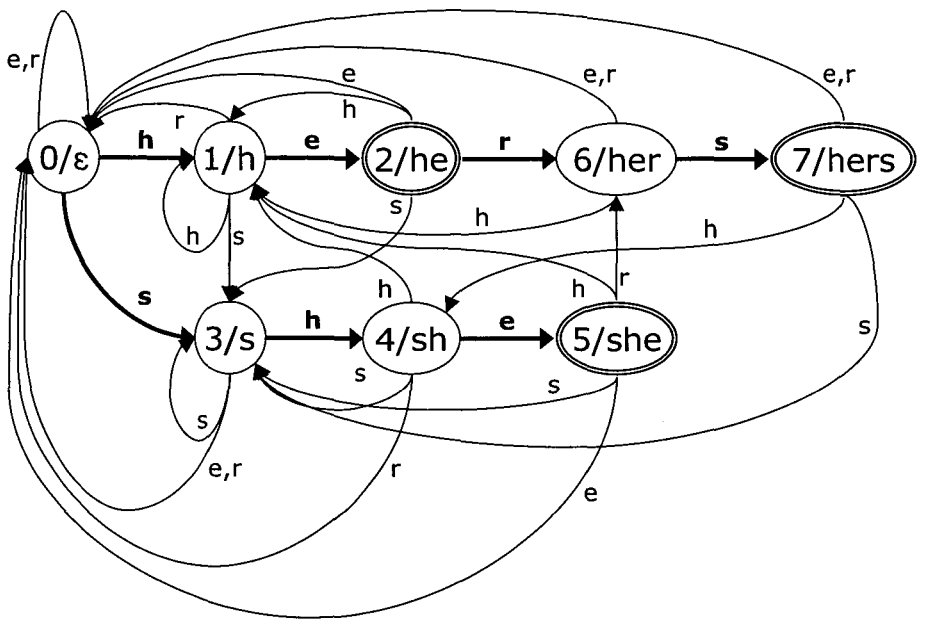

Figure 3.2: Optimal Aho-Corasick Automaton for $P = \{he, she, hers\}$. States are labeled both by their state identifier as well as by the shortest string to the state from state 0.

3.9.3 A bit-parallel Aho-Corasick algorithm

As we mentioned in Example 3.54 of Subsection 3.9.2, the bitvector D_P may contain redundant information, depending on the particular keyword set P used. In terms of finite automata, we can think of the algorithms in that subsection as bit-parallel encodings of $|P|$ separate automata, each recognizing $\mathbf{pref}(p)$ for different $p \in P$.

In this section, we show that it is possible to encode the optimal Aho-Corasick automaton (ACOpt) instead. This automaton is an extension of the trie recognizing $\mathbf{pref}(P)$.[15] Figure 3.2 shows ACOpt(P) for $P = \{he, she, hers\}$.

In that automaton, for each state, except for state 0, all incoming transitions of the state are labeled by the same symbol.

We introduce a bitvector to store the active state of the ACOpt machine:

Definition 3.65 (Bitvector D). Bitvector D is defined for $0 \leq i < |\text{ACOpt}(P)|$ by

$$D[i] \equiv \gamma_f^*(0, u) = i$$

□

[15]The automaton corresponds to function γ_f as used in [Wat95, subsections 4.3.1-4.3.2]. The algorithm we derive in this section thus can be seen as an encoding of (P+, E, AC, AC-OPT) in [Wat95]. That algorithm would be called (P+, E, SP, AC, AC-OPT) in our taxonomy.

We now derive for the update to D:

$$
\begin{aligned}
& D[i](u := u(r\restriction 1)) \\
\equiv & \quad \{ \text{ definition } D \ \} \\
& \gamma_f^*(0, u(r\restriction 1)) = i \\
\equiv & \quad \{ \text{ definition } \gamma_f^* \ \} \\
& \gamma_f(\gamma_f^*(0,u), r\restriction 1) = i \\
\equiv & \quad \{ \text{ "case distinction" (working towards application of definition of } D) \ \} \\
& (\exists j :: \gamma_f^*(0,u) = j \wedge \gamma_f(j, r\restriction 1) = i) \\
\equiv & \quad \{ \text{ definition } D \ \} \\
& (\exists j :: D[j] \wedge \gamma_f(j, r\restriction 1) = i)
\end{aligned}
$$

We now make a case distinction, recalling that incoming transitions to state 0 can have different labels, while all incoming transitions of any other state have the same label, and derive for case $i \neq 0$:

$$
\begin{aligned}
& (\exists j :: D[j] \wedge \gamma_f(j, r\restriction 1) = i) \\
\equiv & \quad \{ \text{ for each state, all incoming transitions are on the same symbol } \} \\
& (\exists j :: D[j] \wedge (\exists a :: \gamma_f(j,a) = i) \wedge (\exists k :: \gamma_f(k, r\restriction 1) = i)) \\
\equiv & \quad \{ \text{ introduction of } B_P \text{ as below } \} \\
& (\exists j :: D[j] \wedge (\exists a :: \gamma_f(j,a) = i) \wedge B_P[r\restriction 1][i]) \\
\equiv & \quad \{ \text{ introduction of } T,\ i \neq 0 \ \} \\
& (\exists j :: D[j] \wedge T[j][i] \wedge B_P[r\restriction 1][i]) \\
\equiv & \quad \{ \text{ calculus } \} \\
& (\exists j :: D[j] \wedge T[j][i]) \wedge B_P[r\restriction 1][i] \\
\equiv & \quad \{ \text{ working towards bitoperations on rows of bitmatrix } T \ \} \\
& (|j : D[j] : T[j])[i] \wedge B_P[r\restriction 1][i] \\
\equiv & \quad \{ \ (|j : D[j] : T[j]) \text{ selects unique row in } T \text{ with row nr. equal to active bit in } D \ \} \\
& (T[D] \& B_P[r\restriction 1])[i]
\end{aligned}
$$

For case $i = 0$ we derive:

$$
\begin{aligned}
& D[0](u := u(r\restriction 1)) \\
\equiv & \quad \{ \text{ definition } D \ \} \\
& \gamma_f^*(0, u(r\restriction 1)) = 0 \\
\equiv & \quad \{ \ \gamma_f \text{ transition function of deterministic automaton } \}
\end{aligned}
$$

$$
\begin{aligned}
& (\forall j : 0 < j : \gamma_f^*(0, u(r\restriction 1)) \neq j) \\
\equiv & \quad \{ \text{ definition } D \ \} \\
& (\forall j : 0 < j : \neg D[j])
\end{aligned}
$$

As noted in these derivations, we introduce bitmatrices B_P—whose rows indicate states reachable by a given symbol—and T—whose rows indicate states reachable from a given state.[16]:

Definition 3.66 (Bitmatrix B_P). The bitmatrix B_P is defined for $v \in V$ and for $0 \leq i < |\text{ACOpt}(P)|$ by

$$B_P[v][i] \equiv (\exists j :: \gamma_f(j, v) = i)$$

□

Definition 3.67 (Bitmatrix T). The bitmatrix T is defined for $0 \leq i < |\text{ACOpt}(P)|$ and $0 \leq j < |\text{ACOpt}(P)|$ by

$$T[i][j] \equiv (\exists v \in V :: \gamma_f(i, v) = j)$$

□

Example 3.68 (Bitmatrix $B_{\{he,she,hers\}}$). For $P = \{he, she, hers\}$, $V = \{e, h, i, r, s\}$, and ACOpt/γ_f as in Figure 3.2, B_P will be defined as follows

State	7	6	5	4	3	2	1	0
$B_P[e]$	0	0	1	0	0	1	0	1
$B_P[h]$	0	0	0	1	0	0	1	0
$B_P[i]$	0	0	0	0	0	0	0	1
$B_P[r]$	0	1	0	0	0	0	0	1
$B_P[s]$	1	0	0	0	1	0	0	0

□

Example 3.69 (Bitmatrix T). For $V = \{e, h, i, r, s\}$, and ACOpt/γ_f as in Figure 3.2, T will be defined as follows

State	7	6	5	4	3	2	1	0
$T[7]$	0	0	0	1	1	0	0	1
$T[6]$	1	0	0	0	0	0	1	1
$T[5]$	0	1	0	0	1	0	1	1
$T[4]$	0	0	1	0	1	0	1	1
$T[3]$	0	0	0	1	1	0	0	1
$T[2]$	0	1	0	0	1	0	1	1
$T[1]$	0	0	0	0	1	1	1	1
$T[0]$	0	0	0	0	1	0	1	1

□

[16] We name these in accordance with the naming of these structures in the bit-parallel Glushkov regular expression pattern matching algorithm as described in [NR02, pages 122-123]. Note that in that algorithm, an NFA is used and more than one state can be active.

Using B_P and T, D may be updated by

$$D := T[q] \,\&\, B_P[r\uparrow 1];$$
$$D[0] := (D = 0 \vee D = 1);$$

where q is the unique state such that $D[q] = 1$.

Algorithm detail 3.70. (BPAC). (Bit parallel Aho-Corasick). Use bitvector D and bitmatrices B_P and T to encode the optimal Aho-Corasick automaton as used in algorithm detail (AC-OPT). □

As in the previous section, we need to replace the update of O_e as well. We first introduce function *Output*, the Aho-Corasick output function, defined as:

Definition 3.71 (Function *Output*). Function $Output \in Q \to \mathcal{P}(P)$ is defined by

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

where w is the shortest string such that $\gamma_f^*(0, w) = q$ (i.e. w is the label of the path to q in the trie part of ACOpt). □

Using this function and given that q is the unique state such that $D[q] = 1$, the update of O_e becomes

$$O_e := O_e \cup Output(q) \times \{r\}$$

The bit-parallel Aho-Corasick algorithm then becomes

Algorithm 3.72(P$_+$, E, SP, AC, AC-OPT, BPAC)

$u, r := \varepsilon, S;$
$InitializeBitVectors;$
$q := 0;$
if $\varepsilon \in P \to O_e := \{(\varepsilon, S)\} \,[\!]\, \varepsilon \notin P \to O_e := \varnothing$ **fi**;
{ invariant: $ur = S \wedge O_e = (\bigcup x, y, z : xz = S \wedge x \leq_p u \wedge y \in \mathbf{suff}(x) \cap P : \{(y, z)\})$
$\wedge$ D represents the active state of ACOpt(P)
$\wedge$ $D[q] = 1$ }
do $r \neq \varepsilon \to$
$D := T[q] \& B_P[r\uparrow 1];$
$D[0] := (D = 0 \vee D = 1);$
$u, r := u(r\uparrow 1), r\downarrow 1;$
as $\varepsilon \in P \to O_e := O_e \cup \{(\varepsilon, r)\}$ **sa**;
$q := 0;$

do $D[q] \neq 1 \rightarrow$
$q := q + 1$
od;
$O_e := O_e \cup Output(q) \times \{r\}$
od
$\{ R_e \}$

Compared to the Multiple Shift-And algorithm derived in Subsection 3.9.2, the following differences can be observed:

- Amount of states to be encoded: The ACOpt machine used in the bit-parallel Aho-Corasick algorithm may have less states than used in Multiple Shift-And, since prefixes of keywords may overlap. As a result, the encoding of the states might still fit the size of a particular computer's word size (e.g. 32 or 64) when the encoding of states in Multiple Shift-And does not.
- Amount of bitoperations for active state update: For Multiple Shift-And, 3 full bitoperations are needed to update D_P (<<, |, and &), whereas for bit-parallel Aho-Corasick, 1 full bitoperation is needed to update D (&), but 2 operations on parts of D are needed as well (1 for comparing all bits except bit 0, 1 for setting the last bit).
- Work needed to update O_e: The updates of O_e are quite different in both algorithms. A more detailed analysis and/or implementation of both algorithms should give more insight into which update is easier to perform.
- Construction time/space for ACOpt: For the bit-parallel Aho-Corasick algorithm, the ACOpt machine needs to be temporarily constructed in order to encode it.
- Memory usage: The bit-parallel Aho-Corasick algorithm will use more memory space in most cases, since it uses two bitmatrices and one bitvector, vs. the one bitmatrix and two bitvectors used for Multiple Shift-And.

Concluding, it seems necessary to actually implement, benchmark and optimize both algorithms before their comparative efficiency can be established.

Initialization of D, B_P and T

Since initially state 0 of the ACOpt automaton is active, D can be initialized (using $s = \sum_{i=0}^{|P|-1} |p_i|$) by

$$D := 0^{s-1}1$$

The bitmatrices B_P and T can be initialized by using a breadth first traversal of the state of the trie that is part of the ACOpt automaton. (Note that during this traversal, transitions that are not part of the trie are used as well.) We do not further discuss this here.

Chapter 4

Constructing factor oracles

In this chapter, we describe two alternative ways of constructing *factor oracles* for a single keyword. Such automata may be used in pattern matching algorithms as described in Section 3.8.

A somewhat shorter version of the research in this chapter, co-authored by Gerard Zwaan and Bruce Watson, was accepted to the 2003 *Prague Stringology Conference* ([CZW03a]), to be held in Prague from September 22 to September 24.

The research as reported in this chapter was also submitted for publication in the *Computing Science Report* series of the Department of Mathematics and Computing Science of the Technische Universiteit Eindhoven ([CZW03b]).

4.1 Introduction

A factor oracle is a data structure for weak factor recognition. It can be described as an automaton built on a string p of length m that (a) is acyclic, (b) recognizes at least all factors of p, (c) has $m + 1$ states (which are all final), and (d) has m to $2m - 1$ transitions (cf. [ACR01]). Some example factor oracles are given in Figure 4.1.

Factor oracles are introduced in [ACR01] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window

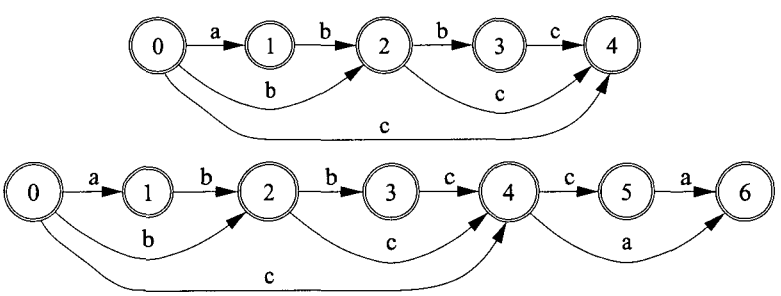

Figure 4.1: Factor oracles for *abbc* (recognizing $abc \notin \mathbf{fact}(p)$), *abbcca* (recognizing $abc, abcc, abcca, abca, abbca, bbca, bca \notin \mathbf{fact}(p)$)

on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information on the longest factor matched and the mismatching character.

Instead of an automaton recognizing exactly the set of factors of the keyword, it is possible to use a factor oracle: although it recognizes more and thus might read backwards longer than necessary, it cannot miss any matches (see Section 3.8 for more information). The advantage of using factor oracles is that they are easier to construct and take less space to represent compared to the automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. This is the result of the latter automata lacking one or more of the four essential properties of the factor oracle.

The factor oracle is introduced in the previously cited article by means of an $\mathcal{O}(m^2)$ construction algorithm that is used as its definition. Furthermore, an $\mathcal{O}(m)$ sequential construction algorithm is described. It is not obvious by just considering the algorithms that it recognizes at least all factors of p and has m to $2m-1$ transitions (i.e. that (b) and (d) hold). For both algorithms, a number of lemmas are needed to prove this.

In this chapter, we give two alternative algorithms for the construction of a factor oracle. Our first algorithm, in Section 4.2, constructs a factor oracle based on the suffixes of p. This algorithm is $\mathcal{O}(m^2)$ and thus not of practical interest, but it is more intuitive to understand and properties (b) and (d)—two important properties of factor oracles—are immediately obvious from the algorithm. The acyclicity of the factor oracle however—corresponding to property (a)—is not immediately obvious. Our proof of this property (part of Property 4.7) is rather involved, whereas the property is immediately obvious from the algorithms in [ACR01]. We prove that the alternative construction algorithm and those given in that article construct equivalent automata in Section 4.3.

Section 4.4 shows that the language of a factor oracle is prefix-, suffix- and therefore factor-closed. A precise characterization of the language (independent of the automaton itself) is still open. We show that the occurrence of repetitions and the occurrence of states (other than the start state) with at least 2 outgoing transitions are necessary for the language to contain strings other than factors.

In Section 4.5 we present our second algorithm, which constructs a factor oracle from the trie recognizing the factors of p. Although this algorithm is $\mathcal{O}(m^2)$ as well, it gives a clear insight in the relationship between the trie and dawg recognizing the factors of p and the factor oracle recognizing a superset thereof.

Finally, Section 4.6 gives a summary and overview of future work.

4.1.1 Related work

As mentioned before, factor oracles were introduced in [ACR01] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. A pattern matching algorithm using the factor oracle is described in that paper as well.

Apart from their use in pattern matching algorithms, factor oracles have been used in a heuristic to compute repeated factors of a string [LL00] as well as to compress text [LL02]. An improvement for those uses of factor oracles is introduced in [LLA02] in the form of the *repeat oracle.*

Related to the factor oracle, the *suffix oracle*—in which only those states corresponding to a suffix of p are marked final—is introduced in [ACR01]. In [AR99] the factor oracle is extended to apply to a set of strings.

4.2 Construction based on suffixes

Our first alternative algorithm for the construction of a factor oracle constructs a 'skeleton' automaton for p—recognizing **pref**(p)—and then constructs a path for each of the suffixes of p in order of decreasing length, such that eventually at least **pref**(**suff**(p)) = **fact**(p) is recognized. If such a suffix of p is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest prefix of such a suffix that is recognized. A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized, to a state from which there is a path leading to state m that spells out the rest of the suffix.

Build_Oracle_2[1]($p = p_1 p_2 \dots p_m$)

1: **for** i from 0 to m **do**
2: Create a new final state i
3: **for** i from 0 to $m-1$ **do**
4: Create a new transition from i to $i+1$ by p_{i+1}
5: **for** i from 2 to m **do**
6: Let the longest path from state 0 that spells a prefix of $p_i \dots p_m$ end in state j and spell out $p_i \dots p_k$ $(i-1 \leq k \leq m)$
7: **if** $k \neq m$ **then**
8: Build a new transition from j to $k+1$ by p_{k+1}

Note that this algorithm is $\mathcal{O}(m^2)$. The factor oracle on p built using this algorithm is referred to as Oracle(p) and the language recognized by it as **factoracle**(p).

The first two properties we give are obvious given our algorithm. They correspond to (b) and (c)-(d) respectively as mentioned in Section 4.1.

Property 4.1. **fact**(p) $\subseteq$ **factoracle**(p).
Proof: The algorithm constructs a path for all suffixes of p and all states are final. □

Property 4.2. For p of length m, Oracle(p) has exactly $m+1$ states and between m and $2m-1$ transitions.
Proof: States can be constructed in steps 1-2 only, and exactly $m+1$ states are constructed there. In step 4 of the algorithm, m transitions are created. In steps 5-8, at most $m-1$ transitions are created. □

Property 4.3 (Glushkov's property). All transitions reaching a state i of Oracle(p) are labeled by p_i.
Proof: The only steps of the algorithm that create transitions are steps 4 and 8. In both, transitions to a state i are created labeled by p_i. □

[1]Although this it the first factor oracle construction algorithm we present, we name it **Build_Oracle_2** to distinguish it from algorithm **Build_Oracle** given in [ACR01] (and repeated in Section 4.3).

Property 4.4 (Weak determinism). For each state of Oracle(p), no two outgoing transitions of the state are labeled by the same symbol.
Proof: The algorithm never creates an outgoing transition by some symbol if such a transition already exists. □

We now define function $poccur(u,p)$ to give the end position of the leftmost occurrence of u in p (equivalent to the same function in [ACR01]):

Definition 4.5. Function $poccur \in V^* \times V^* \rightarrow \mathbb{N}$ is defined as

$$poccur(u,p) = (\mathbf{MIN}\, t,v : p = tuv : |tu|) \quad (p,t,u,v \in V^*)$$

□

Note that if $u \notin \mathbf{fact}(p)$, $poccur(u,p) = \infty$.

Property 4.6. For suffixes and prefixes of factors we have:

$$uv \in \mathbf{fact}(p) \Rightarrow poccur(v,p) \leq poccur(uv,p) \quad (p,u,v \in V^*)$$
$$uv \in \mathbf{fact}(p) \Rightarrow poccur(u,p) \leq poccur(uv,p) - |v| \quad (p,u,v \in V^*)$$

□

We introduce $min(i)$ for the minimum length string recognized in state i—either in one of the partially constructed automata or in the complete automaton.

In the following property, we use j_i and k_i to identify the values j and k attain when considering suffix $p_i \ldots p_m$ of p in steps 5-8 of the algorithm.

Property 4.7. For the partial automaton constructed according to algorithm **Build_Oracle_2** with all suffixes of p of length greater than $m - i + 1$ already considered in steps 5-8 ($2 \leq i \leq m+1$), we have that

i. it is acyclic

ii. for each h with $1 \leq h < i$, all prefixes of $p_h \ldots p_m$ are recognized

iii. for each state n and outgoing transition to a state $q \neq n+1$,
$q \leq k_{max} + 1$ holds where $k_{max} = (\mathbf{MAX}\, h : 1 < h < i \wedge k_h < m : k_h)$

iv. for each state n, $min(n)$ is an element of $\mathbf{fact}(p)$, $min(n)$ is a suffix of each string recognized in n, and $n = poccur(min(n),p)$

v. if $u \in \mathbf{fact}(p)$ is recognized, it is recognized in a state $n \leq poccur(u,p)$

vi. for each state n and each symbol a such that there is a transition from n to a state q by a, $min(n) \cdot a \in \mathbf{fact}(p)$ and $q = poccur(min(n) \cdot a, p)$

vii. for each pair of states n and q, if $min(n) \leq_s min(q)$, then $n \leq q$, and as a result, if $min(n) <_s min(q)$, then $n < q$

viii. if w is recognized in state n, then for any suffix u of w, if u is recognized, it is recognized in state $q \leq n$

Proof:

We first consider the automaton constructed in steps 1-4 of the algorithm. It is straightforward to verify that the properties hold for $i = 2$.

Now assume that the properties hold for the automaton with all suffixes of p of length greater than $m - i + 1$ already considered. We prove that they also hold for the automaton after the suffix of length $m - i + 1$, $p_i \dots p_m$, has been considered.

If $k = m$ in step 6, suffix $p_i \dots p_m$ is already recognized, no new transition will be created, the automaton does not change and the properties still hold.

If $k < m$, then we need to prove that each of the properties holds for the new automaton.

Ad i: By v., string $p_i \dots p_k$ is recognized in state $j \leq poccur(p_i \dots p_k, p)$. Since $p_i \dots p_k \leq_s p_1 \dots p_k$ and $poccur(p_1 \dots p_k, p) = k$, $poccur(p_i \dots p_k, p) \leq k$ due to Property 4.6. Since $j \leq k$, the transition created from j to $k + 1$ is a forward one.

Ad ii: Trivial.

Ad iii: We prove that the property holds for the new automaton by showing that $k = k_i \geq k_{max}$, i.e. k will become the new k_{max}.

If $k_{max} = -\infty$, $k \geq k_{max}$ clearly holds.

If $k_{max} > -\infty$, assume that $k_{max} > k$, then there is an h such that $1 < h < i \wedge k_h < m \wedge k_h = k_{max}$. Factor $p_h \dots p_k$ is recognized in $g \leq k$ due to ii. and v.

If $g = k$, then $p_h \dots p_k$ is recognized in k and $p_h \dots p_m$ is recognized in m; so $k_h = m$ which contradicts $k_h < m$.

If $g < k$, then $p_h \dots p_k$ is recognized in $g < k$. Since $p_i \dots p_k$ is recognized in $j = j_i$ and $p_i \dots p_k \leq_s p_h \dots p_k$, due to viii., $j \leq g$.

If $j = g$, then $p_h \dots p_k$ is the longest prefix of $p_h \dots p_m$ recognized by the old automaton, which contradicts ii.

If $j < g$, then $j < g < k$. We know that $min(g) \leq_s p_h \dots p_k$ (using iv.), $min(j) \leq_s p_h \dots p_k$ (using iv. and $p_i \dots p_k \leq_s p_h \dots p_k$) and therefore that $min(j) <_s min(g)$ (due to vii.). Let l be the state to which the transition by p_{k+1} from g leads, i.e. l is the state in which $p_h \dots p_{k+1}$ is recognized. Using vi., we have that $l = poccur(min(g) \cdot p_{k+1}, p)$. Using Property 4.6 we have that $l \leq poccur(p_h \dots p_{k+1}, p)$ and the latter is $\leq k + 1$ due to the definition of *poccur* (since $k + 1$ marks the end of an occurrence of $p_h \dots p_{k+1}$). We have $poccur(min(j) \cdot p_{k+1}, p) \leq poccur(min(g) \cdot p_{k+1}, p) = l$ since $min(j) \leq_s min(g)$. We want to prove that $k + 1 \leq poccur(min(j) \cdot p_{k+1}, p)$. Assume that $poccur(min(j) \cdot p_{k+1}, p) < k + 1$. If the first occurrence of $min(j) \cdot p_{k+1}$ starts before position i of p, then it is a prefix of a suffix of p longer than $p_i \dots p_m$ and thus by ii. $min(j) \cdot p_{k+1}$ is recognized. Since $min(j)$ is recognized in j, a transition from j by p_{k+1} must exist and we have a contradiction. If the first occurrence of $min(j) \cdot p_{k+1}$ starts at or after position i of p, then there exists a shortest string x such that $x \cdot min(j) \cdot p_{k+1} \in \mathbf{pref}(p_i \dots p_k)$ and $x \cdot min(j) \cdot p_{k+1}$ is recognized in a state $\leq j$. But then $x \cdot min(j)$ is recognized in a state $n < j$. By viii., since $min(j) \leq_s x \cdot min(j)$, this means that $min(j)$ is recognized in state $s \leq n < j$ and we have a contradiction. Thus

$k+1 \leq poccur(min(j) \cdot p_{k+1}, p) \leq l$ and therefore, since $l \leq k+1$ holds, $l = k+1$. In that case, $p_h \dots p_{k+1}$ is recognized in $l = k+1$ and $p_h \dots p_m$ is recognized in m. But then $k_h = m$, and we have a contradiction.

Thus, $k_{max} = k_h \leq k = k_i$ and iii. holds for the new automaton.

Ad iv: Let $s = min(j)$, $t = min(k+1)$ and $u = min(h)$ $(k+1 \leq h \leq m)$ respectively in the old automaton. Due to the proof of iii., $k = k_i \geq k_{max}$ and therefore a unique path between $k+1$ and h exists, labeled r, and—due to iv—$u \leq_s tr$.

If $|sp_{k+1}r| \geq |u|$, u remains the minimal length string recognized in state h. Since $s \leq_s p_i \dots p_k$, $sp_{k+1}r \leq_s p_i \dots p_{k+1}r$. Since $u \leq_s tr$, $tr \leq_s p_1 \dots p_{k+1}r$ and $|sp_{k+1}r| \geq |u|$, $u \leq_s sp_{k+1}r$ and—due to iv.—$u \leq_s s'p_{k+1}r$ as well for any s' recognized in state j.

If $|sp_{k+1}r| < |u|$, $sp_{k+1}r$ is the new minimal length string recognized in state h. Since $s \leq_s p_i \dots p_k$, $sp_{k+1}r \leq_s p_i \dots p_{k+1}r$. Since $u \leq_s tr$, $tr \leq_s p_1 \dots p_{k+1}r$ and $|sp_{k+1}r| < |u|$, $sp_{k+1}r \leq_s u$ and—due to iv.—$sp_{k+1}r \leq_s s'p_{k+1}r$ as well for any s' recognized in state j.

Since $p_i \dots p_{k+1}r$ was not recognized before, it is not a prefix of p, $p_2 \dots p_m$, $\dots$, $p_{i-1} \dots p_m$ (using ii.), hence $poccur(p_i \dots p_{k+1}r, p) = k+1+|r|$. Since $s \leq_s p_i \dots p_k$, $poccur(sp_{k+1}r, p) \leq k+1+|r|$. Assume that $poccur(sp_{k+1}r, p) < k+1+|r|$, then $p_i \dots p_{k+1}r = usp_{k+1}rv$ $(u, v \in V^*, v \neq \varepsilon, |u|$ minimal), since $sp_{k+1}r$ cannot start before p_i because in that case it would have already been recognized by the old automaton. Factor us is recognized in state $g < j$ (using i.) and—since viii. holds—$s \leq_s us$ is recognized in a state $o \leq g < j$. This contradicts s being recognized in j. As a result $poccur(sp_{k+1}r, p) = k+1+|r|$.

Ad v: Any new factor of p recognized after creation of the transition from j to $k+1$ has the form $vp_{k+1}r$ and is recognized in $k+1+|r|$ with $v \in \mathbf{fact}(p)$ recognized in state j. Since $k+1+|r| = poccur(min(k+1)r, p)$ (using iii., iv. holding for the new automaton plus the fact that k is the new k_{max}) and $min(k+1) \cdot r \leq_s vp_{k+1}r$ due to iv. holding for the new automaton, $k+1+|r| \leq poccur(vp_{k+1}r, p)$ using Property 4.6.

Ad vi: The states n we have to consider are $n = j$ and $n = h$ for $k+1 \leq h \leq m$.

For $n = j$, a new transition to $k+1$ is created and by iv., $min(j) \leq_s p_i \dots p_k$., hence we have $min(j) \cdot p_{k+1} \leq_s p_i \dots p_{k+1}$, $p_{k+1-|min(j)|} \dots p_{k+1} = min(j) \cdot p_{k+1}$, $min(j) \cdot p_{k+1} \in \mathbf{fact}(p)$ and $poccur(min(j) \cdot p_{k+1}, p) \leq k+1$. Since $min(j) \cdot p_{k+1}$ is recognized in state $k+1$, due to v. for the new automaton, $k+1 \leq poccur(min(j) \cdot p_{k+1}, p)$. Therefore $k+1 = poccur(min(j) \cdot p_{k+1}, p)$.

For $n = h$ with $k+1 \leq h \leq m$, $min(h)$ changes to $sp_{k+1}r$ if and only if $|sp_{k+1}r| < |u|$ (with r, s, u as in the proof of iv.). We know that $ua \in \mathbf{fact}(p)$ and $q = poccur(ua, p)$. Since $sp_{k+1}r \leq_s u$, $sp_{k+1}ra \leq_s ua$, hence $sp_{k+1}ra \in \mathbf{fact}(p)$ as well and $poccur(sp_{k+1}ra, p) \leq poccur(ua, p) = q$, but due to v., $q \leq poccur(sp_{k+1}ra, p)$ hence $q = poccur(sp_{k+1}ra, p)$.

Ad vii: Assume $min(n) \leq_s min(q)$. We have $poccur(min(n), p) \leq poccur(min(q), p)$ due to Property 4.6, which according to iv. is equivalent to $n \leq q$.

Ad viii: By induction on $|w|$. It is true if $|w| = 0$ or $|w| = 1$. Assume that it is true for all strings x such that $|x| < |w|$. We will show that it is also true for w, recognized in n.

Let $w = xa$ $(x \neq \varepsilon)$, x is recognized in h $(0 < h < n)$. Consider a proper suffix of w, recognized in state q. It either equals ε and is recognized in state $0 \leq n$ or it can be written as va where $v <_s x$.

Suffix va of w is recognized, therefore suffix v of x is recognized and according to the induction hypothesis, v is recognized in state $l \leq h$. Let $\bar{x} = min(h)$ and $\bar{v} = min(l)$. Due to iv. for

the new automaton, $\bar{x} \leq_s x$ and $\bar{v} \leq_s v$. We now prove that $\bar{v} \leq_s \bar{x}$. If $l = h$, then $\bar{v} = \bar{x}$. Now consider the case $l < h$. Since $v \leq_s x$ and $\bar{v} \leq_s v$, $\bar{v} \leq_s x$. Due to vii., $\bar{x} \not\leq_s \bar{v}$. Thus, since $\bar{v}$ and $\bar{x}$ both are suffixes of x, $\bar{v} \leq_s \bar{x}$. Since $\bar{x}$ is recognized in h and there is a transition by a from h to n, by vi. for the new automaton we have that $\bar{x}a \in \mathbf{fact}(p)$ and $n = poccur(\bar{x}a, p)$. Since $\bar{v}$ is recognized in l and there is a transition by a from l to q, $\bar{v}a \in \mathbf{fact}(p)$ and $q = poccur(\bar{v}a, p)$ due to vi. for the new automaton. Since $\bar{v}a \leq_s \bar{x}a$, $poccur(\bar{v}a, p) \leq poccur(\bar{x}a, p)$ due to Property 4.6 and hence $q \leq n$.

We have shown that the properties hold for every partial automaton during the construction. Consequently, they hold for the complete automaton Oracle(p). □

Note that Property 4.7, i. corresponds to property (a) in Section 4.1.

4.3 Equivalence to original algorithms

A factor oracle as introduced in [ACR01] is built by the following $\mathcal{O}(m^2)$ algorithm:

Build_Oracle($p = p_1 p_2 \dots p_m$)

1: **for** i from 0 to m **do**
2: Create a new final state i
3: **for** i from 0 to $m-1$ **do**
4: Create a new transition from i to $i+1$ by p_{i+1}
5: **for** i from 0 to $m-1$ **do**
6: Let u be a minimal length word in state i
7: **for all** $\sigma \in \Sigma, \sigma \neq p_{i+1}$ **do**
8: **if** $u\sigma \in \text{Fact}(p_{i-|u|+1} \dots p_m)$ **then**
9: Build a new transition from i to*
$i - |u| + poccur(u\sigma, p_{i-|u|+1} \dots p_m)$ by σ

To prove the equivalence of the automata constructed by the two algorithms, we need the following properties.

Property 4.8. For any state i of both Oracle(p) (i.e. the factor oracle constructed according to algorithm **Build_Oracle_2** and the factor oracle constructed according to algorithm **Build_Oracle**), if $u = min(i)$ then

$$u\sigma \in \mathbf{fact}(p_{i-|u|+1} \dots p_m) \equiv u\sigma \in \mathbf{fact}(p)$$

Proof: $\Rightarrow$: Trivial. $\Leftarrow$: By Property 4.7, iv. (for **Build_Oracle_2**) and [ACR01, Lemma 1] (for **Build_Oracle**), $i = poccur(u, p)$. By Property 4.6, $poccur(u\sigma, p) \geq i$, hence $u\sigma \in \mathbf{fact}(p_{i-|u|+1} \dots p_m)$. □

Property 4.9. For any state i of an automaton constructed by either algorithm, if $u = min(i)$ and $u\sigma \in \mathbf{fact}(p)$ then

$$i - |u| + poccur(u\sigma, p_{i-|u|+1} \dots p_m) = poccur(u\sigma, p)$$

*Note that in [ACR01] the term $-|u|$ is missing in the algorithm, although from the rest of the paper it is clear that it is used in the construction of the automata

Proof:

$$
\begin{aligned}
& i - |u| + poccur(u\sigma, p_{i-|u|+1} \dots p_m) \\
= & \quad \{ \text{ definition } poccur \ \} \\
& i - |u| + \left(\mathbf{MIN}\, t, v : p_{i-|u|+1} \dots p_m = tu\sigma v : |tu\sigma|\right) \\
= & \quad \{ \ u = min(i), \text{ hence recognized in } i = poccur(u,p) \ \} \\
& i - |u| + (\mathbf{MIN}\, t, v : p = tu\sigma v : |tu\sigma| - (i - |u|)) \\
= & \quad \{ \ u\sigma \in \mathbf{fact}(p), \text{ property of min } \} \\
& i - |u| + (\mathbf{MIN}\, t, v : p = tu\sigma v : |tu\sigma|) - (i - |u|) \\
= & \quad \{ \text{ calculus, definition } poccur \ \} \\
& poccur(u\sigma, p) \qquad \square
\end{aligned}
$$

Property 4.10. The algorithms **Build_Oracle_2** and **Build_Oracle** construct equivalent automata.

Proof: We prove this by induction on the states. Our induction hypothesis is that for each state j $(0 \leq j < i)$, $min(j)$ is the same in both automata, and the outgoing transitions from state j are equivalent for both automata.

If $i = 0$, $u = min(i) = \varepsilon$ in both automata. Consider a transition created by **Build_Oracle_2**, say to state k by $\sigma \neq p_{i+1}$. Since this transition exists, $u\sigma \in \mathbf{fact}(p)$ and $k = poccur(u\sigma, p)$ (due to Property 4.7, vi.). Using Properties 4.8 and 4.9, such a transition was created by **Build_Oracle** as well. Similarly, consider a transition created by **Build_Oracle**, say to state k by σ. This transition, say on symbol σ, leads to state $k = i - |u| + poccur(u\sigma, p_{i-|u|+1} \dots p_m)$ and was created since $u\sigma \in \mathbf{fact}(p_{i-|u|+1} \dots p_m)$ (see the algorithm). Using Properties 4.8 and 4.9, such a transition was created by **Build_Oracle_2** as well.

If $i > 0$, using the induction hypothesis and acyclicity of the automata, i has the same incoming transitions and as a result $min(i)$ is the same for both automata. Using the same arguments as in case $i = 0$, the outgoing transitions from state i are equivalent for both automata.

As a result, the two automata are equivalent. $\square$

4.4 Language of factor oracles

A definition of the language $\mathbf{factoracle}(p)$ not employing an automaton and its construction algorithm was left as an open question in [ACR01]. It is straightforward to see that it is bounded from above by $\mathbf{seq}(p)$, the set of all subsequences of p: factor oracles are acyclic, all transitions go from a state i to $j > i$, and all transitions going to some state j are labeled by p_j.

In this section, we show that the language is prefix-, suffix- and hence factor-closed. We also give two properties showing sufficient conditions for $\mathbf{factoracle}(p) \supset \mathbf{fact}(p)$ to hold.

Property 4.11. The language **factoracle**(p) is prefix-closed:

$$w \in \textbf{factoracle}(p) \Rightarrow \textbf{pref}(w) \subseteq \textbf{factoracle}(p) \quad (p, w \in V^*)$$

Proof: Follows directly from the fact that all states of Oracle(p)—and thus all states on the path from state 0 spelling w—are final. □

In [ACR01, Lemma 5], the authors prove that the language recognized by a factor oracle is suffix-closed. Their proof is rather hard to follow and we therefore give a slightly different, longer version here.

Property 4.12. The language **factoracle**(p) is suffix-closed:

$$\begin{aligned} &w \in \textbf{factoracle}(p) \text{ is recognized in state } n \Rightarrow \\ &\quad \textbf{suff}(w) \subseteq \textbf{factoracle}(p) \text{ and } u \in \textbf{suff}(w) \text{ is} \quad \text{recognized in state } o \leq n \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (p, u, w \in V^*) \end{aligned}$$

Proof:

By induction on $|w|$. It is true if $|w| = 0$ or $|w| = 1$. Assume $|w| \geq 2$ and that it is true for all strings x such that $|x| < |w|$. We show that it is also true for w, recognized in n.

Let $w = xa$ ($x \neq \varepsilon$), x is recognized in h ($0 < h < n$). Consider a proper suffix of w. It either equals ε and is recognized in state $0 \leq n$ or it can be written as va where $v <_s x$.

According to the induction hypothesis, v is recognized in state $l \leq h$. Let $\bar{x} = min(h)$ and $\bar{v} = min(l)$. Due to Property 4.7, iv., $\bar{x} \leq_s x$ and $\bar{v} \leq_s v$. We now prove that $\bar{v} \leq_s \bar{x}$. If $l = h$, then $\bar{v} = \bar{x}$. Now consider the case $l < h$. Since $v \leq_s x$ and $\bar{v} \leq_s v$, $\bar{v} \leq_s x$. Due to Property 4.7, vii., $\bar{x} \not\leq_s \bar{v}$. Thus, since $\bar{v}$ and $\bar{x}$ both are suffixes of x, $\bar{v} \leq_s \bar{x}$. Since $\bar{x}$ is recognized in h and there is a transition by a from h, by Property 4.7, vi. we have that $\bar{x}a \in \textbf{fact}(p)$ and $n = poccur(\bar{x}a, p)$. Since $\bar{x}a \in \textbf{fact}(p)$, $\bar{x}az \leq_s p$ for some $z \in V^*$, hence $\bar{v}az \leq_s p$ as well, hence $\bar{v}a$ is recognized. Since $\bar{v}$ is recognized in l, there is a transition by a from l to a state o. We know that $o = poccur(\bar{v}a, p)$ due to Property 4.7, vi. Since $\bar{v}a \leq_s \bar{x}a$, $poccur(\bar{v}a, p) \leq poccur(\bar{x}a, p)$ due to Property 4.6 and hence $o \leq n$. □

Property 4.13. The language **factoracle** is factor-closed:

$$w \in \textbf{factoracle}(p) \Rightarrow \textbf{fact}(w) \in \textbf{factoracle}(p) \quad (p, w \in V^*)$$

Proof: Follows from Properties 4.11 and 4.12 and $\textbf{fact}(w) = \textbf{pref}(\textbf{suff}(w))$. □

Remark 4.14. For **fact**(p), the equality $\textbf{fact}(p)^r = \textbf{fact}(p^r)$ holds for all $p \in V^*$. The equality $\textbf{factoracle}(p)^r = \textbf{factoracle}(p^r)$ does not hold for all $p \in V^*$. An example of a p for which the equality does not hold is $p = baabba$. The factor oracles for p and p^r are given in Figure 4.2. It is clear that $bab \in \textbf{factoracle}(baabba)^r$ but $bab \notin \textbf{factoracle}((baabba)^r)$. □

Due to Property 4.13, it must be possible to characterize a function $skip(p)$ returning a set of strings such that $\textbf{factoracle}(p) = \textbf{fact}(skip(p))$. The exact definition of this function $skip$ is still unclear, but the following two properties give some insight in the language **factoracle**(p):

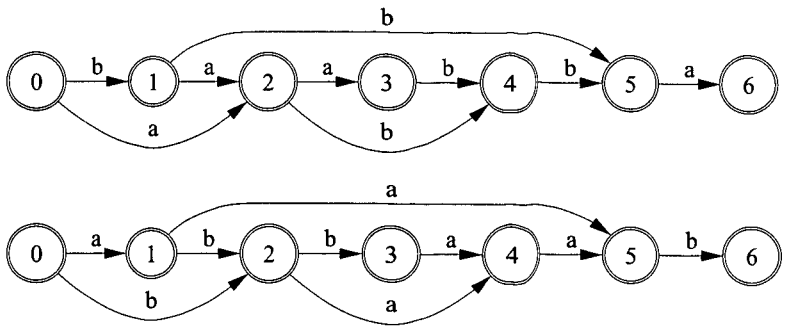

Figure 4.2: Factor oracles for *baabba* and *abbaab*

Property 4.15 (Relationship between non-factor strings and repetitions). If there are no repetitions of any symbol a in p, **factoracle**(p) $\subseteq$ **fact**(p). **Proof:** We prove this based on an induction hypothesis on the partial factor oracle constructed according to the algorithm, with the suffixes of p upto $p_i \ldots p_m$ (i.e. of length $\geq m - i + 1$) already added to the automaton. Our induction hypothesis is that the transitions in this partial factor oracle are exactly those from j to $j+1$ on p_{j+1} for $0 \leq j < m$ and those from 0 to j on p_j for $1 \leq j \leq i$ and that the language recognized by this partial factor oracle is a subset of or equal to **fact**(p).

In case $i = 1$, the automaton clearly recognizes **pref**(p), and **pref**(p) $\subseteq$ **fact**(p).

Assume that the induction hypothesis holds for $0 \leq j \leq i$. The algorithm will construct a transition from 0 to $i+1$ by p_{i+1} in step 8 due to the absence of repetitions in p. New strings recognized will be **pref**($p_{i+1} \ldots p_m$), and **pref**($p_{i+1} \ldots p_m$) $\subseteq$ **fact**(p). Thus, the language recognized is a subset of or equal to **fact**(p) and the transitions are exactly those from j to $j+1$ for $0 \leq j < m$ and those from 0 to j for $1 \leq j \leq i+1$. □

Property 4.16. If there is no state $i > 0$ in the factor oracle on p with at least 2 outgoing transitions, **factoracle**(p) = **fact**(p).

Proof: In this case every path from state 0 to state m is labeled by a suffix of p. □

4.5 Construction based on trie

There is a close relationship between the data structures Trie(**fact**(p)) —the *trie* ([Fre60]) on **fact**(p)—recognizing exactly **fact**(p), DAWG(**fact**(p)) —the *directed acyclic word graph* ([CR94]) on **fact**(p)—recognizing exactly **fact**(p), and Oracle(p)—the factor oracle on p—which recognizes at least **fact**(p). It is well known that DAWG(**fact**(p)) can be constructed from Trie(**fact**(p)) by merging states whose right languages are identical (see for example [CR94]). The factor oracle as defined by Oracle(p) can also be constructed from Trie(**fact**(p)), by merging states whose right languages have identical longest strings (which are suffixes of p). An example of a trie, DAWG and factor oracle for the factors of *abbc* can be seen in Figure 4.3.

Definition 4.17. We define Trie(S) as a 5-tuple <Q, V, δ, ε, F> where S is a finite set of strings, Q = **pref**(S) is the set of states, V is the alphabet, δ is the transition function,

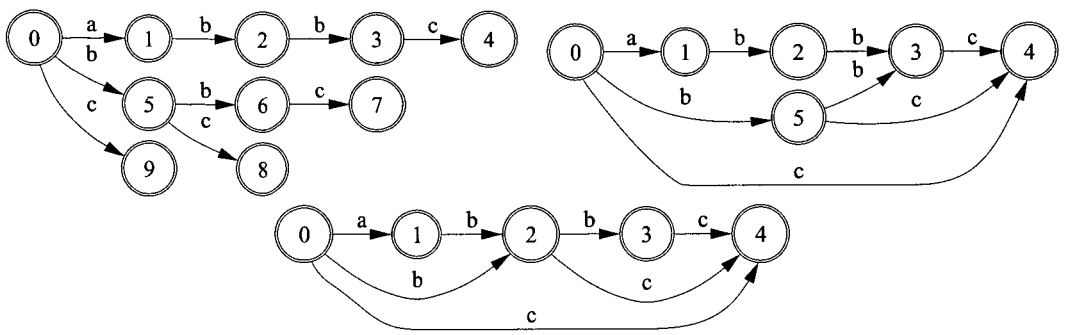

Figure 4.3: (Left to right, top to bottom) Trie, DAWG and factor oracle recognizing **fact**$(abbc)$, **fact**$(abbc)$ and **fact**$(abbc) \cup \{abc\}$ respectively

defined by

$$\delta(u,a) = \begin{cases} ua & \text{if } ua \in \mathbf{pref}(S) \\ \perp & \text{if } ua \notin \mathbf{pref}(S) \end{cases} \qquad (u \in \mathbf{pref}(S), a \in V),$$

ε is the single start state and $F = S$ is the set of final states. □

Property 4.18. For $u, v \in \mathbf{fact}(p)$ we have :

$$uv \in \mathbf{fact}(p) \wedge (\forall w : uw \in \mathbf{fact}(p) : |w| \leq |v|) \qquad \Rightarrow uv \in \mathbf{suff}(p)$$

$$uv_1 \in \mathbf{fact}(p) \wedge (\forall w : uw \in \mathbf{fact}(p) : |w| \leq |v_1|)$$
$$\wedge\, uv_2 \in \mathbf{fact}(p) \wedge (\forall w : uw \in \mathbf{fact}(p) : |w| \leq |v_2|) \quad \Rightarrow v_1 = v_2$$

□

Property 4.19. For $u \in \mathbf{fact}(p)$,

$$(\forall w : uw \in \mathbf{fact}(p) : |w| \leq C) \equiv (\forall w : uw \in \mathbf{suff}(p) : |w| \leq C)$$

Proof: $\Rightarrow$: trivial. $\Leftarrow$: Let $ux \in \mathbf{fact}(p)$, then $(\exists y : : uxy \in \mathbf{suff}(p))$, hence $(\exists y : : |xy| \leq C)$, and since $|y| \geq 0$, $|x| \leq C$.

Using Properties 4.18 and 4.19, $max_p(u)$ can be defined as the unique longest string v such that $uv \in \mathbf{suff}(p)$:

Definition 4.20. Define $max_p(u) = v$ where v is such that

$$uv \in \mathbf{suff}(p) \wedge (\forall w : uw \in \mathbf{suff}(p) : |w| \leq |v|)$$

□

Trie_To_Oracle$(p = p_1 p_2 \ldots p_m)$

1: Construct **Trie**(**fact**(p))
2: **for** i from 2 to m **do**

3: Merge all states u for which $max_p(u) = p_{i+1} \dots p_m$ into the single state $p_1 \dots p_i$

The order in which the values of i are considered is not important. In addition, note that it is not necessary to consider the states u for which $max_p(u) = p_2 \dots p_m$ since there is precisely one such state u in Trie(**fact**(p)), $u = p_1$. Due to Property 4.18, it is sufficient to only consider suffixes of p as longest strings.

Also note that the intermediate automata may be nondeterministic, but the final automaton will be weakly deterministic (as per Property 4.4).

To prove that algorithm **Trie_To_Oracle** constructs Oracle(p), we define a partition on the states of the trie, induced by an equivalence relation on the states.

Definition 4.21. Relation $\sim_p$ on states of Trie(**fact**(p)) is defined by

$$t \sim_p u \equiv max_p(t) = max_p(u) \quad (t, u \in \textbf{fact}(p))$$

Note that relation $\sim_p$ is an equivalence relation. □

We now show that the partitioning into sets of states of Trie(**fact**(p)) induced by $\sim_p$, is the same as the partitioning of Trie(**fact**(pa)) induced by $\sim_{pa}$, restricted to the states of Trie(**fact**(p)), i.e.

Property 4.22. $\quad t \sim_p u \equiv t \sim_{pa} u \quad (t, u \in \textbf{fact}(p), a \in V)$

Proof:

$t \sim_p u$

$\equiv$ { definition $\sim_p$ }

$max_p(t) = max_p(u)$

$\equiv$ { }

$max_p(t)a = max_p(u)a$

$\equiv$ { ($\star$) }

$max_{pa}(t) = max_{pa}(u)$

$\equiv$ { definition $\sim_{pa}$ }

$t \sim_{pa} u$

where we prove ($\star$) by

$v = max_{pa}(u)$

$\equiv$ { definition max_{pa} }

$uv \in \textbf{suff}(pa) \wedge (\forall w : uw \in \textbf{suff}(pa) : |w| \leq |v|)$

$\equiv$ { $u \in \textbf{fact}(p)$, hence $(\exists x : : uxa \in \textbf{suff}(pa))$,
hence $|xa| > 0$ and $|v| > 0$; $\textbf{suff}(pa) = \textbf{suff}(p)a \cup \{\varepsilon\}$ }

$$
\begin{aligned}
& uv \in \mathbf{suff}(p)a \wedge (\forall w : uw \in \mathbf{suff}(pa) : |w| \leq |v|) \\
\equiv & \quad \{\ |v| > 0, \text{introduction } v'\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall w : w \neq \varepsilon \wedge uw \in \mathbf{suff}(pa) : |w| \leq |v|) \wedge v = v'a \\
\equiv & \quad \{\ \mathbf{suff}(pa) = \mathbf{suff}(p)a \cup \{\varepsilon\}\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall w : w \neq \varepsilon \wedge uw \in \mathbf{suff}(p)a : |w| \leq |v|) \wedge v = v'a \\
\equiv & \quad \{\ w = w'a\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall w' : uw'a \in \mathbf{suff}(p)a : |w'a| \leq |v'a|) \wedge v = v'a \\
\equiv & \quad \{\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall w' : uw' \in \mathbf{suff}(p) : |w'| \leq |v'|) \wedge v = v'a \\
\equiv & \quad \{\ v = v'a\ \} \\
& uv' \in \mathbf{suff}(p) \wedge (\forall w' : uw' \in \mathbf{suff}(p) : |w'| \leq |v'|) \wedge v = v'a \\
\equiv & \quad \{\ \text{definition } max_p\ \} \\
& v' = max_p(u) \wedge v = v'a \\
\equiv & \quad \{\ \text{elimination } v'\ \} \\
& v = max_p(u)a
\end{aligned}
$$

□

Property 4.23. Algorithm **Trie_To_Oracle** constructs Oracle(p).
Proof: By induction on $|p| = m$. If $m = 0$, $p = \varepsilon$, and Trie(**fact**(ε)) = Oracle(ε). If $m = 1$, $p = a$ ($a \in V$), and Trie(**fact**(a))=Oracle(a). If $m > 1$, $p = xa$ ($x \in V^*, a \in V$), and we may assume the algorithm to construct part Oracle(x) of Oracle(xa) correctly (using **fact**(ua) = **fact**(u) $\cup$ **suff**(u)a, Trie(**fact**(xa)) being an extension of Trie(**fact**(x)), and Oracle(xa) being an extension of Oracle(x) (which is straightforward to see from algorithm **Build_Oracle_2** as well as [ACR01, page 57, after Corollary 4]), and Property 4.22). Now consider the states of this partially converted automaton in which suffixes of x are recognized. By construction of the trie, there are transitions from these states by a. The factor oracle construction according to algorithm **Oracle_Sequential** in [ACR01] creates Oracle(xa) from Oracle(x)+a (i.e. the factor oracle for x extended with a single new state m reachable from state $m-1$ by symbol $p_m = a$) by creating new transitions to state m from those states in which suffixes of x are recognized and that do not yet have a transition on a. Since **Trie_To_Oracle** merges all states t for which $max_{xa}(t) = a$ into the single state m, Oracle(xa) is constructed correctly from Trie(**fact**(xa)). □

4.6 Conclusions and future work

We have presented two alternative construction algorithms for factor oracles and shown the automata constructed by them to be equivalent to those constructed by the algorithms

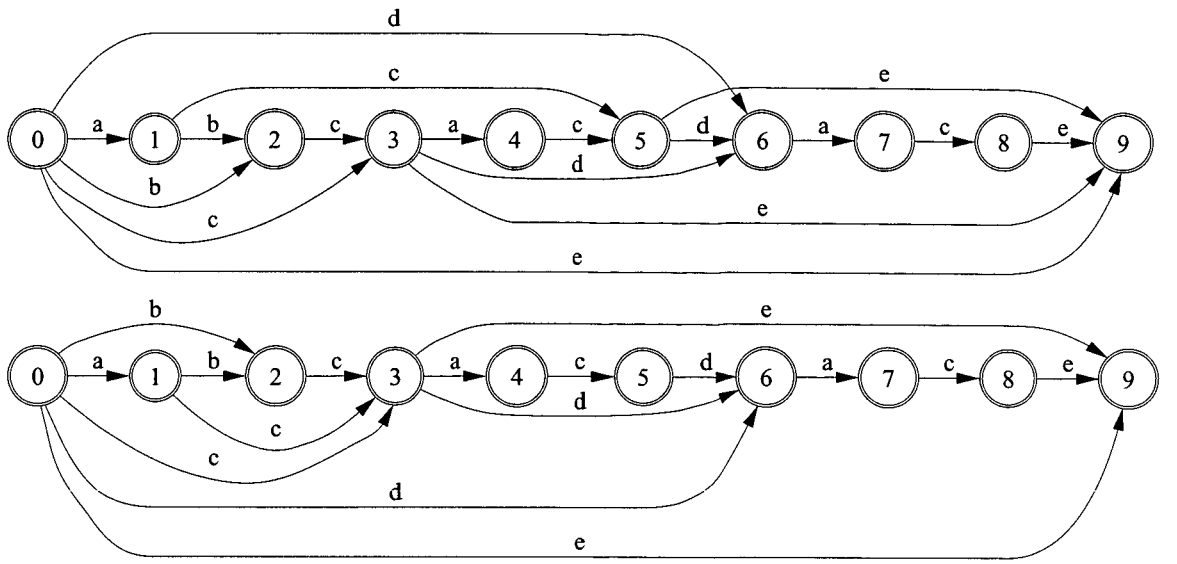

Figure 4.4: Factor oracle recognizing a superset of **fact**(p) (including for example *cace* $\notin$ **fact**(p)) and alternative factor oracle with $m + 1$ states satisyfing Glushkov's property yet recognizing a *different* superset of **fact**(p) (including for example *acacdace* $\notin$ **factoracle**(p), but not *cace*) and having less transitions, for p = *abcacdace*.

in [ACR01]. Although both our algorithms are $\mathcal{O}(m^2)$ and thus practically inefficient compared to the $\mathcal{O}(m)$ sequential algorithm given in [ACR01], they give more insight into factor oracles.

Our first algorithm is more intuitive to understand and makes it immediately obvious, without the need for several lemmas, that the factor oracle recognizes at least **fact**(p) and has m to $2m - 1$ transitions.

Our second algorithm gives a clear insight into the relationship between the trie or dawg recognizing **fact**(p) and the factor oracle recognizing a superset thereof.

Although an automaton-independent characterization of the language **factoracle**(p) remains to be defined, we have given clear proofs that **fact**(p) $\subseteq$ **factoracle**(p). In addition, we have shown some sufficient conditions for **fact**(p) $\subset$ **factoracle**(p) to hold.

We are still working on an automaton-independent characterization of the language. Such a characterization would enable us to calculate how many strings are recognized that are not factors of the original string. This could be useful in determining whether to use a factor oracle-based algorithm in pattern matching or not.

As stated in [ACR01], the factor oracle is not minimal in terms of number of transitions among the automata with $m + 1$ states recognizing at least **fact**(p). We note that it is not even minimal among the subset of such automata having Glushkov's property (see Figure 4.4).

Part III

The implementation

Chapter 5

From SPARE Parts to SPARE Time: a new toolkit for String PAttern REcognition in C++

In this chapter, the design and implementation of SPARE TIME are discussed. Since the new toolkit is based on the 1995 SPARE PARTS we will only briefly discuss some issues in library[1] design and implementation in C++. We then consider the 1995 version of SPARE PARTS and how it was updated to use current C++ and Standard Template Library features. We discuss the extension—based on the new taxonomy—of this 2003 version of SPARE PARTS to SPARE TIME, and how to obtain this new toolkit.

As mentioned above, the work described in this chapter builds on previous work in the form of the design and implementation of SPARE PARTS, which is described in [Wat95, Chapter 8 and 9]. The update of the 1995 version of SPARE PARTS to the 2003 version is also described in some detail in a paper to be published in *Software: Practice and Experience* ([WC03]).

5.1 Introduction

According to [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.

Although Gamma et al. used the term *reusable* in the meaning of reuse in user applications, the term can be used to refer to the ease with which a toolkit is redesigned or extended as well. Since we will extend an existing toolkit, this meaning is important as well.

The first meaning of reusable, i.e. that of reuse within user applications, is met by the original SPARE PARTS by design: the toolkit was designed to provide a collection of classes

[1] We will use the terms *library*, *class library* and *toolkit* interchangeably.

as described in the above definition. [Wat95, Chapter 8 and 9] discusses the design principles and implementation aspects used to achieve this goal.

Our experience in updating and extending the original SPARE PARTS shows that that toolkit indeed also satisfies the second meaning of reuse to a large degree. We will discuss this in somewhat more detail in Sections 5.4 and 5.5.

The 1995 SPARE PARTS toolkit was developed with a number of aspects and design goals in mind (see [Wat95, p. 217]). As we are not building a new toolkit, we will not repeat all these here. Instead, we mention the ones that most relate to the update and extension of toolkits:

- 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.

It is clear that these aspects influence the changes we make in working towards the new SPARE TIME toolkit. We will point out how we have taken them into account when we discuss the update of SPARE PARTS and the extension of the 2003 SPARE PARTS in Sections 5.4 and 5.5 respectively.

5.1.1 Use of C++

The original SPARE PARTS was implemented in the C++ programming language, due to that language's support for object-orientation and generics as well as its widespread availability. Since effort was made not to use obscure features of C++ or features that were not generally found in other languages, it would have been possible to use another programming language for the new SPARE TIME toolkit. We have refrained from doing this for a number of reasons:

- C++ is one of the few mainstream languages supporting generics; for Java for example, limited support for generics has been added only in the most recent version of the language, published in 2003.
- It would have taken quite some extra time to port the existing toolkit to another language and then extend it, or to implement a new toolkit—of algorithms from both the old and the new taxonomy parts—in such a language.[2]
- C++, although receiving a lot of competition from languages like Java, C# and Delphi, is still in widespread use.
- C++ compilers are available for numerous platforms, whereas languages like C# and Delphi are restricted to one or two platforms.

[2]We note that building a new toolkit from scratch in C++ itself would surely have taken more time than the approach of revising and expanding the existing toolkit, which we chose.

5.1.2 Useful references

Object orientation and C++

In this chapter, we assume that the reader is familiar with both C++ and object-oriented terminology. A brief list of some object-oriented terms used in this thesis is given in Appendix B. [Wat95, Chapter 8] lists various books on object-oriented design and programming as well as C++ that were available at the time. Based on that list, we give a short list of books that are particularly useful and/or have become available since 1995:

- [Boo94], [Bud97], and [Mey98a] provide information on object-oriented design and programming in general.
- [Lip99] and [Str97] are (basic resp. thorough) introductory books on C++.
- [Cop92], [Cop98], [SE90], and [VJ03] discuss various advanced C++ topics.
- [Mey98b], [Mey96], and [Dew02] offer C++ tips, tricks and techniques.

Toolkit design

Since we are not designing a library from scratch, we do not discuss the general process of toolkit design here. We refer the reader to a number of relevant books instead, some of which are mentioned in [Wat95] as well:

- [CE95], [Str97], and [Str94, Chapter 8] discuss general C++ library design.
- [GHJV95] and [BMR+96] discuss *design patterns* (not related to pattern matching as we discuss it), which play an important role in toolkit design.
- [Aus99], [Jos99], and [Bre00] provide in-depth treatment of both generic programming and the application and implementation of the Standard Template Library.
- [Ale01] combines the use of generic programming and design patterns to implement generic components in C++.
- [Fow96] discusses best practices and guidelines for improving the design of existing code, particularly when extending such code.

5.2 Code structure and class presentation

C++ source code is often split between a class declaration in a `.hpp` file and a class definition in a `.cpp` file. We use a more fine-grained form of splitting classes. We separate the public and private parts of the declaration, and the code of out-of-line member functions from those that can be inlined, in order to make the code easier to access and understand:

- The public part of class `class`'s declaration will be in file `class.hpp`.
- The private portion of the class's declaration will be in `class_p.hpp`.

- The `class.cpp` file contains all out-of-line member function definitions.
- The `class_i.hpp` file contains member functions which can be inlined for increased performance.

The latter file is conditionally included into the `class.hpp` or the `class.cpp` file, depending on whether the macro `INLINING` is defined or not. Inlining should be disabled during debugging, or if the resulting executables would otherwise be too large.

Almost every class (of the two SPARE PARTS toolkit versions and SPARE TIME) has a class invariant member function called *c_inv* which returns *true* if the class satisfies its structural invariant and *false* otherwise. These invariants have been particularly useful in debugging and extending the code, both because they often give insight into a class's structure and because when bugs arise, these are often detected by the class invariant function. Although the *c_inv* member functions have therefore been left in the released code, they are only used within assertions. They can be disabled by disabling assertions (which is done by defining macro `NDEBUG` and/or by not defining macro `DEBUG`, depending on the C++ compiler used).

In Section 5.5, we will describe any new or expanded classes in a common format. The class descriptions (similar to those in [Wat95, Chapter 9]) consist of:

1. The class name, and whether it is a User class or an Implementation class (see Appendix B).
2. A **Files** clause listing the base file name (which is followed by `.hpp`, `_p.hpp`, `_i.hpp` or `.cpp` as described before).
3. A **Description** clause giving a description of the class's purpose.
4. An optional **Implementation** clause describing implementation details of the class.
5. An optional **Performance** clause describing performance details of the class.

5.3 The design and implementation of SPARE Parts

The original, 1995 version of SPARE PARTS was the second generation string pattern matching toolkit from the Technische Universiteit Eindhoven. The first toolkit, the EINDHOVEN PATTERN KIT, had been a procedural C library, based on an earlier version of the original pattern matching taxonomy ([WZ92]). Experience with this toolkit led to the development of SPARE PARTS in C++ in order to overcome some deficiencies, mainly due to C's memory management and the difficulty in understanding the toolkit's interface.

SPARE PARTS follows the structure of (a newer version of) the taxonomy more closely, makes use of C++'s better memory management facilities, uses an easier interface and supports multi-threaded use of a single pattern matching object.

We briefly describe the most important aspects of the original SPARE PARTS to give the reader an idea of its design and implementation, before moving on to the 2003 version of SPARE PARTS and SPARE TIME.

- In order to prevent the overhead of virtual function calls, a shallow inheritance hierarchy was used (template classes were used instead, see below). The inheritance hierarchy thus consists of:
  - An empty class *PM* at the top of the pattern matcher hierarchy.
  - Abstract classes *PMMulti* and *PMSingle* derived from *PM*. Each provides a member function *match*, but their signature is different.
  - Actual single keyword pattern matchers derived from *PMSingle* include *PMKMP* (Knuth-Morris-Pratt), *PMBM* (Boyer-Moore) and *PMBFSingle* (a brute force pattern matcher).
  - Actual multiple keyword pattern matchers derived from *PMMulti* include *PMAC* (for Aho-Corasick), *PMCW* (Commentz-Walter) and *PMBFMulti* (a brute force pattern matcher).
- Template classes were used for the various automaton, shifter and skip loop classes used by the pattern matchers:
  - Machines *ACMachineOpt*, *ACMachineFail* and *ACMachineKMPFail* for *PMAC*.
  - Match orders *STravFWD*, *STravREV*, *STravOM* and *STravRAN* for *PMBM*.
  - Skip loops *SLNone*, *SLSFC*, *SLFast1* and *SLFast2* for *PMBM*.
  - Shifters *BMShiftNaive*, *BMShift11* and *BMShift12* for *PMBM*.
  - Shifters *CWShiftNaive*, *CWShiftNLA*, *CWShiftWBM*, *CWShiftNorm*, *CWShiftOpt* and *CWShiftRLA* for *PMCW*.

  Although inheritance was not used here, inside each such category of classes the same interface was used.
- Foundation classes, such as states, strings, containers, maps, tries and failure functions:
  - Often these are not classes but primitive datatypes, to save overhead (*State* is `typedef`'d to be an integer for example)
  - No use of STL was made, since it had not been standardized and was not generally supported at the time. Instead, proprietary classes *String*, *Set* and *Array* were used, and no use of STL iterators was made.
- A call-back interface is used for the pattern matchers: the user of a particular pattern matcher supplies both a text string and a pointer to a client-defined function to member function *match*. Whenever a match is detected, this client-defined function is called to handle the match(es) ending at a particular position. The advantage of the call-back interface is its support for multi-threaded pattern matching using a single pattern matching object. (See [WC03, Subsection 3.1] for information on alternative interfaces.)
- By default, the entire ASCII character set is used as the alphabet (i.e. the alphabet is represented by type `char`). This can be inefficient (in terms of the size of automata and other data structures) when smaller alphabets are used, such as in the case of DNA (where the alphabet consists of *a*, *c*, *g* and *t*). Therefore, SPARE PARTS has rudimentary support for other alphabets: a different alphabet size and implementations

of functions *alphabetNormalize* and *alphabetDenormalize* can be defined (by default these are the identity functions).[3]

For more detailed information on the original SPARE PARTS we refer the reader to [Wat95, Chapter 9]. In particular, descriptions of all classes and data structures used in the original toolkit can be found in [Wat95, Sections 9.3-9.5].

From the above description, it should be clear that the original SPARE PARTS toolkit was well-structured. The use of primitive datatypes for many foundation classes, combined with high-performance implementations of classes such as *String*, *Set* and *Array*, results in the toolkit offering efficiency comparable to that of hand-coded special purpose routines.

5.4 Bringing SPARE Parts up-to-date

The original SPARE PARTS toolkit was developed in the early nineties and reached its final form in 1995. At that time, the C++ programming language and the standard library—including the Standard Template Library which had recently been developed—were still being standardized.

Because of the first reason, "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." This led for example to the call-back functions having to return a value of type integer. The boolean type had only recently been added to the draft C++ standard and was not yet supported by the compilers used to compile SPARE PARTS (versions of Borland C++ and Watcom C++, see [Wat95, page 251]).

"Likewise, the use of classes from the proposed standard library, or from the Standard Template Library, was greatly restricted." ([Wat95, page 221]). As we saw in the previous section, classes such as *String*, *Array* and *Set* were therefore implemented from scratch. The intention in doing so was that they could later be replaced by standard library classes relatively easy. In addition, no use of iterators was made, but proprietary traverser classes were used instead.

Finally, it is clear that SPARE PARTS makes extensive use of templates, which was a deliberate choice (as discussed in the previous section and—in more detail—in [Wat95, Subsection 8.2.1] and [WC03]). The exact definition of templates in the C++ standard and implementation of template support in various compilers have not been totally stable in the past however, which leads to some problems with current compilers, as we will see.

5.4.1 Using the Standard Template Library

The Standard Template Library has been standardized for a number of years now, and is supported to a high level by most compilers. The implementations offer good performance as well, and most C++ programmers are familiar with its structure and working. We think that the use of STL classes improves understandability of the toolkit's client interface, and does not compromise efficiency. We therefore decided to replace the classes *String*, *Array* and *Set* by their STL equivalents *std::string*, *std::vector* and *std::set*. Due to the coherent

[3] In fact, only alphabet sizes of at most 256 characters are currently supported, since the alphabet is represented by type `char`.

design and clear structure of the original toolkit, it was relatively straightforward to replace the proprietary classes.

For the string class, this could basically be achieved by `typedef`'ing *String* as *std::string*, since the interfaces were (mostly) the same.

For the other two classes, more work was required: the interfaces of the original classes were somewhat different from those of the STL classes. A number of options was available to deal with this problem:

- Create a *wrapper* class around the existing classes, using the *Adapter* and/or *Composite* design pattern ([GHJV95]). This class then translates the calls from one interface to the other.

- Replace every reference to the classes directly, and update calls to the class's interface within the classes that used *Array* and *Set*.

Although the second choice meant more work (every call to a member function in the interface of *Array* or *Set* that has a different name or signature in the interface of *std::vector* or *std::set* has to be replaced), we opted for it nonetheless. The first choice would have led to an added level of indirection, as each call to a member function of the wrapper class would lead to a call to a member function of the wrapped class. Since classes *Array* and *Set* were used throughout the toolkit, this would have had an impact on performance.

All in all, the impact of and effort involved in replacing the above proprietary classes by appropriate STL counterparts was relatively small. This was due for a large part to the consistent design and structure of the original toolkit. Effort was also made in the design of the original toolkit to facilitate future replacement of these classes by STL components, based on the version of STL available at that time (see [Wat95, Subsection 8.3].

5.4.2 C++ Language Issues

The C++ language itself has changed little based on the version of the language that was in widespread use by 1995. As we mentioned, the boolean type was not used in the original SPARE PARTS since it had only been recently added to the draft standard for the language. We have replaced all use of integers as booleans by genuine booleans in the 2003 version. This mainly meant changing the signatures of the *c_inv* member functions (checking the structural invariant of a class) as well as the signatures of the call-back functions used in the example applications provided with SPARE PARTS (`test`, `kmpgrep`, `acgrep`, `bmgrep` and `cwgrep`).

In addition to the introduction of the boolean type, some definitions and implementations of template functionality have also changed in recent years. This resulted in some of the template code no longer compiling under modern compilers. In the original toolkit, a number of template classes had `operator<<` declared as `friend` inside their class declaration, without explicitly templatizing that function declaration itself (i.e. the compilers at the time apparently used the explicit templatization of the whole class declaration for friend functions declared inside the class as well):

```
template<class T>
class SomeClass {
  public:
    ...
    friend std::ostream& operator<<( std::ostream& os,
                                     const SomeClass<T>& t );
    ...
};
```

This is no longer valid C++ according to the standard and does not work with current compilers. We solved this in the new version by explicitly templatizing the declaration of the friend function:

```
template<class T>
class SomeClass {
  public:
    ...
    template<class T2>
      friend std::ostream& operator<<<>( std::ostream& os,
                                         const SomeClass<T2>& t );
    ...
};
```

5.5 New or changed classes in SPARE Time

As we saw in Chapter 3, the most important changes to the taxonomy compared to the original taxonomy are the generalization of Commentz-Walter suffix-based pattern matching (including the introduction of different automata than just the trie recognizing **suff**(P)), the introduction of new shift functions (mostly for the multiple-keyword generalized Commentz-Walter algorithm, but including one for the single-keyword Boyer-Moore algorithm) and the introduction of bit-parallel prefix-based pattern matching algorithms. The single-keyword Boyer-Moore-Horspool shifter and the bit-parallel algorithms have not yet been implemented.

In this section, we first discuss how the Commentz-Walter pattern matcher in SPARE PARTS, class *PMCW*, was generalized to support different automata. We then turn to the Commentz-Walter shifter classes. We discuss the new shifter classes *CWShiftBMH*, *CWShiftNFS* and *CWShiftMax* that have been added to the toolkit, and how the signature of each of the Commentz-Walter shifters' shift function had to be changed. We also note how the actual precomputation of the shift function values has not yet been generalized. Finally, we discuss the use of automata classes other than class *Trie* with the *PMCW* class, for example class *Factoracle*.

In creating the SPARE TIME toolkit from the SPARE PARTS toolkit by including the above new or revised classes, it has been our goal to adhere to the design goals mentioned in

Section 5.1 on page 70. The three new shifter classes and the new automaton class *Factoracle*, and the corresponding existing classes (i.e. the various *CWShift...* classes and the *Trie* class) all have the same interface[4]. As a result, the coherency of the design of and clearness of the relationship between classes of the toolkit is still the same. The client interface to the library has not even changed, and in writing the new code, we have tried to achieve the same level of efficiency as in the SPARE PARTS code.

5.5.1 The Commentz-Walter pattern matcher

We showed in Section 3.6 that it is possible to generalize Commentz-Walter suffix-based pattern matching by using functions other than **suff**(P) for strengthening the guard of the backward matching. In SPARE TIME, the *PMCW* class has therefore been adapted to support automata other than the reverse trie on **suff**(P) (implemented using class *Trie*). The *PMCW* class in SPARE PARTS had one template parameter, to indicate the particular shifter class to be used. The new version takes a second template parameter, indicating the type of automaton to use. The description of class *PMCW*, given below, is based on that in [Wat95, Subsection 9.4.4].

It is important to note that, although the *PMCW* class has been generalized to support other automata, the actual precomputation of various Commentz-Walter shifter classes has not. Those shift functions that depend on the automaton itself will therefore currently not work correctly with automata other than the reverse trie. To use the *PMCW* pattern matcher with for example a reverse factor oracle or reverse DAWG[5], only shift functions whose preprocessing do not depend on the automaton should therefore be used. These include *CWShiftNaive*, *CWShiftNFS* and *CWShiftMax*.

Implementation class *PMCW*

Files: `pm-cw`

Description: Class *PMCW* implements the Commentz-Walter algorithm skeleton. As we described in Section 5.3, it inherits from *PMMultiple* and implements that class's public interface. The first template argument should be one of the *CWShift..* shifter classes, providing a safe shift distance during the text processing. The second argument indicates the type of automaton to be used in the backward matching of the text, including for example a trie recognizing **suff**$(P)^R$, a DAWG recognizing **fact**$(P)^R$ or a factor oracle recognizing at least **fact**$(P)^R$.

Implementation: A *PMCW* contains a shifter object, a reverse automaton *RAut*, and a *CWOutput* function (an output function for detecting a match). The constructor passes the set of keywords to these subobjects.

Performance: Performance can be improved most easily by improving performance of the subobjects (i.e. the shifter, the reverse automaton and/or the output function). In addition,

[4]In fact, the signature of the shifter classes's *shift* function had to be slightly modified from the version in SPARE PARTS, as we will see in Subsection 5.5.2.

[5]The DAWG is currently not implemented, see Subsection 5.5.3.

it might be possible to move the output of matches out of the inner loop of the algorithm, as suggested in [Wat00].

5.5.2 The Commentz-Walter shifters

As mentioned before, SPARE TIME implements three new shift functions, by means of the classes *CWShiftBMH*, *CWShiftNFS* and *CWShiftMax*.

In Chapter 3, the shift functions take l, v and r as parameters. Since none of the shift functions implemented in the original SPARE PARTS toolkit used the actual string v, the function *shift* of each of the *CWShift..* classes had the following signature:

```
int shift( const RTrie& t,
           const char l,
           const State v,
           const char r ) const;
```

That is, the *shift* function took a reference to the reverse trie, the character $l\restriction 1$, the state of the trie automaton reached after reading v, and the character $r\restriction 1$. The No-Factor Shift—implemented by class *CWShiftNFS* presented below—uses $|v|$ however, while the Boyer-Moore-Horspool function also needs to know the value $v\restriction 1$ in case $|v| = 0$. We therefore needed to extend the function signature. It seemed somewhat inefficient however to keep track of the complete v in the class *PMCW*, when only $|v|$ and $v\restriction 1$ are ever used. We have therefore extended the signature to the following:

```
int shift( const RTrie& t,
           const char l,
           const State v,
           const int vlen,
           const char vrtake1,
           const char r ) const;
```

In SPARE TIME, the values $|v|$ and $v\restriction 1$ are thus available to the shift distance computation as well.

Implementation class *CWShiftBMH*

Files: `cwshbmh`

Description: *CWShiftBMH* implements the Boyer-Moore-Horspool shift distance of Definition 3.13.

Implementation: The implementation uses *CharBM* to give the shift distance.

Implementation class *CWShiftNFS*

Files: `cwshnfs`

Description: *CWShiftNFS* implements the shift distance 1 **max**($lmin_P - |v|$) of Algorithm Detail 3.36. As can be seen from this definition, this shift function may be combined with other shift functions, using class *CWShiftMax* described below.

Implementation: The constructor computes and stores the value $lmin_P$ as variable *lminP* for efficiency reasons. The shift function simply returns $max(1, lminP - vlen)$.

Implementation class *CWShiftMax*

Files: `cwshmax`

Description: *CWShiftMax* implements a shift function that takes the maximum of two shift distances. The two shift classes used for these distances are supplied as template arguments to class *CWShiftMax*.

Implementation: The implementation is almost trivial: the shift distance member function returns the maximum of the return values of the shift distance member functions of the two template arguments supplied to the constructor.

5.5.3 New automata

As described in Subsection 5.5.1, it is possible to use automata other than the reverse trie recognizing **suff**(P) with the Commentz-Walter pattern matcher *PMCW*. In Chapter 3, the DAWG recognizing **fact**$(P)^R$ and the factor oracle recognizing a superset thereof are mentioned as possibilities. Due to lack of time, these are not currently implemented as part of SPARE TIME, although a simple version of the single-keyword factor oracle—in the form of class *Factoracle*—is part of the toolkit. We plan to add the multiple keyword DAWG and factor oracle construction algorithms in the future.

Implementation class *Factoracle*

Files: `factoracle`, `tries`

Description: The *Factoracle* template class implements a (currently single-keyword only) factor oracle (Chapter 4). The class has a constructor taking a set of strings (of which currently only the first one is used) which constructs the factor oracle corresponding to the keywords. The direction in which the strings are traversed is determined by the string traverse class supplied as the template argument of class *Factoracle*. Using traverser *STravFWD* leads to a forward factor oracle, while *STravREV* leads to a reverse factor oracle. Both forward and reverse factor oracles are **typedef**'d in `tries.hpp`.

Implementation: Like the *Trie* ([Wat95, page 248]), this class is implemented using a *StateTo* < *SymbolTo* < *State* > >. The factor oracle is constructed according to the new algorithm given in Section 4.2.

Performance: Memory space could be conserved by not explicitly storing the transitions on the path spelling out the whole keyword(s), but this would lead to a loss of efficiency in taking transitions of the factor oracle during pattern matching.

5.6 Obtaining SPARE Time

Although SPARE TIME is not currently available to the public, it will be made available at some point in the not too distant future via `http://fastar.cs.up.ac.za`. More details on how to use the toolkit will be made available at that time as well[6].

The current version of SPARE TIME has been verified to compile succesfully using the MICROSOFT VISUAL STUDIO .NET 2002 C++ compiler. Since we have made an effort to use only standard C++ language constructs and STL features, it should be relatively easy to get the toolkit working under other recent compilers with good standards support.

[6]For an idea of the toolkit's use, see the use of the 2003 version of SPARE PARTS as described in [WC03].

Part IV

Epilogue

Chapter 6

Conclusions

The work that is reported in this thesis contains a number of significant results, which we summarize here by chapter.

Chapter 3 - A new taxonomy

In this chapter, we revised and expanded the original taxonomy of [WZ96, Wat95] , deriving various algorithms and giving them a place in the taxonomy.

- The original taxonomy and the use of formal techniques in constructing it not only helped in overcoming the three deficiencies mentioned in Section 1.1, but also made it relatively easy to extend and generalize the taxonomy, as we did in Chapter 3.
- We gave two different derivations of the Boyer-Moore-Horspool algorithm, and showed that they can both be added as variants of the Commentz-Walter (Section 3.4.2) resp. Boyer-Moore (Section 3.5) algorithm skeleton.
- In Sections 3.7 and 3.8, we showed that factor- and factor oracle-based pattern matching can be seen as variants of a generalized Commentz-Walter—just like the original suffix-based Commentz-Walter pattern matching. Although this had been (implicitly) known, we have explicitly shown that it also means that all shift functions can be reused.
- As shown in Subsection 3.9.3, there exists a bit-parallel Aho-Corasick algorithm, as suggested by Bruce W. Watson.
- In Subsection 3.9.2, we succeeded in formally deriving the Shift-And, Shift-Or and Multiple Shift-And bit-parallel prefix-based pattern matching algorithms and giving them a place in the taxonomy.

Chapter 4 - Constructing factor oracles

In Chapter 4, we gave two new algorithms for factor oracle construction. In addition, we mentioned and proved some properties related to the language recognized by a factor oracle.

- Two new algorithms for factor oracle construction (Section 4.2 and 4.5), although not practically efficient, give more insight into factor oracles and their properties.

- We gave a new, clearer proof in Section 4.4 that the language recognized by a factor oracle on a string is a superset of the set of factors of the string.
- In Section 4.6, we showed that the factor oracle is not even the smallest acyclic automaton with Glushkov's property recognizing the set of factors of the string.

Chapter 5 - From SPARE Parts to SPARE Time

In this chapter, we briefly discussed toolkit design. We then discussed the structure of the original toolkit SPARE PARTS, and showed how we revised it to arrive at its 2003 version, and then expanded it to get to the new toolkit, SPARE TIME.

- It was easy to revise and expand the original SPARE PARTS toolkit to SPARE TIME, as the existing toolkit was very reusable (see Chapter 5).
- The use of STL instead of proprietary foundation classes in SPARE TIME was relatively straightforward, as discussed in Section 5.4.1.
- Generalizing the Commentz-Walter pattern matcher and adding the new shift functions was relatively easy (Section 5.5), although the actual generalization of shift function precomputations to be compatible with automata other than a reverse trie has not yet been implemented.
- The pattern matching algorithms should be implemented (for those algorithms for which this has not already been done), benchmarked and compared, especially those algorithms (such as the bit-parallel Aho-Corasick in Subsection 3.9.3) for which there are many different and possibly interrelated aspects influencing performance.

Chapter 7

Future work

Although we have extended the original taxonomy and toolkit of keyword pattern matching algorithms by adding numerous algorithms and algorithm variants, there are still many keyword pattern matching algorithms that we have not considered. Possible future work on the taxonomy and toolkit includes:

1. Implementing the bit-parallel prefix-based algorithms such as Shift-And and a bit-parallel Aho-Corasick variant, derived in Section 3.9.

2. Deriving and implementing the (Multi-)BNDM algorithm ([NR00]), a bit-parallel factor-based pattern matching algorithm.

3. Deriving and implementing the Wu-Manber algorithm ([WM94]), a block (instead of single character) suffix-based pattern matching algorithm that is often efficient in practice for multiple keyword pattern matching (see [NR02, p. 74-76]).

4. Deriving and implementing Sunday's variant of Boyer-Moore-Horspool ([Sun90]) to see if it is indeed less efficient than Boyer-Moore-Horspool due to decreased locality of reference (as mentioned in [NR02, p. 26]).

5. Reconstructing the toolkit such that the suffix-based shift functions are precomputed correctly for factor- and factor oracle-based Commentz-Walter pattern matching and may thus be used for those as well.[1] From the literature it appears that for such algorithms the no-factor shift (NFS) is often larger, and that therefore such suffix-based shift functions are not used at all. It would still be interesting to see whether benchmark results do indeed confirm this, or whether the small gains in shifts that can occur in certain cases do outweigh precomputation times.

6. Looking into the possibility of a bit-parallel suffix-based pattern matching algorithm, i.e. a bit-parallel version of (suffix-based) Commentz-Walter, and comparing its performance to that of the (Multi-)BNDM algorithm.

[1] Note that it is conceivable that there exist suffix-based shift functions for which such a generalization is not possible, since they could use specific properties of a suffix automaton. This is related to how the shift function discusses in Subsection 3.7.2 could only be used efficiently in combination with a DAWG used as a factor automaton.

7. Deriving and implementing the algorithms discussed in [KST03]. This paper discusses tuning string matching for pattern sets of 1000-100000 patterns. The algorithms presented combine bit-parallel algorithms—such as Shift-And, a bit-parallel Set Horspool and MultiBNDM—with the Karp-Rabin filtering approach ([KR87]).

8. Improving the efficiency of multiple keyword suffix-based (Commentz-Walter) algorithms (and possibly of factor- and factor oracle-based algorithms as well) by moving the update of the set of matches out of the inner repetition ([Wat00]).

The current version of SPARE TIME has been verified to compile succesfully using the MICROSOFT VISUAL STUDIO .NET 2002 C++ compiler. Since we have made every effort to use only standard C++ language constructs and STL features, it should be relatively easy to get the toolkit working under other recent compilers with good standards support. We plan to create makefiles and test the toolkit under GCC 3.x ourselves in the near future (both on the Intel/AMD platform as well as on the PowerPC/Apple G4 platform[2]).

In addition to this, we plan to incorporate the toolkit into FIRE STATION, an application intended as a finite automata and regular expression "playground". This application is currently under development as part of the master's thesis of Michiel Frishert ([Fri]).

The algorithms that are in the SPARE TIME toolkit—and any new algorithms added to it—should be benchmarked using various sets of input data. This would enable comparison of the real-time efficiency of the algorithms discussed. The benchmarks results could also be compared with other benchmarking data (such as that reported in [NR02]) to see whether the results correspond with each other or not.

It would also be useful to develop a small configuration language (also know as *little language* or *Domain Specific Language* (DSL)) to support use of the toolkit. In such a language, a user who has little experience in or knowledge of pattern matching should be able to describe what information he has on the size and composition of his text and pattern set, as well as what—if any—restrictions there are on memory usage and preprocessing time. This description should then lead to a particular algorithm from the toolkit being selected. Such a DSL could also be developed for a larger field than just keyword pattern matching, including for example support for regular expression and/or approximate pattern matching.

Similar taxonomies to that discussed in this thesis could also be constructed for a number of related pattern matching problems. A toolkit could then be constructed for each of those taxonomies, or the algorithms could be combined in a single toolkit to take advantage of component reuse. The related fields include:

- Approximate pattern matching
- Regular expression pattern matching
- Multi-dimensional pattern matching
- Tree pattern matching
- Graph pattern matching

[2]Initial work indicates that it compiles and seems to function correctly under GCC on the PowerPC/Apple G4 platform, with only a relatively small number of changes.

- Pattern matching on compressed text

Finally, as discussed in Section 4.6, an automaton-independent characterization of the language recognized by a factor oracle is still an open question. In addition, it would be interesting to formalize the algorithm that was used to come up with the alternative Glushkov automaton with less transitions than the factor oracle, shown in Figure 4.4.

Appendix A

Algorithm and problem details

In this appendix we list the algorithm and problem details together with a short description.

P	Examine prefixes of a given string in any order.
P_+	Examine prefixes of a given string in order of increasing length.
S	Examine suffixes of a given string in any order.
S_+	Examine suffixes of a given string in order of increasing length.
GS=S	Use guard strengthening to increment the length of a suffix only for as long as a string which is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.
GS=F	Use guard strengthening to increment the length of a suffix only for as long as a string which is a factor of some keyword, preceded by a symbol is again a factor of some keyword.
GS=FO	Use guard strengthening to increment the length of a suffix only for as long as a string whose reverse is part of the language of the factor oracle on the reverse of the set of keywords, preceded by a symbol is again part of that language.
GS=SO	Use guard strengthening to increment the length of a suffix only for as long as a string whose reverse is part of the language of the suffix oracle on the reverse of the set of keywords, preceded by a symbol is again part of that language.
EGC=RSA	Usage of automaton recognizing the reverse of the set of suffixes of the keywords to check whether a string which is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.

RT — The original name (in [WZ96]) of what we refer to by the combination of details (GS=S) and (EGC=RSA).

EGC=RFA — Usage of an automaton recognizing the reverse of the set of factors of the keywords to check whether a string which is a factor of some keyword, preceded by a symbol is again a factor of some keyword.

EGC=RFO — Usage of a factor oracle on the reverse of the keywords, to check whether a string which is part of the language of the factor oracle, preceded by a symbol is again part of the language of that factor oracle.

LMIN — When using an automaton in one of the (EGC) details, construct this automaton on the prefixes of length equal to the length of the shortest keyword instead of on the complete keywords.

SSD — Consider any shift distance that does not lead to the missing of any matches. Such shift distances are called *safe*.

ONE — Use a safe shift distance of 1.

LSKP — Use a property of the DAWG to maintain a variable representing the longest suffix (of the recognized factor) that is a prefix of some keyword, and use this variable as the basis for the safe shift distance.

NLAU — No lookahead at the symbols of the unscanned part of the input string when computing a safe shift distance.

OPT — When computing a safe shift distance use the recognized suffix and only the immediately preceding (mismatching) symbol, strictly coupled.

NLA — When computing a safe shift distance do not look at the symbols preceding the recognized suffix.

BMCW — When computing a safe shift distance on the one hand use the recognized suffix and the fact that the symbol preceding it is mismatching, and on the other hand, but strictly independent, the identity of that symbol.

BMH — When computing a safe shift distance, use the first symbol compared against, whether it is matching or not.

NFS — When computing a safe shift distance, use the fact that the recognized factor preceded by the symbol preceding it is not a factor of any keyword.

OKW — (problem detail) The set of keywords contains only one keyword.

OBM — Introduce a particular algorithm skeleton as a starting point for the derivation of the different Boyer-Moore variants.

INDICES — Represent substrings by indices into the complete strings, converting a string-based algorithm into an indexing-based algorithm.

MO — 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 — The forward match order is used to compare the (single) keyword against a potential match in a left to right direction.
- REV — 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 — 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 — Before an attempt to match a candidate string and the keyword, a 'skip loop' is used to skip portions of the input string that cannot possibly lead to a match. Particular 'skip loops' are:

- NONE — No 'skip loop' is used.
- SFC — 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 — As with (SFC), but the last symbol of the candidate and the keyword are compared and possibly a larger shift distance (than with (SFC)) is used.
- SLFC — As with (FAST), but a low frequency symbol of the keyword is first compared.

MI — 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.

E — Matches are registered by their endpoints.

SP	Maintain the set of suffixes of the currently attempted match that are prefixes of a keyword, in order to easily compute new matches.
BPSP	Use bit-parallellism to maintain the set described in detail (SP).
INV	Inversion of the bits used in detail (BPSP), leading to the removal of a particular bitvector and less bit-operations, if detail (OKW) has been applied.
AC	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	A single 'optimized' transition function is used to update the state variable in the Aho-Corasick algorithm.
LS	Use linear search to update the state variable in the Aho-Corasick algorithm.
AC-FAIL	Implement the linear search using the transition function of the extended forward trie and the failure function.
KMP-FAIL	Implement the linear search using the extended failure function.
BPAC	Use bit-parallelism to encode the automaton used in detail (AC-OPT).

Appendix B

Object-oriented terminology

In this appendix we introduce the most important object-oriented terms used in Part III of this thesis. These terms include those used in [Wat95].

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.

Abstract: A class of which no instances can be created.

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.

Bibliography

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[ACR01] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient Experimental String Matching by Weak Factor Recognition. In *Proceedings of the 12th conference on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 51–72, 2001.

[Ale01] Andrei Alexandrescu. *Modern C++ Design.* Addison-Wesley, 2001.

[AR99] Cyril Allauzen and Mathieu Raffinot. Oracle des facteurs d'un ensemble de mots. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, June 1999.

[Aus99] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Templates Library.* Addison-Wesley, 1999.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20, 1977.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns.* John Wiley & Sons Ltd., 1996.

[Boo94] Grady Booch. *Object oriented analysis and design, with applications.* Benjamin/Cummings, 2nd edition, 1994.

[Bre00] Ulrich Breymann. *Designing Components with the C++ STL: A New Approach to Programming.* Addison-Wesley, 2000.

[Bud97] Timothy A. Budd. *An introduction to object-oriented programming.* Addison-Wesley, 2nd edition, 1997.

[BYG89] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proceedings of the 12th International Conference on Research and Development in Information Retrieval*, pages 168–175. ACM Press, 1989.

[CCG+94] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CE95] M.D. Carroll and M.A. Ellis. *Designing and coding reusable C++.* Addison-Wesley, 1995.

[CH97] Maxime Crochemore and Cristophe Hancart. Automata for Matching Patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2. Springer, 1997.

[Cop92] James O. Coplien. *Advanced C++: programming styles and idioms.* Addison-Wesley, 1992.

[Cop98] James O. Coplien. *Multi-Paradigm DESIGN for C++.* Addison-Wesley, 1998.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[CW79a] B. Commentz-Walter. A string matching algorithm fast on the average. In H.A. Maurer, editor, *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, pages 118–132. Springer Verlag, 1979.

[CW79b] B. Commentz-Walter. A string matching algorithm fast on the average. Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[CZW03a] Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. In *Proceedings of the Prague Stringology Conference 2003*, 2003.

[CZW03b] Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. *Constructing Factor Oracles.* Computing Science Report 03/?? Technische Universiteit Eindhoven, 2003.

[Dew02] Stephen C. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design.* Addison-Wesley, 2002.

[DF88] Edsger W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison Wesley, 1988.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[Fow96] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1996.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(10):490–499, 1960.

[Fri] Michiel Frishert. FIRE Station: a FInite automata & Regular Expression playground. MSc thesis, Technische Universiteit Eindhoven, to be published.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Hor80] R. Nigel Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[Jon83] H.B.M. Jonkers. Abstraction, specification and implementation techniques, with an application to garbage collection. *Mathematical Centre Tracts*, 166, 1983.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[Kal90] Anne Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall, 1990.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[KST03] Jari Kytöjoki, Leena Salmela, and Jorma Tarhio. Tuning String Matching for Huge Pattern Sets. In R. Baeza-Yates et al., editor, *Proceedings of the 14th conference on Combinatorial Pattern Matching Conference*, volume 2676 of *LNCS*, pages 211–224, 2003.

[Lip99] Stanley B. Lippman. *Essential C++.* Addison-Wesley, 1999.

[LL00] Arnaud Lefebvre and Thierry Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms*, pages 145–158, 2000.

[LL02] Arnaud Lefebvre and Thierry Lecroq. Compror: on-line losless data compression with a factor oracle. *Inf. Process. Lett.*, 83(1):1–6, 2002.

[LLA02] Arnaud Lefebvre, Thierry Lecroq, and J. Alexandre. Drastic improvements over repeats found with a factor oracle. In E. Billington, D. Donovan, and A. Khodkar, editors, *Proceedings of the 13th Australasian Workshop on Combinatorial Algorithms*, pages 253–265, 2002.

[Mar90] A.J.J.M. Marcelis. On the classification of attribute evaluation algorithms. *Science of Computer Programming*, 14:1–24, 1990.

[Mey96] Scott Meyers. *More Effective C++.* Addison-Wesley, 1996.

[Mey98a] Bertrand Meyer. *Object-Oriented Software Construction.* Addison-Wesley, 2nd edition, 1998.

[Mey98b] Scott Meyers. *Effective C++.* Addison-Wesley, 2nd edition, 1998.

[NR00] Gonzalo Navarro and Mathieu Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5(4), 2000. http://www.jea.acm.org.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[SE90] Bjarne Stroustrup and M. Ellis. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[vdE92] J.P.H.W. van den Eijnde. *Program derivation in acyclic graphs and related problems.* Computing Science Report 92/04. Technische Universiteit Eindhoven, 1992.

[VJ03] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley, 2003.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Technische Universiteit Eindhoven, 1995.

[Wat00] Bruce W. Watson. A new family of Commentz-Walter-style multiple-keyword pattern matching algorithms. In *Proceedings of the Prague Stringology Club Workshop 2000*, pages 71–76, 2000.

[WC03] Bruce W. Watson and Loek Cleophas. SPARE Parts: A C++ toolkit for String PAttern REcognition. *Software: Practice and Experience*, 2003. To be published.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[WM94] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[WZ92] B. W. Watson and G. Zwaan. *A taxonomy of keyword pattern matching algorithms.* Computing Science Report 92/27. Technische Universiteit Eindhoven, 1992.

[WZ96] B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, 27(2):85–118, September 1996.