A taxonomy of sublinear multiple keyword pattern matching algorithms

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
A taxonomy of sublinear multiple keyword pattern matching algorithms

ISSN 0926-4515

All rights reserved
editors: prof. dr. J.C.M. Baeten
        prof. dr. M. Rem

Computing Science Report 95/13
Eindhoven, April 1995
A taxonomy of sublinear multiple keyword pattern matching algorithms

B.W. Watson & G. Zwaan
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
email: watson@win.tue.nl or wsinswan@win.tue.nl

April 21, 1995

Abstract
This paper presents a taxonomy of sublinear keyword pattern matching algorithms related to the Boyer-Moore algorithm [BM77] and the Commentz-Walter algorithm [CW79a, CW79b]. The taxonomy includes, amongst others, the multiple keyword generalization of the single keyword Boyer-Moore algorithm and an algorithm by Fan and Su [FS93, FS94]. The corresponding precomputation algorithms are presented as well. The taxonomy is based on the idea of ordering algorithms according to their essential problem and algorithm details, and deriving all algorithms from a common starting point by successively adding these details in a correctness preserving way. This way of presentation not only provides a complete correctness argument of each algorithm, but also makes very clear what algorithms have in common (the details of their nearest common ancestor) and where they differ (the details added after their nearest common ancestor). Introduction of the notion of safe shift distances proves to be essential in the derivation and classification of the algorithms. Moreover, the paper provides a common derivation for and a uniform presentation of the precomputation algorithms, not yet found in the literature.
1 Introduction

The keyword (or string) pattern matching problem can informally be described as the problem
of finding all occurrences of keywords (strings) from a given set as substrings in a given (input)
string. This problem is encountered in many areas and in several forms. In computing science, for
instance, it plays a role in text search/analysis, lexical analysis, and data processing. In biology
it is encountered in the analysis of, amongst others, DNA sequences. The problem can also be
generalized to the matching of regular expressions, tree patterns, and graph patterns, none of
which is treated here.

The keyword pattern matching problem has been extensively studied and a multitude of di­
verse solutions/algorithms exists. Single keyword algorithms are for instance described by Knuth,
Morris, and Pratt [KMP77] and Boyer and Moore [BM77]; multiple keyword algorithms by Aho
and Corasick [AC75], by Commentz-Walter [CW79a, CW79b], and by Fan and Su [FS93, FS94].
An overview of keyword pattern matching algorithms can be found in [Aho90].

Due to the diversity of the algorithms and their descriptions—that tend to be rather involved
and verbal—it is hard to get a good overview and to make a sound comparison between algorithms.
In order to fulfill these needs a taxonomy of keyword pattern matching algorithms was presented
by Watson and Zwaan in [WZ92, WZ93]. Here, we focus our attention on a part of that taxonomy
containing a family of multiple keyword pattern matching algorithms that have a matching time
that may be sublinear in the length of the input string (taking the number of symbol comparisons
as a measure of matching time). Amongst others, it comprises the multiple keyword generalization
of the single keyword Boyer-Moore algorithm [BM77], the Commentz-Walter algorithm [CW79a,
CW79b], and the algorithm by Fan and Su [FS93, FS94] (only after deriving this algorithm we
found its description by Fan and Su). Both the Boyer-Moore and Commentz-Walter algorithms
provided the inspiration for our derivations and classifying principle.

The main results of this paper are comprised in the taxonomy graph shown in figure 1 (a more
detailed description of the graph is found further on in this section). This taxonomy graph can
be viewed as an alternative table of contents to this paper. It was obtained in order to meet the
following goals:

- the systematic and formal derivation of the algorithms from a common starting point through
  a series of refinements to either algorithm or problem

- to factor out common portions of (the derivations of) well-known algorithms in order to
  facilitate the understanding of these algorithms and their comparison

- the presentation of the algorithms in a common framework to permit an easier comprehension
  of and a better comparison between the algorithms

It is the first concise and systematic presentation of and comparison between the algorithms from
the family considered here. Another taxonomy of (single keyword) pattern matching algorithms
by Hume and Sunday [HS91] does not meet the goals set in this paper since there any derivations
and proofs of algorithms are missing.

Moreover, we show that all functions that need to be precomputed for the pattern matching
algorithms of this family can in a simple way be expressed in a small set of base functions. The
definitions of the base functions can all be written in forms satisfying one general pattern. From
this general pattern a precomputation algorithm scheme has been derived that can be instantiated
for each base function to yield a precomputation algorithm for that function (sometimes, the
resulting algorithm can be simplified and/or some trivial postprocessing has to be added to it).
Hence, the precomputation algorithms can also be derived and presented in a uniform way. This
includes formal derivations of the precomputation algorithms for the Boyer-Moore algorithm, the
Commentz-Walter algorithm and the algorithm presented by Fan and Su not yet found in the
literature.

The taxonomy given here differs from the corresponding part of the taxonomy in [WZ92, WZ93]
in that we here present the correct multiple keyword generalization of the Boyer-Moore algorithm
Figure 1: Taxonomy graph of keyword pattern matching algorithms from the Boyer-Moore family. Each vertex corresponds to an algorithm. Vertices are labelled with either an algorithm number or a subsection number preceded by § referring to an algorithm or a subsection in this paper. Some vertices are additionally labelled with literature references. Each edge corresponds to the addition of either a problem or algorithm detail (see appendix B for a complete list of details and their descriptions). The sequence of edge labels that are encountered when going from the top vertex to another vertex forms a characterization of the algorithm corresponding to that vertex. For instance, the algorithm of the vertex labelled §3.5 (the Commentz-Walter algorithm) is characterized by $(p_+s_+,rt,ssd,nlau,opt,bmcw,cw)$. The dashed edge leading to [KMP77, AC75] indicates the path to the rest of the taxonomy presented in [WZ92, WZ93] that contains the Knuth-Morris-Pratt and the Aho-Corasick algorithms and is omitted in this paper.
and a common ancestor of this algorithm and the Commentz-Walter algorithm. Because of this the derivations and the structure of the taxonomy have changed. Moreover, we present a completely new derivation of the precomputation algorithms.

An implementation as a C procedural library of almost all algorithms from [WZ92] (called "Eindhoven Pattern Kit") and an analysis of their performance is given in [Wat94] (the implementation is in a somewhat rudimentary form being meant for the benchmarking only). Only the Commentz-Walter algorithm and a common descendant of both the Boyer-Moore and Commentz-Walter algorithm are implemented and discussed there. The performance results for these algorithms conform with the qualitative predictions made here. An implementation of all algorithms from this paper and from [WZ92] as a C++ class library is called "SPARE Parts: A C++ toolkit for String Pattern REcognition" and will be available at URL ftp://ftp.win.tue.nl/pub/techreports/pi/pattm/spare/. This implementation is entirely based on the abstract algorithms described in this paper—in fact it is a systematic translation of them (this in contrast to for instance the implementations given in [HS91]).

1.1 Basic algorithm and derivation principles

The algorithms in this family traverse the input string in a direction that is opposite to the direction in which keyword symbols are matched to symbols in the input string. In this paper we choose to inspect suffixes of prefixes of the input string both in order of increasing length. This algorithm will be the starting point of all further derivations. Choosing such a basic algorithm we have the possibility to attain matching times that are sublinear in the length of the input string (i.e. not all symbols of the input string are inspected). It is achieved by taking steps through the input string of which the length is determined by a shift function based on the information of the last matching attempt and possibly on additional information. The example in figure 2 illustrates this principle. Notice that not all symbols of the input string are scanned, although it is possible that some symbols of the input string are scanned more than once. The most simple

```
....aaabacdabef....
  #===
  baab
  | shift 3
  ....aaabacdabef....
     #
     baab
     | shift 4
     ....aaabacdabef...
        #
        baab
```

Figure 2: Example of larger shift distances

shift function is the function that always yields 1. However, one can imagine larger shifts being possible. Ideally, such a shift would take us to the next occurrence of a match, but then calculating the value of the shift function is equivalent to the pattern matching problem itself. Therefore, we strive for shift functions that are easier to calculate and that do not exceed the ideal shift (called safe shift functions) i.e. we aim at approximations of the ideal shift from below. The ideal shift function is the minimum over a domain characterized by some predicate. We derive various approximations from below by systematically weakening this predicate and derived predicates, by applying rules for minimum and maximum over disjunctive or conjunctive domains, and by enlarging domains. Considerations that play a role in these derivations are, for instance, whether or not to look ahead at symbols of the unscanned part of the input string, what information to use
on the last scanned (non-matching) symbol, and the extent to which this information is coupled with the information on the recognized suffix. Thus, we obtain several shift functions that meet the aforementioned requirements leading to an equal number of algorithms amongst which are the well-known Boyer-Moore and Commentz-Walter algorithms. The most simple weakening of the predicate is the weakening to the predicate true yielding the shift function that is always equal to 1. The reason to derive ever smaller shift functions is that a smaller shift functions usually takes less precomputation time and less storage space (for instance, a one dimensional table instead of a two dimensional table).

The techniques we use to systematically derive shift functions from the ideal shift function enable us to clearly delineate the relations between the resulting shift functions and algorithms. Furthermore, they may be used in the derivation of yet other members of this family. This truly systematic approach, especially the predicate weakening technique, is not known from literature. Among other things we derive that the Commentz-Walter algorithm [CW79a, CW79b] is not the multiple keyword generalization of the Boyer-Moore algorithm [BM77]. In fact, we show that the algorithms are incomparable (meaning that the shift distance in one algorithm is not always at least the shift distance in the other) and that they have a faster common ancestor that combines the properties of both. As it is, this common ancestor is derived first and subsequently the Boyer-Moore algorithm and the Commentz-Walter algorithm are derived from it. Both algorithms also have a common descendant (the algorithm that was incorrectly identified as the Boyer-Moore algorithm in [WZ92, WZ93]). Furthermore, it is shown that the algorithm described by Fan and Su [FS93, FS94] is an even faster ancestor of the common ancestor of the Boyer-Moore and Commentz-Walter algorithms.

1.2 The taxonomy

The algorithms are derived from a common starting point by successively adding either algorithm or problem details (see appendix B for a complete list of details and their descriptions). The only problem detail considered here is the restriction to the one keyword case. Among the algorithm details considered are restriction of nondeterminacy, introduction of the reverse trie, introduction of a shift function, and choice of a particular shift function. Each addition of an algorithm detail gives a new algorithm satisfying the same specification as the original algorithm; thus, the correctness of the algorithms is preserved. The sequence of details introduced in the derivation of an algorithm can be used to identify its similarities and its differences with other algorithms. These ordered detail sequences are used to identify the algorithms and to give a taxonomy of the algorithms in this family. The taxonomy is depicted as a graph in figure 1. Our method of developing a taxonomy was inspired by the method described by Jonkers [Jon83]. There it is applied to develop a taxonomy of garbage collection algorithms. The method is also applied to attribute evaluation algorithms by Marcelis [Mar90]. Other examples of algorithm taxonomies are found in [Bro83, Dar78].

All algorithms are presented in a somewhat extended version of the guarded command language of Dijkstra [Dij76] in order to avoid the peculiarities of any particular programming language. The algorithms use string type variables in order to abstract from any implementation detail like, for instance, indexing. Derivation of the algorithms is done a calculational way following Dijkstra's style of program derivation.

1.3 Overview

In section 2 we give a formal definition of the pattern matching problem. From a trivial solution to this problem we derive, by addition of a number of algorithm details, an algorithm that is the starting point for the derivation of the algorithms in section 3. In section 3 we start by adding the algorithm detail that states that shifts larger than one may be possible. Addition of this program detail accounts for the possible sublinear matching time of all of the algorithms to be derived in this section. Subsequently, we derive by systematic approximation from below of the maximal safe shift distance the various algorithms of this family (in order of decreasing matching speed). The
2 The problem and some naive solutions

The keyword pattern matching problem is to find all occurrences of keywords from a set as sub­strings in an input string. Formally, given an alphabet \( V \) (a non-empty finite set of symbols), an input string \( S \in V^* \), and a finite non-empty pattern set \( P \subset V^* \), establish

\[
R : \quad O = \left( \bigcup I, v, r : lv = S : \{I\} \times (\{v\} \cap P) \times \{r\} \right).
\]

Throughout this paper we will adopt the convention that, unless stated otherwise, program vari­ables and bound variables with names from the beginning of the Latin alphabet (i.e. \( a, b, c \)) will range over \( V \), while variables with names from the end of the Latin alphabet (i.e. \( l, q, r, u, v, w \)) will range over \( V^* \). A trivial (but unrealistic) solution to the problem is

Algorithm 2.1

\[
O := ( \bigcup I, v, r : lv = S : \{I\} \times (\{v\} \cap P) \times \{r\})
\]

The sequence of details describing this algorithm is the empty sequence (sequences of details are introduced in subsection 1.2 and figure 1).

There are two basic directions in which to proceed while developing naive algorithms to solve this problem. Informally, a substring of \( S \) can be considered a “suffix of a prefix of \( S \)” or a “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 Boyer-Moore, Commentz-Walter, and Fan and Su algorithms treat substrings of input string \( S \).

Formally, we can consider “suffixes of prefixes of \( S \)” as follows:

\[
( \bigcup I, v, r : lv = S : \{I\} \times (\{v\} \cap P) \times \{r\})
\]

= 

\[
\{ \text{introduce } u : u = lv \}
\]

\[
( \bigcup I, v, r, u : ur = S \land lv = u : \{I\} \times (\{v\} \cap P) \times \{r\})
\]

= 

\[
\{ \text{\( I, v \) only occur in the latter range conjunct, so restrict their scope} \}
\]

\[
( \bigcup u, r : ur = S : ( \bigcup I, v : lv = u : \{I\} \times (\{v\} \cap P) \times \{r\}) )
\]

A simple non-deterministic algorithm is obtained by applying “examine prefixes of a given string in any order” (algorithm detail (p)) to input string \( S \). In the following algorithm we use a for-rof statement in order to express its non-determinism. Statement for \( x : P \rightarrow S \) rof amounts to executing statement list \( S \) once for each value of \( x \) that satisfies \( P \) initially, assuming only a finite number of such values exist. The order in which the values of \( x \) are chosen is arbitrary. It results in

Algorithm 2.2(p)

\[
O := \emptyset;
\]

for \( (u, r) : ur = S \rightarrow \)

\[
O := O \cup ( \bigcup I, v : lv = u : \{I\} \times (\{v\} \cap P) \times \{r\})
\]

rof{ \( R \) }
The update of $O$ (with another quantifier) in the inner repetitions of algorithm (P) 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 2.3(PS)

\[
O := \varnothing; \\
\text{for }(u, r) : ur = S \to \text{for }(l, v) : lv = u \to \\
O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\} \\
\text{rof} \\
\text{rof}\{ R \}
\]

Algorithm (PS) 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 and Commentz-Walter algorithms examine string $S$ from left to right and the patterns in $P$ from right to left we focus our attention on (the operators $|$, $\|$, and $\|$ are defined in definition A.1; relation $\leq_p$ is defined in definition A.3):

Algorithm 2.4(P+4S+)

\[
u, r := e, S; \quad O := \{e\} \times (\{e\} \cap P) \times \{S\}; \\
\{ \text{invariant: } O = (\cup x, y, z : xyz = S \land xy \leq_p u : \{x\} \times (\{y\} \cap P) \times \{z\}) \}
\]

\[
do r \neq e \to \\
u, r := u(r[1]), r[1]; \quad l, v := u, e; \quad O := O \cup \{u\} \times (\{e\} \cap P) \times \{r\}; \\
do l \neq e \to \\
l, v := l[1], (l[1])v; \quad O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}
\]

od\{ R \}

This algorithm has running time $O(|S|^2)$, assuming that intersection with $P$ is a $O(1)$ operation. We will now improve the running time of this algorithm. Consider the set of suffixes of keywords $\text{suff}(P)$ (the functions $\text{suff}$ and $\text{pref}$ are defined in definition A.2). A string $w$ is an element of $\text{suff}(P)$ if and only if it can be extended on the left to a pattern in $P$, i.e. $(\exists w' : w' \in V^* : w'w \in P)$. It follows that if $w \notin \text{suff}(P)$ any extension of $w$ on the left is not an element of $\text{suff}(P)$ either. Consequently, the inner repetition in algorithm 2.4 can terminate as soon as $(l[1])v \notin \text{suff}(P)$ holds, since then all suffixes of $u$ that are equal to or longer than $(l[1])v$ are not in $\text{suff}(P)$ and hence not in $P$. The inner repetition guard is therefore strengthened to

\[ l \neq e \text{ cand } (l[1])v \notin \text{suff}(P). \]

Observe that $v \in \text{suff}(P)$ is an now invariant of the inner repetition. This invariant is initially established by the assignment $v := e$ since $P \neq \varnothing$ and thus $e \in \text{suff}(P)$. Direct evaluation of $(l[1])v \notin \text{suff}(P)$ is expensive. Therefore, it is done using the transition function $\tau_P$ of the reverse trie [Fre60] corresponding to $P$ where $\tau_P : \text{suff}(P) \times V \to \text{suff}(P) \cup \{\perp\}$ is defined by

\[
\tau_P(w, a) = \begin{cases} 
aw & \text{if } aw \in \text{suff}(P) \\
\perp & \text{if } aw \notin \text{suff}(P)
\end{cases} (w \in \text{suff}(P), a \in V).
\]

(algorithm detail (rr)). Since we usually refer to the trie corresponding to $P$ we will write $\tau$ instead of $\tau_P$. Transition function $\tau$ can be computed beforehand. Its precomputation is discussed in subsection 4.1. The guard becomes $l \neq e \text{ cand } \tau(v, l[1]) \neq \perp$ yielding algorithm
This algorithm has \( O(|S| \cdot \text{MAX}_P : p \in P : |p|) \) running time. It will serve as a starting point for the derivation of all algorithms in the following section.

3 Sublinear pattern matching algorithms

In this section we derive sublinear pattern matching algorithms starting with algorithm 2.5 by exploring the possibility of safely (without missing matches) making shifts of more than one symbol, i.e. replacing assignment \( u, r := u(r[1]), r[1] \) by assignment \( u, r := u(r[k]), r[k] \) for some \( k \) satisfying

\[
1 \leq k \leq (\text{MIN}_n : 1 \leq n \wedge \text{suff}(u(r[n])) \cap P \neq \emptyset : n)
\]

(algorithm detail (ssd) (safe shift distance)). The upperbound is the distance to the next match, the maximal safe shift distance. A number \( k \) satisfying this condition is called a safe shift distance. Since computing the upperbound on \( k \) is essentially the same as the problem we are trying to solve we aim at easier to compute approximations from below of the upperbound. These are derived by systematically weakening the predicate \( \text{suff}(u(r[n])) \cap P \neq \emptyset \) in the range of the upperbound, weakening resulting predicates, applying rules for minimum and maximum over a disjunctive or conjunctive domain, and enlarging domains. Considerations that play a role in these derivations are, for instance, whether or not to look ahead at symbols of the unscanned part of the input string, whether to use or not to use information on the last scanned symbol (usually a non-matching symbol), and the extent to which this information is coupled with the information on the recognized suffix. Thus, several algorithms are obtained amongst which the generalized version of the Boyer-Moore algorithm [BM77], the Commentz-Walter algorithm [CW79a, CW79b], and the algorithm by Fan and Su [FS93, FS94]. We derive ever smaller shift functions since the smaller the shift function the less precomputation time and storage space for the functions constituting the shift function is usually needed. For instance, the algorithm by Fan and Su [FS93, FS94] is faster than the Commentz-Walter algorithm [CW79a, CW79b], but it needs a two dimensional table to store one of the functions constituting its shift function whereas the Commentz-Walter algorithm only needs one dimensional tables.

In the derivations we use part of the postcondition of the inner repetition in algorithm 2.5 \((u = lv \wedge v \in \text{suff}(P))\). Adding \( l, v := \epsilon, \epsilon \) to the initial assignments in algorithm 2.5 turns \( u = lv \wedge v \in \text{suff}(P) \) into an invariant of the outer repetition. Due to the dependence of the upperbound on \( l, v, \) and \( r \) we will aim at shift functions \( k \) that depend on \( l, v, \) and \( r \) and write \( k(l,v,r) \). Hence, we arrive at the following algorithm scheme for all algorithms to be derived in this section:
Algorithm 3.1 \((p, q, r, s, t, u, v, w, x, y, z)\)

\[ u, r := e, S; O := \{e\} \times \{e\} \times \{P\}; l, v := e, e; \]

{ invariant: \(O = \{(u, y, z : x y S \leq u : \{x\} \times \{y\} \times \{z\}\) \}

\[ \land u = lv \land v \in \text{suff}(P) \land (l = \varepsilon \text{ cor } (l|1)v \notin \text{suff}(P)) \] \}

\[ \text{do } r \neq e \rightarrow \]

\[ u, r := u(l, v, r), r|k(l, v, r); \]

\[ l, v := u, e; O := O \cup \{l\} \times \{e\} \times \{r\}; \]

\[ \text{do } l \neq \varepsilon \text{ cand } \tau(l, |1l) \neq 1 \rightarrow \]

\[ l, v := l|1, (l|1)v; \]

\[ O := O \cup \{l\} \times \{v\} \times \{r\} \]

\[ \text{od} \]

\[ \text{od}\{ R \} \]

Particular algorithms are obtained by substituting their shift functions for \(k(l, v, r)\). Such a substitution may not always yield an algorithm that exactly corresponds with its original description in the literature; sometimes an additional transformation of the resulting algorithm is needed (for instance, a phase shift of the repetition; see [WZ92] for a phase shifted version of the algorithm scheme). Conjunct \(l = \varepsilon \text{ cor } (l|1)v \notin \text{suff}(P)\) is added to the invariant in order to stress that provided \(l\) is nonempty symbol \(l|1\) is non-matching.

### 3.1 No lookahead at the unscanned part of the input string

In this subsection we derive an approximation from below of the upperbound on \(k\) that does not depend on \(r\) and that will be a starting point of most of our further derivations. In terms of algorithms this means that we refrain from looking ahead at the symbols of \(r\), the yet unscanned part of the input string (algorithm detail \(NLAW) \(\text{No Lookahead at Unscanned part of the input string}\)). This is in accordance with most of the algorithms we are aiming at. One symbol lookahead at the unscanned part of the input string is discussed in subsection 3.8. We derive

\[ (\text{MIN } n : 1 \leq n \land \text{suff}(u(r|n)) \cap P \neq \emptyset : n) \]

\[ = \{ \text{domain split, } r|n = r \text{ if } n \geq |r| \} \]

\[ \text{(MIN } n : 1 \leq n \leq |r| \land \text{suff}(u(r|n)) \cap P \neq \emptyset : n) \]

\[ \min (\text{MIN } n : |r| < n \land \text{suff}(ur) \cap P \neq \emptyset : n) \]

\[ \geq \{ 1 \leq n \leq |r| : r|n \in V^n, \text{ monotonicity of suff and } \cap \} \]

\[ \text{(MIN } n : 1 \leq n \leq |r| \land \text{suff}(uV^n) \cap P \neq \emptyset : n) \]

\[ \min (\text{MIN } n : |r| < n \land \text{suff}(ur) \cap P \neq \emptyset : n) \]

\[ = \{ r \in V^{|r|}, \text{ monotonicity of suff and } \cap, \text{suff}(ur) \cap P \neq \emptyset \Rightarrow \text{suff}(uV^{|r|}) \cap P \neq \emptyset \} \]

\[ \text{(MIN } n : 1 \leq n \leq |r| \land \text{suff}(uV^n) \cap P \neq \emptyset : n) \]

\[ \geq \{ \text{enlarging domain} \} \]

\[ \text{(MIN } n : 1 \leq n \land \text{suff}(uV^n) \cap P \neq \emptyset : n) \]

Since the last formula is to be the starting point of our further derivations we will from here on aim at shift functions \(k\) being dependent only on \(l\) and \(v\) (remember \(u = lv \land v \in \text{suff}(P)\)). We will write \(k(l, v)\) instead of \(k(l, v, r)\).

### 3.2 Restriction to one symbol lookahead

In all derivations in this subsection and the following subsections we assume

\[ u = lv \land v \in \text{suff}(P). \]

Restriction to one symbol lookahead (l|1, the last symbol of u scanned in the inner loop) leads to the algorithm by Fan and Su [FS93, FS94]. It is obtained by weakening the predicate in the approximation of the upper bound in subsection 3.1 in the following way:

\[
\text{suff}(uV^n) \cap P \neq \emptyset \\
= \{ u = lv \}
\]

\[
\text{suff}(lv^n) \cap P \neq \emptyset \\
\Rightarrow \{ l = (l|1)(l|1), l|1 \in V^*, \text{monotonicity of \text{suff and } \cap} \}
\]

\[
\text{suff}(V^*(l|1)uV^n) \cap P \neq \emptyset \\
= \{ \text{property A.4} \}
\]

\[
V^*(l|1)uV^n \cap V^*P \neq \emptyset \\
= \{ l = \varepsilon: \text{property A.5.i; } l \neq \varepsilon: \text{property A.5.ii} \}
\]

\[
V^*(l|1)uV^n \cap P \neq \emptyset \lor uV^n \cap V^*P \neq \emptyset
\]

Notice that we have obtained a weaker predicate solely by discarding any information on l|1. The only information on l that is still taken into account is l|1 being either empty or consisting of one symbol. In the latter case we say to have one symbol lookahead. Observe that the symbol is the last symbol of u scanned in the inner loop and that it is a non-matching symbol. After substituting the weaker predicate we obtain shift distance \( k_{opt}(l,v) \) where \( k_{opt} \in V^* \times \text{suff}(P) \to N \) is defined for \( x \in V^* \) and \( y \in \text{suff}(P) \) by

\[
k_{opt}(x,y) = (\text{MIN } n : n \geq 1 \land (V^*(x|1)yV^n) \cap P \neq \emptyset \lor yV^n \cap V^*P \neq \emptyset) : n).
\]

Function \( k_{opt} \) can be expressed as follows

\[
k_{opt}(x,y) = \begin{cases} 
  d_{opt}(x|1,y) \min d_{sp}(y) & x \neq \varepsilon \\
  d_{i}(y) \min d_{sp}(y) & x = \varepsilon
\end{cases}
\]

where \( d_{opt} \in V \times \text{suff}(P) \to N \) is defined by

\[
d_{opt}(a,y) = (\text{MIN } n : n \geq 1 \land V^*ayV^n \cap P \neq \emptyset) : n \) 
\]

\( a \in V, y \in \text{suff}(P), \)

\( d_{sp} \in \text{suff}(P) \to N \) is defined by

\[
d_{sp}(y) = (\text{MIN } n : n \geq 1 \land yV^n \cap V^*P \neq \emptyset) : n \) 
\]

\( y \in \text{suff}(P), \)

(function \( d_2 \) in [CW79a, CW79b]), and \( d_{i} \in \text{suff}(P) \to N \) is defined by

\[
d_{i}(y) = (\text{MIN } n : n \geq 1 \land V^*yV^n \cap P \neq \emptyset) : n \) 
\]

\( y \in \text{suff}(P), \)

(function \( d_1 \) in [CW79a, CW79b]). Functions \( d_{opt} \) and \( d_{i} \) account for occurrences of \( ay \) and \( y \), respectively, within some keyword (i.e. as infix of some keyword), whereas function \( d_{sp} \) accounts for occurrences of suffixes of \( y \) as proper prefixes of some keyword. Precomputation of \( d_{opt}, d_{sp}, \) and \( d_{i} \) is discussed in subsection 4.2.

Calculating the shift distance in this way is referred to as algorithm detail \((\text{OPT})\) and results in algorithm \((P,S,RT,SSD,NLAU,OPT)\). We arrived at this algorithm not knowing it had already been described by Fan and Su in [FS93, FS94]. From their informal description it undoubtedly follows that they describe the same algorithm though their formal treatment of the algorithm and, especially, the precomputation is rather involved. Finally, notice that to store function \( d_{opt} \) one needs a two dimensional table, whereas functions \( d_{i} \) and \( d_{sp} \) only need one dimensional tables. In the following subsections we derive shift functions smaller than \( k_{opt} \) that are expressed solely in functions needing one dimensional tables for storage.
3.3 Lookahead symbol is mismatching

We derive an approximation from below of $d_{opt}$ that yields an algorithm that is the common ancestor of the multiple keyword generalization of the Boyer-Moore algorithm [BM77J and the Commentz-Walter algorithm [CW79a, CW79b]. Essentially, the resulting shift function is not based on the identity of the lookahead symbol $l_1$ but only uses the fact that the lookahead symbol is mismatching, as is done in the Boyer-Moore shift function. In this way one might say that the recognized suffix and the (mismatching) lookahead symbol have to some extent been decoupled.

We start by weakening the predicate from $d_{opt}$. Assume $l \neq \epsilon$ and $(l_1)v \notin \text{suff}(P)$. We derive

\[ V^*(l_1)vV^n \cap P \neq \emptyset \]

\[ = \{ v \in V^{|v|}, \text{monotonicity of } \cap \} \]

\[ V^*(l_1)V^{|v|+n} \cap P \neq \emptyset \land V^*(l_1)vV^n \cap P \neq \emptyset \]

\[ \Rightarrow \{ (l_1)v \notin \text{suff}(P), \text{ so } l_1 \in \{ a | a \in V \land av \notin \text{suff}(P) \}, \text{definition } MS \} \]

\[ V^*(l_1)V^{|v|+n} \cap P \neq \emptyset \land V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset \]

where $MS \in \text{suff}(P) \rightarrow V$ is defined by

\[ MS(y) = \{ a | a \in V \land ay \in \text{suff}(P) \} \quad (y \in \text{suff}(P)). \]

The first conjunct will lead to a shift component based on the identity of the lookahead symbol that is identical to a component of the Commentz-Walter shift function. The second conjunct will lead to a shift component—based on the recognized suffix and the fact that the lookahead symbol is mismatching—that is identical to a component of the Boyer-Moore shift function. Replacing the range predicate of $d_{opt}$ by the last predicate of the preceding derivation we proceed

\[ (\text{MIN } n : n \geq 1 \land V^*(l_1)V^{|v|+n} \cap P \neq \emptyset \land V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset : n) \]

\[ \geq (\text{MIN with conjunctive range}) \]

\[ (\text{MIN } n : n \geq 1 \land V^*(l_1)V^{|v|+n} \cap P \neq \emptyset) \]

\[ \max (\text{MIN } n : n \geq 1 \land V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset : n) \]

\[ = (\text{change of bound variable: } m = |v| + n; \text{definition } d_{vi} \text{ (after derivation)}) \]

\[ (\text{MIN } m : m \geq |v| + 1 \land V^*(l_1)V^m \cap P \neq \emptyset : m - |v|) \max d_{vi}(v) \]

\[ \geq (\text{enlarging domain}) \]

\[ (\text{MIN } m : m \geq 1 \land V^*(l_1)V^m \cap P \neq \emptyset : m - |v|) \max d_{vi}(v) \]

\[ = \{ l \neq \epsilon, \text{definition of } char_{cw} \text{ (after derivation)} \} \]

\[ char_{cw}(l_1, |v|) \max d_{vi}(v) \]

where $d_{vi} \in \text{suff}(P) \rightarrow N$ is defined by

\[ d_{vi}(y) = (\text{MIN } n : n \geq 1 \land V^*(V \setminus MS(v))vV^n \cap P \neq \emptyset : n) \quad (y \in \text{suff}(P)) \]

and $char_{cw} \in V \times N \rightarrow N$ is defined by

\[ char_{cw}(a, z) = (\text{MIN } n : n \geq 1 \land V^*aV^n \cap P \neq \emptyset : n - z) \quad (a \in V, z \in N). \]

This results in shift distance $k_{mcw}(l, v)$ where $k_{mcw} \in V^* \times \text{suff}(P) \rightarrow N$ is defined by

\[ k_{mcw}(x, y) = \begin{cases} (char_{cw}(x_1, |y|) \max d_{vi}(y)) \min d_{sp}(y) & (x \in V^+, y \in \text{suff}(P)) \\ d_i(y) \min d_{sp}(y) & (x = \epsilon, y \in \text{suff}(P)). \end{cases} \]

Precomputation of $char_{cw}$ is discussed in subsection 4.3 and of $d_{vi}, d_{sp},$ and $d_i$ in subsection 4.2.

Notice that we have

\[ k_{opt}(x, y) \geq k_{mcw}(x, y) \quad (x \in V^*, y \in \text{suff}(P)). \]
Approximation from below of $k_{opt}$ by $k_{bmcw}$ is referred to as algorithm detail (BMCW). We chose this name to reflect that essential ideas from both the Boyer-Moore and Commentz-Walter algorithms are introduced. In the next two subsections these algorithms are derived from the algorithm presented in this subsection and characterized by detail sequence $\langle P, S, RT, SSD, NLAU, OPT, BMCW \rangle$.

### 3.4 The multiple keyword Boyer-Moore algorithm

We proceed by deriving the multiple keyword generalization of the Boyer-Moore algorithm [BM77] from the algorithm in subsection 3.3. It only differs from the algorithm there in the way the lookahead symbol is taken into account. Assuming $l \neq \varepsilon$ we derive

$$char_{ew}(l|1, |v|)$$

$$= \begin{cases} \text{definition } char_{ew} \\
(MIN n: n \geq 1 \land V^*(l|1)V^n \cap P \neq \emptyset : n - |v|) \\
\geq \{ V^*(l|1)V^n \cap P \neq \emptyset \Rightarrow V^*(l|1)V^n \cap V^*P \neq \emptyset \} \\
(MIN n: n \geq 1 \land V^*(l|1)V^n \cap V^*P \neq \emptyset : n - |v|) \\
= \{ P \neq \emptyset, V^*(l|1)V^n \cap V^*p \neq \emptyset \text{ for all } p \in P, \text{ nonempty domain} \} \\
(MIN n: n \geq 1 \land V^*(l|1)V^n \cap V^*P \neq \emptyset : n - |v|) \\
= \{ l \neq \varepsilon, \text{ definition of char}_{bm} \text{ (after derivation)} \} \\
char_{bm}(l|1) - |v| \end{cases}$$

where $char_{bm} \in V \rightarrow \mathbb{N}$ is defined by

$$char_{bm}(a) = (MIN n: n \geq 1 \land V^*aV^n \cap V^*P \neq \emptyset : n) \quad (a \in V).$$

It results in shift distance $k_{bm}(l, v)$ where $k_{bm} \in V^* \times \text{suff}(P) \rightarrow \mathbb{N}$ is defined by

$$k_{bm}(x, y) = \begin{cases} ((char_{bm}(x|1) - |y|) \max d_{se}(y)) \min d_{sp}(y) & (x \in V^*, y \in \text{suff}(P)) \\
d_{i}(y) \min d_{sp}(y) & (x = \varepsilon, y \in \text{suff}(P)). \end{cases}$$

Precomputation of $char_{bm}$ is discussed in subsection 4.3 and of $d_{se}$, $d_{sp}$, and $d_{i}$ is discussed in subsection 4.2. Approximating $k_{bmcw}$ from below by $k_{bm}$ is referred to as algorithm detail (BM). It results in the multiple keyword generalization of the regular Boyer-Moore algorithm [BM77]. The algorithm is characterized by detail sequence $\langle P, S, RT, SSD, NLAU, OPT, BMCW, BM \rangle$. The regular Boyer-Moore algorithm can be obtained by restricting $P$ to one keyword (problem detail (OKW) (one keyword)). Notice that we have

$$k_{bmcw}(x, y) \geq k_{bm}(x, y) \quad (x \in V^*, y \in \text{suff}(P)).$$

Inequality can only occur if the lookahead symbol does not occur in any keyword except as the last symbol.

The formula for the Boyer-Moore shift function given here differs from the ones given in [BM77] and [Aho90]. We will show that all formulas are equivalent. First we define

$$m_p = \begin{cases} 1 & \text{if } \varepsilon \in P \\
(MIN p: p \in P : |p|) & \text{if } \varepsilon \not\in P. \end{cases}$$

We now have for all $a \in V$

$$char_{bm}(a) \leq m_p.$$
Finally, we derive for \( x \in V^+ \) and \( y \in \text{suff}(P) \)
\[
k_{bm}(x, y) = \begin{cases} 
\text{definition } k_{bm} \\
((\text{char}_{bm}(x|1) - |y|) \max d_{vi}(y)) \min d_{sp}(y)
\end{cases}
\]
\[
= \begin{cases} 
\{ d_{sp}(y) \geq m_p - |y| \geq \text{char}_{bm}(x|1) - |y| \} \\
((\text{char}_{bm}(x|1) - |y|) \max d_{vi}(y)) \min((\text{char}_{bm}(x|1) - |y|) \max d_{sp}(y))
\end{cases}
\]
\[
= \begin{cases} 
\text{distributivity} \\
(\text{char}_{bm}(x|1) - |y|) \max(d_{vi}(y) \min d_{sp}(y)).
\end{cases}
\]
The last formula in the preceding derivation coincides with the ones in [BM77J and [Ah090J.

3.5 The Commentz-Walter algorithm

Instead of approximating \( \text{char}_{cw} \) in \( k_{bm,cw} \) from below by \( \text{char}_{bm} \), we now approximate \( d_{vi} \) in \( k_{bm,cw} \) from below by \( d_{i} \). This results in the Commentz-Walter algorithm [CW79a, CW79b]. We derive
\[
d_{vi}(v) = \begin{cases} 
\text{definition } d_{vi} \\
(\text{MIN } n : n \geq 1 \land V^*(V \setminus MS(v)) v V^n \cap P \neq \emptyset : n) \\
\geq (V^*(V \setminus MS(v)) v V^n \cap P \neq \emptyset \Rightarrow V^* v V^n \cap P \neq \emptyset) \\
(\text{MIN } n : n \geq 1 \land V^* v V^n \cap P \neq \emptyset : n) \\
= \text{definition } d_{i} \\
d_{i}(v).
\end{cases}
\]
This results in shift distance \( k_{cw}(l, v) \) where \( k_{cw} \in V^* \times \text{suff}(P) \rightarrow \mathbb{N} \) is defined by
\[
k_{cw}(x, y) = \begin{cases} 
(\text{char}_{cw}(x|1, |y|) \max d_{i}(y)) \min d_{sp}(y) & (x \in V^+, y \in \text{suff}(P)) \\
(\overline{d_{i}(y) \min d_{sp}(y)} & (x = x, y \in \text{suff}(P)).
\end{cases}
\]
Precomputation of \( \text{char}_{cw} \) is discussed in subsection 4.3 and of \( d_{i} \) and \( d_{sp} \) in subsection 4.2. Approximating \( k_{bm,cw} \) from below by \( k_{bw} \) is referred to as algorithm detail (cw). It results in the Commentz-Walter algorithm [CW79a, CW79b] that is characterized by detail sequence \( (P, S, T, R, S, SD, N, LAU, OPT, BMCW, CW) \). Notice that we have
\[
k_{bm,cw}(x, y) \geq k_{cw}(x, y) & (x \in V^*, y \in \text{suff}(P)).
\]
Such a comparison can not be made between \( k_{bm} \) and \( k_{cw} \) as the following example shows.

Example 3.1 Let \( V \in \{a, b, c, d\}, P = \{cababa\}, \) and \( x \in V^* \). Shift functions \( k_{bm} \) and \( k_{cw} \) are incomparable since
\[
\begin{align*}
k_{opt}(xd, a) &= +\infty \min 6 = 6 \\
k_{bm,cw}(xd, a) &= (+\infty \max 4) \min 6 = 6 \\
k_{bm}(xd, a) &= ((6 - 1) \max 4) \min 6 = 5 \\
k_{cw}(xd, a) &= (+\infty \max 2) \min 6 = 6
\end{align*}
\]
and
\[
\begin{align*}
k_{opt}(xa, a) &= +\infty \min 6 = 6 \\
k_{bm,cw}(xa, a) &= ((2 - 1) \max 4) \min 6 = 4 \\
k_{bm}(xa, a) &= ((2 - 1) \max 4) \min 6 = 4 \\
k_{cw}(xa, a) &= ((2 - 1) \max 2) \min 6 = 2.
\end{align*}
\]
It also follows that in some cases \( k_{bm,cw} \) is smaller than \( k_{opt} \) and that in some cases \( k_{bm} \) and \( k_{cw} \) are smaller than \( k_{bm,cw} \). ∎
It is not possible that both $k_{bmcw}(x,y) > k_{bm}(x,y)$ and $k_{bmcw}(x,y) > k_{cw}(x,y)$ hold for some $x \in V^+$ and $y \in \text{suff}(P)$ since the first inequality implies $\text{char}_{cw}(x|1,|y|) = +\infty$ and this in its turn implies $k_{bmcw}(x,y) = d_{sp}(y) = k_{cw}(x,y)$.

3.6 Complete decoupling of recognized suffix and lookahead symbol

The derivations in the previous subsections effect an ever stronger decoupling of the recognized suffix $v$ and the lookahead symbol $I_1$ in the subsequent shift functions. By approximating $d_v$ in $k_{bm}$ from below by $d$, or $\text{char}_{cw}$ in $k_{cw}$ by $\text{char}_{bm}$ (or both in $k_{bmcw}$) we obtain a complete decoupling. It results in shift distance $k_{dal}(I,v)$ where $k_{dal} \in V^* \times \text{suff}(P) \rightarrow N$ is defined by

$$k_{dal}(x,y) = \begin{cases} 
((\text{char}_{bm}(x|1) - |y|) \max d_i(y)) \min d_{sp}(y) & (x \in V^+, y \in \text{suff}(P)) \\
(d_i(y) \min d_{sp}(y)) & (x = e, y \in \text{suff}(P)).
\end{cases}$$

Precomputation of $\text{char}_{bm}$ is discussed in subsection 4.3 and of $d_i$ and $d_{sp}$ in subsection 4.2. The algorithm can be characterized by detail sequences $(P+S+,RT,SSD,NLAU,OPT,BMCW,BM,CW)$ and $(P+S+,RT,SSD,NLAU,OPT,BMCW,CW,BM)$.

3.7 Discarding the lookahead symbol

We weaken the predicate in the range of $k_{opt}$ by weakening its first disjunct to $V^*vV^+ \cap P \neq \emptyset$ due to $V^*(I|1) \subseteq V^*$ and the monotonicity of $\cap$. This weakening step is referred to as discarding the lookahead symbol $I_1$. The shift distance corresponding to this weakening is $k_{nla}(v)$ where $k_{nla} \in \text{suff}(P) \rightarrow N$ is defined by

$$k_{nla}(y) = d_i(y) \min d_{sp}(y) \quad (y \in \text{suff}(P)).$$

Notice that this shift function can also be viewed as an approximation from below of $k_{dal}$. Precomputation of $d_i$ and $d_{sp}$ is discussed in subsection 4.2. Approximating $k_{opt}$ from below by $k_{nla}$ is referred to as algorithm detail (NLA) (No LookAhead at mismatching symbol) and results in algorithm $(P+S+,RT,SSD,NLAU,OPT,NLA)$.

3.8 One symbol lookahead at the unscanned part of the input string

In this subsection we consider looking ahead at the first symbol of the unscanned part $r$ of the input string. The first symbol of $r$ will be taken into account independently of the other available information. In this way we obtain stronger variants of all of the shift functions derived thus far. Assuming $r \neq e$ we derive

$$(\text{MIN } n : 1 \leq n \wedge \text{suff}(u(r|n)) \cap P \neq \emptyset : n) = \{ \text{domain split, } 1 \leq n \leq |r|: r|n = (r|1)((r|1)|(n-1)), |r| < n: r|n = r \}$$

$$(\text{MIN } n : 1 \leq n \leq |r| \wedge \text{suff}(u(r|1)((r|1)|(n-1))) \cap P \neq \emptyset : n) \min (\text{MIN } n : |r| < n \wedge \text{suff}(ur) \cap P \neq \emptyset : n) \geq \{ r \neq e, r \in (r|1)V^{|r|-1}, \text{suff}(ur) \cap P \neq \emptyset \Rightarrow \text{suff}(u(r|1)V^{|r|-1}) \cap P \neq \emptyset \} \geq \{ u \in V^*, \text{monotonicity of suff and } \cap \}$$

$$(\text{MIN } n : 1 \leq n \leq |r| \wedge \text{suff}(u(r|1)V^{|r|-1}) \cap P \neq \emptyset : n) \min (\text{MIN } n : |r| < n \wedge \text{suff}(ur) \cap P \neq \emptyset : n) \geq \{ r \neq e, r \in (r|1)V^{|r|-1}, \text{suff}(ur) \cap P \neq \emptyset \Rightarrow \text{suff}(u(r|1)V^{|r|-1}) \cap P \neq \emptyset \} \geq \{ u \in V^*, \text{monotonicity of suff and } \cap \}$$

$$(\text{MIN } n : 1 \leq n \leq |r| \wedge \text{suff}(V^*(r|1)V^{|r|-1}) \cap P \neq \emptyset : n) \geq \{ u \in V^*, \text{monotonicity of suff and } \cap \}$$

$$(\text{MIN } n : 1 \leq n \leq |r| \wedge \text{suff}(V^*(r|1)V^{|r|-1}) \cap P \neq \emptyset : n) \geq \{ u \in V^*, \text{monotonicity of suff and } \cap \}$$

$$(\text{MIN } n : 1 \leq n \leq |r| \wedge \text{suff}(V^*(r|1)V^{|r|-1}) \cap P \neq \emptyset : n) \geq \{ u \in V^*, \text{monotonicity of suff and } \cap \}$$
(MIN \(m \geq 0 \land \text{suff}(V^* (r1) V^m) \cap P \neq \emptyset : m + 1\))

\[=\] \{ property A.4, \(P \neq \emptyset, V^* (r1) V^p \neq \emptyset \) for all \(p \in P\), nonempty domain \}

( MIN \(m : 0 \leq m \land V^* (r1) V^m \cap V^* P \neq \emptyset : m\) + 1

\[=\] \{ definition \(\text{char}_{ia}\) (after derivation) \}

\(\text{char}_{ia}(r1) + 1\)

where \(\text{char}_{ia} \in V \rightarrow N\) is defined by

\(\text{char}_{ia}(a) = (\text{MIN } n : 0 \leq n \land V^* a V^n \cap V^* P \neq \emptyset : n)\) \((a \in V)\).

Precomputation of \(\text{char}_{ia}\) is discussed in subsection 4.3.

Let \(M(u, r)\) denote the first expression in the preceding derivation as well as the first expression in the derivation in subsection 3.1, and let \(N(u)\) denote the last expression in the derivation in subsection 3.1. We then have

\(M(u, r)\)

\[=\] \{ property max \}

\(M(u, r) \ \text{max} \ \ M(u, r)\)

\[\geq\] \{ derivation in subsection 3.1, preceding derivation \}

\(N(u) \ \text{max}(\text{char}_{ia}(r1) + 1)\)

Since all shift functions derived in the previous subsections are approximations from below of \(N(u)\) the preceding derivation shows that they all may be extended with \(\text{max}(\text{char}_{ia}(r1) + 1)\) to form a class of stronger shift functions of signature \(k(l, v, r)\) (algorithm detail (OLAU) (one symbol lookahead at unscanned part of the input string)). The first derivation in this subsection shows that it is also possible to couple the information on \(r1\) with the information on \(I\) and \(v\) \((u = lv)\).

We will not pursue that direction any further in this paper.

## 4 Precomputation

In this section we derive algorithms for the precomputation of the functions used in the pattern matching algorithms in sections 2 and 3. The algorithms are correct due to their formal derivation. This can not always be said about the algorithms found in the literature, mostly due to the absence of any formal derivation (see for instance the single keyword Boyer-Moore precomputation algorithms given in [BM77], [KMP77], and [Ryt80], where each article shows the preceding article to give an incorrect precomputation algorithm). Moreover, we give the first formal derivation of the precomputation algorithms for the Boyer-Moore family of algorithms. They can, amongst others, be specialized to a correct precomputation algorithm for the single keyword Boyer-Moore algorithm. In fact, we show that the definition of all \(d\)-functions introduced in section 3 can be rewritten into a form in accordance with one general pattern. Subsequently, a general precomputation algorithm scheme for this general pattern is derived that can be instantiated for every \(d\)-function.

### 4.1 Precomputation of \(\tau_P\)

The transition function \(\tau_P \in \text{suff}(P) \times V \rightarrow (\text{suff}(P) \cup \{\bot\})\) of the reverse trie corresponding to \(P\) is defined by

\(\tau_P(u, a) = \begin{cases} au & \text{if } au \in \text{suff}(P) \\ \bot & \text{if } au \notin \text{suff}(P) \end{cases}\) \((u \in \text{suff}(P), a \in V)\).

Since \(\text{suff}\) is idempotent and the definition of \(\tau_P\) only depends on \(\text{suff}(P)\), we have \(\tau_P = \tau_{\text{suff}(P)}\). Set \(P\) being nonempty we have \(\text{suff}(P) = \{\varepsilon\} \cup \text{suff}(P)\) and \(\tau_{\text{suff}(P)} = \tau_{\{\varepsilon\} \cup \text{suff}(P)}\). These
observations lead to the following algorithm (cf. [AC75, section 3, algorithm 2]) to compute \( \tau_P \) in which variable \( \tau \) is used to calculate and store \( \tau_P \) thereby viewing \( \tau \) as a set of ordered pairs (the usual notion of a function) and abbreviate statements like \( \tau := \tau + \{((x, a), y)\} \) to \( \tau(x, a) := y \):

\[
\begin{array}{l}
\tau := \emptyset; \\
\{ \tau = \tau \} \\
\text{for } a : a \in V \rightarrow \tau(a) := \bot \text{ rof}; \\
\{ \tau = \tau(\varepsilon) \} \\
P_d, P_r := \emptyset, P; \\
\{ \text{invariant: } P_d \cup P_r = P \land P_d \cap P_r = \emptyset \land \tau = \tau(\varepsilon) \cup \text{suff}(P_d) \} \\
d \neq \emptyset \rightarrow \\
p : p \in P_r; \\
u, v := p, \varepsilon; \\
\{ \text{invariant: } uv = p \land \tau = \tau(\varepsilon) \cup \text{suff}(P_d) \cup \text{suff}(v) \} \\
do \neq \varepsilon \rightarrow \\
\text{if } \tau(v, u[1]) = \bot \rightarrow \tau(v, u[1]) := (u[1])v; \\
\text{for } a : a \in V \rightarrow \tau((u[1])v, a) := \bot \text{ rof} \\
\mid \tau(v, u[1]) \neq \bot \rightarrow \text{skip} \\
fi; \\
u, v := u[1], (u[1])v \\
\text{od}; \\
P_d, P_r := P_d + \{p\}, P_r - \{p\} \\
\text{od} \{ \tau = \tau_P \}
\end{array}
\]

In the algorithms we use \( + \) for the union of disjoint sets and \( - \) for the difference of a set and a subset of it. Notice that the algorithm does a depth first traversal of the reverse trie. Also notice that variable \( P_d \) is only needed to formulate an invariant for \( \tau \), so it may safely be removed from the algorithm. Furthermore, the states of the reverse trie are represented by strings. In practice, one can resort to a more suitable representation, for instance a representation by natural numbers. We will not elaborate this here.

### 4.2 Precomputation of \( d \)-functions

In this subsection we show that all \( d \)-functions introduced in section 3 can be written according to a general pattern. For this general pattern a general precomputation algorithm is derived. In order to obtain a precomputation algorithm for a particular \( d \)-function one only has to instantiate the general precomputation algorithm and possibly simplify the resulting algorithm.

The general pattern we strive for is a function \( d \in V \times \text{suff}(P) \rightarrow \mathbb{N} \) defined for \( a \in V \) and \( y \in \text{suff}(P) \) by

\[
d(a, y) = \left( \min t : t \in \text{suff}(P) \setminus \{ \varepsilon \} \land Q(a, t) \land R(a, y, t) \land y <_p t : |t| - |y| \right)
\]

where \( Q \) is a predicate on \( V \times V^* \) and \( R \) a predicate on \( V \times V^* \times V^* \). Why both \( Q \) and \( R \) are introduced will become clear when we derive an algorithm scheme for the computation of \( d \). We will now show that all \( d \)-functions introduced in section 3 can be expressed in this pattern. In the following derivations let \( a \in V \) and \( y \in \text{suff}(P) \).

- We derive

\[
d_{\text{opt}}(a, y) = \left( \min n : n \geq 1 \land V^* a y V^n \cap P \neq \emptyset : n \right)
\]

\[
= \left( \text{property A.4} \right)
\]
(\text{MIN } n : n \geq 1 \land aV^n \cap \text{suff}(P) \neq \emptyset : n) \]
\[
= \quad \{ \text{change of bound variable: } n = |s| \}
\]
(\text{MIN } s : s \in V^+ \land ays \in \text{suff}(P) : |s|) \]
\[
= \quad \{ \text{change of bound variable: } t = ys \}
\]
(\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land at \in \text{suff}(P) \land y <_P t : |t| - |y|) \]

Hence, we have expressed \(d_{sp}\) according to the general pattern with \(Q(a, t) = at \in \text{suff}(P)\) and \(R(a, y, t) = \text{true}\). Notice that \(at \in \text{suff}(P) \equiv \tau(t, a) \neq \perp\).

**Assuming \(y = \epsilon\) we derive**

\[
d_{sp}(\epsilon)
\]
\[
= \quad \{ \text{definition } d_{sp} \}
\]
(\text{MIN } n : n \geq 1 \land V^n \cap V^*P \neq \emptyset : n) \]
\[
= \quad \{ \text{calculus} \}
\]
\[
(\text{MIN } t : t \in P : (\text{MIN } n : n \geq 1 \land V^n \cap V^*t \neq \emptyset : n)) \]
\[
= \quad \{ \text{domain split: } P = P \setminus \{\epsilon\} \cup (P \cap \{\epsilon\}) \}
\]
(\text{MIN } t : t \in P \setminus \{\epsilon\} : |t|) \min (\text{MIN } t : t \in P \cap \{\epsilon\} : 1) \]
\[
= \quad \{ \text{rewriting in order to obtain general pattern} \}
\]
(\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land t \in P \land \epsilon <_P t : |t| - |\epsilon|) \min (\text{MIN } t : t \in P \cap \{\epsilon\} : 1) \]

**Assuming \(y \neq \epsilon\)** we derive

\[
d_{sp}(y)
\]
\[
= \quad \{ \text{definition } d_{sp} \}
\]
(\text{MIN } n : n \geq 1 \land yV^n \cap V^*P \neq \emptyset : n) \]
\[
= \quad \{ \text{property A.4} \}
\]
(\text{MIN } n : n \geq 1 \land \text{suff}(yV^n) \cap P \neq \emptyset : n) \]
\[
= \quad \{ y \neq \epsilon, \text{ hence } \text{suff}(yV^n) = yV^n \cup \text{suff}((y|1)V^n), \text{ set calculus} \}
\]
(\text{MIN } n : n \geq 1 \land yV^n \cap P \neq \emptyset : n) \]
\[
= \quad \{ \text{change of bound variable: } n = |s|, \text{ property A.4} \}
\]
(\text{MIN } s : s \in V^+ \land ys \in P : |s|) \min (\text{MIN } n : n \geq 1 \land (y|1)V^n \cap V^*P \neq \emptyset : n) \]
\[
= \quad \{ \text{change of bound variable: } t = ys, y \in \text{suff}(P), y|1 \in \text{suff}(P), \text{ definition } d_{sp} \}
\]
(\text{MIN } t : t \in P \land y <_P t : |t| - |y|) \min d_{sp}(y|1) \]
\[
= \quad \{ \text{rewriting in order to obtain general pattern} \}
\]
(\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land t \in P \land \epsilon <_P t : |t| - |\epsilon|) \min d_{sp}(y|1) \]

Although the derived definition of \(d_{sp}\) is recursive and \(d_{sp}\) does not have an argument \(a \in V\) one can still discern the general pattern with \(Q(a, t) = t \in P\) and \(R(a, y, t) = \text{true}\). Precomputation of \(d_{sp}\) can be done according to the general precomputation algorithm without an iteration over \(V\), followed by a breadth first traversal of the reverse trie.
• We derive

\[ d_i(y) \]
\[ = \{ \text{definition } d_i \} \]

\[ (\text{MIN } n : n \geq 1 \land V^*yV^n \cap P \neq \emptyset : n) \]
\[ = \{ \text{property A.4} \} \]

\[ (\text{MIN } n : n \geq 1 \land yV^n \cap \text{suff}(P) \neq \emptyset : n) \]
\[ = \{ \text{change of bound variable: } n = |s| \} \]

\[ (\text{MIN } s : s \in V^+ \land ys \in \text{suff}(P) : |s|) \]
\[ = \{ \text{change of bound variable: } t = ys \} \]

\[ (\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land y <_p t : |t| - |y|) \]

Although \( d_i \) does not have an argument \( a \in V \) its definition still matches the general pattern with \( Q(a, t) = \text{true} \) and \( R(a, y, t) = \text{true} \).

• We derive

\[ d_{vi}(y) \]
\[ = \{ \text{definition } d_{vi} \} \]

\[ (\text{MIN } n : n \geq 1 \land V^*(V \setminus MS(y))yV^n \cap P \neq \emptyset : n) \]
\[ = \{ \text{property A.4} \} \]

\[ (\text{MIN } n : n \geq 1 \land (V \setminus MS(y))yV^n \cap \text{suff}(P) \neq \emptyset : n) \]
\[ = \{ \text{change of bound variable: } n = |s| \} \]

\[ (\text{MIN } s : s \in V^+ \land (V \setminus MS(y))ys \cap \text{suff}(P) \neq \emptyset : |s|) \]
\[ = \{ \text{change of bound variable: } t = ys \} \]

\[ (\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land (V \setminus MS(y))t \cap \text{suff}(P) \neq \emptyset \land y <_p t : |t| - |y|) \]
\[ = \{ \text{definition } MS \} \]

\[ (\text{MIN } t : t \in \text{suff}(P) \setminus \{\epsilon\} \land MS(t) \cap (V \setminus MS(y)) \neq \emptyset \land y <_p t : |t| - |y|) \]

Apart from the fact that \( d_{vi} \) does not have an argument \( a \in V \) its definition matches the general pattern with \( Q(a, t) = \text{true} \) and \( R(a, y, t) = MS(t) \cap (V \setminus MS(y)) \neq \emptyset \).

Having expressed all \( d \)-functions from section 3 in the general pattern we proceed by giving a rather straightforward and nondeterministic algorithm to compute \( d \) which will serve as a starting point for further algorithm derivations (notice that program variable \( \text{dee} \) is used to compute and finally store function \( d \)).
for $y : y \in \text{suff}(P) \rightarrow$
  for $a : a \in V \rightarrow \text{dec}(a, y) := +\inf \text{rof}$

$\text{rof}$;

for $t : t \in \text{suff}(P) \setminus \{\varepsilon\} \rightarrow$
  for $a : a \in V \rightarrow$
    if $Q(a, t) \rightarrow$ for $y : y \in \text{suff}(P) \land y <_P t \rightarrow$
      if $R(a, y, t) \rightarrow \text{dec}(a, y) := \text{dec}(a, y) \min(|t| - |y|)$
        $\neg R(a, y, t) \rightarrow \text{skip}$
    $\text{rof}$
  $\text{fi}$

$\text{rof}$

$\text{rof}\{\text{dec} = d\}$.

For $d_1$ and $d_{sp}$ the loop iterating over $V$ can be omitted. For $d_{sp}$ the following additional breadth first traversal of the reverse trie ($\text{suff}(P)$) is needed to complete the computation:

$\text{n} := 1$;

$\text{do \: suff}(P) \cap V^n \neq \emptyset \rightarrow$
  for $t : t \in \text{suff}(P) \cap V^n \rightarrow \text{deep}(t) := \text{deep}(t) \min \text{deep}(t \cup 1)$ $\text{rof}$

$\text{od}\{\text{deep} = d_{sp}\}$.

First, we concentrate on making the innermost repetition deterministic. Define $s_p \in \text{suff}(P) \rightarrow \mathcal{P}(\text{suff}(P))$ by

$s_p(t) = \{ y \mid y \in \text{suff}(P) \land y <_P t \} \quad (t \in \text{suff}(P)),$

the set of all suffixes of keywords that are a proper prefix of $t$. Notice that for all $t \in \text{suff}(P) \setminus \{\varepsilon\}$ $s_p(t)$ is finite, nonempty, and linearly ordered with respect to $\leq_P$. Therefore, we can define $m_{sp} \in \text{suff}(P) \setminus \{\varepsilon\} \rightarrow \text{suff}(P)$ by

$m_{sp}(t) = (\text{MAX}_{\leq_P} y : y \in \text{suff}(P) \land y <_P t : y) \quad (t \in \text{suff}(P) \setminus \{\varepsilon\})$

being the maximal element of $s_p(t)$. In literature [AC75, BM77, CW79a, KMP77] function $m_{sp}$ is known as the failure function corresponding to the reverse trie. For $t \in \text{suff}(P) \setminus \{\varepsilon\}$ we derive a recursive definition of $s_p(t)$ in terms of function $m_{sp}$:

$s_p(t)$
  $=$  \{ definition $s_p$ \}
  \{ $y : y \in \text{suff}(P) \land y <_P t$ \}
  $=$  \{ $t \in \text{suff}(P) \setminus \{\varepsilon\}$, property A.6 \}
  \{ $y : y \in \text{suff}(P) \land (y = m_{sp}(t) \lor y <_P m_{sp}(t))$ \}
  $=$  \{ definition $s_p$ and $m_{sp}$ \}
  \{ $m_{sp}(t)$ \} $\cup$ $s_p(m_{sp}(t))$

Provided $m_{sp}$ is already computed (precomputation of $m_{sp}$ is discussed in subsection 4.4) the innermost repetition traversing the set $s_p(t)$ can be replaced by the following deterministic repetition (variable $v$ is a ghost variable needed to express the invariant):
\[ v := t; \{ \overline{v} := \emptyset; \} \]
\{ invariant: \( \forall v \in sp(t) \cup \{t\} \wedge sp(t) = \overline{v} \cup sp(v) \wedge \overline{v} \cap sp(v) = \emptyset \} \]
do \( \forall v \neq \epsilon \rightarrow \{ sp(v) = \{msp(v)\} \cup \epsilon \} \)
\[ v := msp(v); \{ \overline{v} := \overline{v} + \{v\}; \} \]
if \( R(a, v, t) \rightarrow dee(a, v) := \min(|t| - |v|) \)
| \( \neg R(a, v, t) \rightarrow \text{skip} \)
fi
enddo
\{ v = \epsilon, so \( sp(v) = \emptyset \) and \( sp(t) = \overline{v} \} \]

The invariant expresses that we have a bipartition of \( sp(t) \) in \( \overline{v} \) (elements of \( sp(t) \) that have already contributed to the computation of \( d \)) and \( sp(v) \) (the other elements of \( sp(t) \)).

In case \( R(a, y, t) = \text{true} \) for all \( a, y, \) and \( t \) the inner repetition can be made more efficient. Notice that this can be done for all presented \( d \)-functions except \( d_{vi} \). In the following assume that \( R(a, y, t) = \text{true} \) for all \( a, y, \) and \( t \). Suppose that for some \( v \in sp(t) \) in the above repetition we have \( |t| - |v| \geq dee(a, v) \). From the structure of the algorithms we infer that \( dee(a, v) = |t_0| - |v| \) for some \( t_0 \in \text{suff}(P) \setminus \{\epsilon\} \) with \( |t_0| \leq |t| \) that has already contributed to the computation of \( d \).

Therefore, for all \( s \in sp(v) \) we have \( dee(a, s) \leq |t_0| - |s| \) leading to
\[
\begin{align*}
\text{dee}(a, s) & \leq \{ \} \\
|t_0| - |s| & = \{ \text{dee}(a, v) = |t_0| - |v| \} \\
\text{dee}(a, v) + |v| - |s| & \leq \{ \text{dee}(a, v) \leq |t| - |v| \} \\
|t| - |s| & \end{align*}
\]

Hence, the contribution of \( t \) will not change the already computed value of \( \text{dee}(a, s) \) for \( s \in sp(v) \) and the inner repetition can be terminated. This yields the following repetition using an additional boolean variable \text{contributes} (notice that ghost variable \( \overline{v} \) is omitted):
\[ v := t; \text{contributes} := \text{true}; \]
do \( \forall v \neq \epsilon \wedge \text{contributes} \rightarrow \)
\[ v := msp(v); \]
if \( |t| - |v| < \text{dee}(a, v) \rightarrow \text{dee}(a, v) := |t| - |v| \)
| \( |t| - |v| \geq \text{dee}(a, v) \rightarrow \text{contributes} := \text{false} \)
fi
enddo.

Variable \text{contributes} can be removed resulting in the following repetition
\[ v := t; \]
do \( \forall v \neq \epsilon \rightarrow \)
\[ v := msp(v); \]
if \( |t| - |v| < \text{dee}(a, v) \rightarrow \text{dee}(a, v) := |t| - |v| \)
| \( |t| - |v| \geq \text{dee}(a, v) \rightarrow v := \epsilon \)
fi
enddo.

In order to further exploit this phenomenon the elements of \( \text{suff}(P) \setminus \{\epsilon\} \) are dealt with in order of increasing length, i.e. the outermost repetition of the general precomputation algorithm does a breadth first traversal of the reverse trie. This results in the following algorithm:
for \( y : y \in \text{suff}(P) \rightarrow \)
\[
\text{for } a : a \in V \rightarrow \text{dee}(a, y) := +\text{inf} \text{ rof}
\]
\[
\text{rof};
\]
\[
n := 1;
\]
\[
\text{do } \text{suff}(P) \cap V^n \neq \emptyset \rightarrow
\]
\[
\text{for } t \in \text{suff}(P) \cap V^n \rightarrow
\]
\[
\text{for } a : a \in V \rightarrow
\]
\[
\text{if } Q(a, t) \rightarrow v := t;
\]
\[
\text{do } v \neq \varepsilon \rightarrow
\]
\[
v := \text{msp}(v);
\]
\[
\text{if } |t| - |v| < \text{dee}(a, v) \rightarrow \text{dee}(a, v) := |t| - |v|
\]
\[
| |t| - |v| \geq \text{dee}(a, v) \rightarrow v := \varepsilon
\]
\[
\text{od}
\]
\[
| -Q(a, t) \rightarrow \text{skip}
\]
\[
\text{fi}
\]
\[
\text{rof};
\]
\[
n := n + 1
\]
\[
o d.
\]

In this optimized breadth first precomputation algorithm for each node \( v \) in the reverse trie and each symbol from \( V \) the step from \( v \) to \( \text{msp}(v) \) is done at most two times. Therefore, the precomputation time is \( O(|\text{suff}(P)| \cdot |V|) \). If the traversal of \( V \) can be omitted (as is the case for the precomputation of \( d_i \) and \( d_{sp} \)) it is \( O(|\text{suff}(P)|) \).

The breadth first precomputation algorithm for \( d_i \) can be simplified further by observing that since \( Q(a, t) = \text{true} \) the steps taken from \( t \) are always preceded by the steps taken from \( \text{msp}(t) \) (provided \( \text{msp}(t) \neq \varepsilon \)) since \( |\text{msp}(t)| < |t| \). So, only the contribution of \( t \) to \( d_i(\text{msp}(t)) \) has to be considered. This results in the following precomputation algorithm for \( d_i \) (program variable \( \text{deei} \) is used to compute and finally store \( d_i \)):

\[
\text{for } y : y \in \text{suff}(P) \rightarrow \text{deei}(y) := +\text{inf} \text{ rof};
\]
\[
n := 1;
\]
\[
\text{do } \text{suff}(P) \cap V^n \neq \emptyset \rightarrow
\]
\[
\text{for } t \in \text{suff}(P) \cap V^n \rightarrow
\]
\[
\text{deei}(\text{msp}(t)) := \text{deei}(\text{msp}(t)) \text{ min}(|t| - |\text{msp}(t)|)
\]
\[
\text{rof};
\]
\[
n := n + 1
\]
\[
o d.
\]

Notice that the breadth first traversal in this algorithm may be replaced by an arbitrary traversal of the reverse trie.

4.3 Precomputation of char\(_{cw} \), char\(_{bm} \), and char\(_{la} \)

Function char\(_{cw} \) can be expressed as
\[
\text{char}_{cw}(a, z) = \begin{cases} 
  +\text{inf} & \text{if } \text{char}_{cw}(a) = +\text{inf} \\
  \text{char}_{cw}(a) - z & \text{if } \text{char}_{cw}(a) \neq +\text{inf} 
\end{cases} \quad (a \in V, z \in \mathbb{N})
\]

where function \( \overline{\text{char}}_{cw} \in V \rightarrow \mathbb{N} \) is defined by
\[
\overline{\text{char}}_{cw}(a) = (\text{MIN} n : n \geq 1 \land V^*aV^n \cap P \neq \emptyset : n) \quad (a \in V).
\]

From the definition \( \overline{\text{char}}_{cw} \) it immediately follows that that its computation can be interwoven with the precomputation of the reverse trie \( \tau \).
Next, we derive for $a \in V$

\[
\text{char}_{\text{bm}}(a) = \begin{cases} 
\text{def} \text{inition} \text{ char}_{\text{bm}} \\
\end{cases}
\]

\[
\text{MIN} n : n \geq 1 \land V^*aV^n \cap V^*P \neq \emptyset : n
\]

\[
\text{property A.5.iii}
\]

\[
\begin{cases} 
\text{disjunctive range, definition char}_{\text{cw}} \\
\text{char}_{\text{cw}}(a) \text{ min } (\text{MIN} n : n \geq 1 \land aV^n \cap V^*P \neq \emptyset : n)
\end{cases}
\]

\[
\begin{cases} 
\text{domain split, char}_{\text{bm}}(a) \text{ min } (\text{MIN} n : n \geq 1 \land V^n \cap V^*P \neq \emptyset : n)
\end{cases}
\]

\[
\begin{cases} 
\text{domain split, definition } m_P \text{ (subsection 3.4)}
\end{cases}
\]

Finally, we derive for $a \in V$

\[
\text{char}_{\text{la}}(a) = \begin{cases} 
\text{def} \text{inition} \text{ char}_{\text{la}} \\
\end{cases}
\]

\[
\text{MIN} n : 0 \leq n \land V^*aV^n \cap V^*P \neq \emptyset : n
\]

\[
\text{domain split: } n \geq 1 \lor n = 0, \text{ definition char}_{\text{bm}}
\]

\[
\text{char}_{\text{bm}}(a) \text{ min } (\text{MIN} n : n = 0 \land V^*a \cap V^*P \neq \emptyset : 0)
\]

\[
\text{property A.5.ii}
\]

\[
\text{char}_{\text{bm}}(a) \text{ min } (\text{MIN} n : n = 0 \land (a \in \text{ suffix}(P) \lor \varepsilon \in P) : 0)
\]

We conclude that \(\text{char}_{\text{cw}}, \text{char}_{\text{bm}},\) and \(\text{char}_{\text{la}}\) can be computed from \(\text{char}_{\text{cw}}\).

**4.4 Precomputation of \(msp\)**

We conclude this section with the derivation of an algorithm computing \(msp\). We start by deriving

- for $a \in \text{suffix}(P) \cap V$

\[
msp(a) = \begin{cases} 
\text{def} \text{inition } msp \\
\end{cases}
\]

\[
\begin{cases} 
\text{MAX}_{\leq p} y : y \in \text{suffix}(P) \land y <_p a : y
\end{cases}
\]

\[
\begin{cases} 
\text{domain split, } a \equiv y = \varepsilon, \varepsilon \in \text{suffix}(P)
\end{cases}
\]

- and for $ax \in \text{suffix}(P), a \in V, \text{ and } x \in \text{suffix}(P) \setminus \{\varepsilon\}$

\[
msp(ax) = \begin{cases} 
\text{def} \text{inition } msp \\
\end{cases}
\]

\[
\begin{cases} 
\text{MAX}_{\leq p} y : y \in \text{suffix}(P) \land y <_p ax : y
\end{cases}
\]

\[
\begin{cases} 
\text{domain split, } \varepsilon <_p ax, \varepsilon \in \text{suffix}(P)
\end{cases}
\]
22 5 Conclusions

\[ (\text{MAX}_{\leq_p} y : y \in \text{suff}(P) \land y \leq_p ax \land y \neq \varepsilon : y) \max_{\leq_p} \varepsilon \]

\[ = \quad \{ \text{change of bound variable: } y = ay' \} \]

\[ (\text{MAX}_{\leq_p} y' : ay' \in \text{suff}(P) \land y' <_p x : ay') \max_{\leq_p} \varepsilon \]

\[ = \quad \{ ay' \in \text{suff}(P) \Rightarrow y' \in \text{suff}(P), x \in \text{suff}(P) \setminus \{\varepsilon\}, \text{definition } sp \} \]

\[ (\text{MAX}_{\leq_p} y' : ay' \in \text{suff}(P) \land y' \in sp(x) : ay') \max_{\leq_p} \varepsilon \]

From this it follows that \( msp(ax) \) can be computed by a linear search in downward order over \( sp(x) \) (remember that \( sp(x) \) is linearly ordered with respect to \( \leq_p \)) starting with \( msp(x) \). Provided the computation of \( msp \) is done using a breadth first traversal of the reverse trie (\( \text{suff}(P) \)) the value of \( msp \) is already computed for all all elements of \( sp(x) \cup \{x\} \) and can therefore be used to implement the linear search over \( sp(x) \). This results in the following algorithm (variable \( emsp \) is used to compute and finally store \( msp \)):

\[
\text{for } a : a \in \text{suff}(P) \cap V \rightarrow emsp(a) := e \text{ ref;}
\]

\[
n := 1;
\]

\[
\{ \text{invariant: } 1 \leq n \leq (\text{MAX} p : p \in P : |p|) + 1
\]

\[
\land (\forall y : y \in \text{suff}(P) \setminus \{\varepsilon\} \land |y| \leq n : emsp(y) = msp(y)) \}
\]

\[
\text{do } \text{suff}(P) \cap V^n \neq \emptyset \rightarrow
\]

\[
\text{for } t : t \in \text{suff}(P) \cap V^n \rightarrow
\]

\[
\text{for } a : a \in V \rightarrow
\]

\[
\text{if } at \in \text{suff}(P) \rightarrow v := emsp(t);
\]

\[
\{ \text{linear search } \}
\]

\[
\text{do } av \notin \text{suff}(P) \land v \neq \varepsilon \rightarrow v := emsp(v) \text{ od;}
\]

\[
\text{if } av \in \text{suff}(P) \rightarrow emsp(at) := av
\]

\[
| \text{ av } \notin \text{suff}(P) \land v = \varepsilon \rightarrow emsp(at) := \varepsilon
\]

\[
\}
\]

\[
\text{if } at \notin \text{suff}(P) \rightarrow \text{skip}
\]

\[
\text{fi}
\]

\[
\text{rof}
\]

\[
n := n + 1
\]

\]

\text{od.}

This breadth first algorithm computing \( msp \) can be combined with the breadth first algorithm computing the \( d \)-functions. The precomputation time now is \( O(|\text{suff}(P)|^2 \cdot |V|) \). This can be reduced to \( O(|\text{suff}(P)| \cdot |V|) \) at the expense of \( O(|\text{suff}(P)| \cdot |V|) \) additional storage space by also computing and storing the transition function \( \gamma_r \) of the reverse trie where \( \gamma_r \in V \times \text{suff}(P) \rightarrow \text{suff}(P) \) is defined by

\[
\gamma_r(a, x) = (\text{MAX}_{\leq_p} y : y \in \text{suff}(P) \land y \leq_p ax : y) \quad (a \in V, x \in \text{suff}(P)).
\]

The details of this approach can be found in [WZ92].

5 Conclusions

In this paper we derived and presented a taxonomy of sublinear keyword pattern matching algorithms closely related to the Boyer-Moore algorithm [BM77] and the Commentz-Walter algorithm [CW79a, CW79b]. It includes, amongst others, the multiple keyword generalization of the single keyword Boyer-Moore algorithm and the algorithm presented by Fan and Su [FS93, FS94]. We presented the algorithms within a common framework permitting an easier comprehension of and a better comparison between the algorithms. This was achieved by the systematic and formal derivation of the algorithms from a common starting point and by factoring out of common portions of the derivations. The derivations were done through series of refinements to either
A Definitions and properties

This section provides a series of definitions and properties which are used throughout this paper. In the following let $V$ be an alphabet.

For a string $w \in V^*$, $w^R$ denotes the reversal of $w$. For any language $L \subseteq V^*$ we define $L^R = \{w^R \mid w \in L\}$ (the reversal of language $L$).

23

algorithm or problem. A refinement to the algorithm/problem is referred to as the introduction of an algorithm/problem detail. The sequence of details that are subsequently introduced in a derivation characterizes the algorithm obtained by that derivation. Detail sequences can therefore be used to classify the algorithms in the taxonomy. Algorithms can now be compared by looking at their detail sequences. The taxonomy graph in figure 1 constitutes a concise presentation and classification of the pattern matching algorithms discussed, vertices representing algorithms and edges representing the addition of an algorithm or problem detail. It can be viewed as an alternative table of contents to this paper. Our results show how fruitful the applied method of developing a taxonomy is (it was inspired by the method described by Jonkers [Jon83]).

Introduction of the notion of safe shift distances proved to be essential for the derivation of the various algorithms. All algorithms are characterized by a—systematically derived and more or less easy to compute—approximation from below of the maximal safe shift distance, computation of the latter being equivalent to the keyword pattern matching problem itself. The systematic derivation provided a means to compare the algorithms and their matching speeds, and to get a better understanding of the algorithms and their interrelations. Perhaps this better understanding will help further the use of the algorithms from this family. Our derivations show the Commentz-Walter algorithm not to be the multiple keyword generalization of the Boyer-Moore algorithm (as was the original intention of Commentz-Walter) and that such a generalization can indeed be obtained. Of the algorithms presented the algorithm by Fan and Su [FS93, FS94] is the fastest (at the expense of additional precomputation time and additional storage requirements), followed by the common ancestor of the Boyer-Moore and Commentz-Walter algorithms, and then by both the multiple keyword generalization of the Boyer-Moore algorithm [BM77] and the Commentz-Walter algorithm [CW79a, CW79b]. The latter two are incomparable in matching speed. It is clear that we have not derived and presented all algorithms of the Boyer-Moore family. Our derivation method, however, clearly indicates how yet other members of this family of algorithms may be derived.

Apart from giving a taxonomy of pattern matching algorithms from the Boyer-Moore family we presented the first formally derived and therefore correct precomputation algorithms (this can not always be said about the algorithms found in the literature, mostly due to the absence of any formal derivation; see for instance the many solutions for the Boyer-Moore precomputation that have been published, corrected and republished). In fact, we showed that most of the precomputation algorithms can be obtained as instantiations of a general precomputation algorithm scheme derived for a general function pattern in which most components of the various shift functions can be expressed. Thus, we provided a common framework for the precomputation algorithms as well.

6 Acknowledgements

We thank Kees Hemerik and Frans Kruseman Aretz for their careful reading of earlier drafts of this paper and their constructive criticisms. We also thank Anne Kaldewaij and Lex Bijlsma for reading the final version.

A Definitions and properties

This section provides a series of definitions and properties which are used throughout this paper. In the following let $V$ be an alphabet.

For a string $w \in V^*$, $w^R$ denotes the reversal of $w$. For any language $L \subseteq V^*$ we define $L^R = \{w^R \mid w \in L\}$ (the reversal of language $L$).
Definition A.1. The infix operators $\lceil, \rceil, \lbrack, \rbrack : V^* \times N \rightarrow V^*$ are defined by

$\lceil 0 \rceil = \epsilon$ (if $v \in V^*$)

$\lceil k + 1 \rceil = \epsilon$ (if $k \geq 0$)

$(aw) \lceil k + 1 \rceil = a(w \lceil k \rceil)$ (if $k \geq 0, a \in V, w \in V^*$)

$v \lceil 0 \rceil = v$ (if $v \in V^*$)

$\lceil k + 1 \rceil = \epsilon$ (if $k \geq 0$)

$(aw) \lceil k + 1 \rceil = w \lceil k \rceil$ (if $k \geq 0, a \in V, w \in V^*$)

$\lceil k \rceil = (v \lceil R \rceil)\lceil k \rceil$ and $\lceil k \rceil = (v \lceil L \rceil)\lceil k \rceil$. The operators $\lceil, \rceil, \lbrack, \rbrack$ are called "left take," "left drop," "right take," and "right drop" respectively.

Define $\lceil \rceil$ as $v \lceil k \rceil = (v \lceil R \rceil)\lceil k \rceil$ and $\lceil \lbrack k \rceil$ as $v \lceil k \rceil = (v \lceil L \rceil)\lceil k \rceil$. The operators $\lceil, \rceil, \lbrack, \rbrack$ are called "left take," "left drop," "right take," and "right drop" respectively.

Definition A.2. Functions $\text{pref} : \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ and $\text{suff} : \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ are defined by

$\text{pref}(L) = \{ w | w \in V^* \land (\exists x : x \in V^* : wx \in L) \}$

and

$\text{suff}(L) = (\text{pref}(LR))R = \{ w | w \in V^* \land (\exists x : x \in V^* : xw \in L) \}$.

For $w \in V^*$ we will write $\text{pref}(w)$ (suff($w$)) instead of $\text{pref}\{(w)\}$ (suff($\{w\}$)).

Definition A.3. The relations $\leq_p$ and $\leq_s$ over $V^* \times V^*$ are defined by $u \leq_p v \equiv u \in \text{pref}(v)$ and $u \leq_s v \equiv u \in \text{suff}(v)$.

The following two properties are used in the derivation of the Commentz-Walter precomputation algorithm.

Property A.4. Let $A, B \subseteq V^*$. Then $\text{pref}(A) \cap B \neq \emptyset \equiv A \cap BV^* \neq \emptyset$ and $\text{suff}(A) \cap B \neq \emptyset \equiv A \cap V*B \neq \emptyset$.

Property A.5. Let $A, B \subseteq V^*$ and $a \in V$. Then

i. $V^*A \cap V^*B \neq \emptyset \equiv V^*A \cap B \neq \emptyset \lor A \cap V^*B \neq \emptyset$

ii. $V^*aA \cap V^*B \neq \emptyset \equiv V^*aA \cap B \neq \emptyset \lor A \cap V^*B \neq \emptyset$

iii. $V^*A \cap V^*B \neq \emptyset \equiv V^*A \cap B \neq \emptyset \lor A \cap V^*B \neq \emptyset$

Property A.6. For $x, y \in \text{suff}(P)$ and $y \neq \epsilon$ we have

$x <_p y \equiv x \leq_p \text{msp}(y)$.

Proof. Let $x, y \in \text{suff}(P)$ and $y \neq \epsilon$. We derive

$x <_p y$

$\Rightarrow \{ \text{definition of } <_p \text{ and } \text{pref}, x \in \text{suff}(P) \}$

$x \in \text{pref}(y) \setminus \{ y \} \cap \text{suff}(P)$

$\Rightarrow \{ \text{pref}(y) \setminus \{ y \} \cap \text{suff}(P) \text{ is nonempty } (y \neq \epsilon), \text{finite and linearly ordered w.r.t. } \leq_p \}$

$x \leq_p (\text{MAX}_{<_p} w : w \in \text{pref}(y) \setminus \{ y \} \cap \text{suff}(P) : w)$

$\Rightarrow \{ y \neq \epsilon, \text{definition of msp} \}$

$x \leq_p \text{msp}(y)$

$\Rightarrow \{ y \neq \epsilon, \text{msp}(y) <_p y \text{ (by definition of msp), transitivity of } <_p \}$

$x <_p y$

□
Algorithm and problem details

In this appendix we list the algorithm and problem details introduced in this paper with a short description.

P Examine prefixes of a given string in any order.

S Examine suffixes of a given string in any order.

+ Examine the strings from a given set in order of increasing length (this program detail can only be applied after, for instance, program details P and S).

RT Usage of the transition function of the reverse trie corresponding to the set of keywords to check whether a string, that is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.

SSD Allow any shift distance at least one that is safe, i.e. that does not cause the omission of any matches.

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

OLAU One symbol lookahead at 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.

BM Lessen the contribution of the symbol preceding the recognized suffix to the shift distance in case it does not occur in any keyword.

CW When computing a safe shift distance do not use the fact that the symbol preceding the recognized suffix is mismatching (use the recognized suffix and the symbol preceding it independently).

OKW The set of keywords contains only one keyword (in contrast to the preceding program details this is a problem detail).

References


<table>
<thead>
<tr>
<th>Computing Science Reports</th>
<th>Department of Mathematics and Computing Science</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this series appeared:</td>
<td>Eindhoven University of Technology</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>93/01 R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of program-</td>
</tr>
<tr>
<td></td>
<td>ming methods, p. 36.</td>
</tr>
<tr>
<td>93/02 T. Verhoeff</td>
<td>A continuous version of the Prisoner's Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03 T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/05 J.C.M. Baeten, C. Verhoef</td>
<td></td>
</tr>
<tr>
<td>93/06 J.P. Veltkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
</tr>
<tr>
<td>93/07 P.D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
</tr>
<tr>
<td>93/08 J. Verhoostel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/09 K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td>93/10 K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td></td>
<td>Part II: Frameworks, p. 44.</td>
</tr>
<tr>
<td>93/11 K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td>93/12 K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td></td>
<td>Part IV: Analysis Methods, p. 63.</td>
</tr>
<tr>
<td>93/13 K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td>93/14 J.C.M. Baeten, J.A. Bergstra</td>
<td>On Sequential Composition, Action Prefixes and</td>
</tr>
<tr>
<td></td>
<td>Process Prefix, p. 21.</td>
</tr>
<tr>
<td>93/16 H. Schepers, J. Hoornan</td>
<td>A Trace-Based Compositional Proof Theory for</td>
</tr>
<tr>
<td></td>
<td>Fault Tolerant Distributed Systems, p. 27.</td>
</tr>
<tr>
<td>93/17 D. Alstein, P. van der Stok</td>
<td>Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.</td>
</tr>
<tr>
<td>93/18 C. Verhoef</td>
<td>A congruence theorem for structured operational</td>
</tr>
<tr>
<td></td>
<td>semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/19 G.J. Houben</td>
<td>The Design of an Online Help Facility for ExSpect, p. 21.</td>
</tr>
<tr>
<td>93/21 M. Codish, D. Dams, G. Filé, M. Bruynooghe</td>
<td>Freeness Analysis for Logic Programs - And Correctness, p. 24</td>
</tr>
<tr>
<td>93/22 E. Poll</td>
<td>A Typechecker for Bijective Pure Type Systems, p. 28.</td>
</tr>
<tr>
<td>93/23 E. de Kogel</td>
<td>Relational Algebra and Equational Proofs, p. 23.</td>
</tr>
<tr>
<td>93/24 E. Poll and Paula Severi</td>
<td>Pure Type Systems with Definitions, p. 38.</td>
</tr>
<tr>
<td>93/26 W.M.P. van der Aalst</td>
<td>Multi-dimensional Petri nets, p. 25.</td>
</tr>
<tr>
<td>93/27 T. Klos and D. Kratsch</td>
<td>Finding all minimal separators of a graph, p. 11.</td>
</tr>
<tr>
<td>93/28 F. Kamareddine and R. Nederpels</td>
<td>A Semantics for a fine (\lambda)-calculus with de Bruijn indices, p. 49.</td>
</tr>
<tr>
<td>93/29 R. Post and P. De Bra</td>
<td>GOLD, a Graph Oriented Language for Databases, p. 42.</td>
</tr>
<tr>
<td>93/30 J. Deogun, T. Klos, D. Kratsch, H. Müller</td>
<td>On Vertex Ranking for Permutation and Other Graphs, p. 11.</td>
</tr>
</tbody>
</table>
Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.


IIAS, a sequential language for parallel matrix computations, p. 20.

Real Time Process Algebra with Infinitesimals, p. 39.

Abstract Reduction and Topology, p. 17.

Non Interleaving Process Algebra, p. 17.

Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.

A general conservative extension theorem in process algebra, p. 17.

Job Shop Scheduling by Constraint Satisfaction, p. 22.


Temporal operators viewed as predicate transformers, p. 11.

Automatic Verification of Regular Protocols in P/T Nets, p. 23.

A taxonomy of finite automata construction algorithms, p. 87.

A taxonomy of finite automata minimization algorithms, p. 23.

A precise clock synchronization protocol, p. 11.


Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

The object-oriented paradigm, p. 28.

Canonical typing and II-conversion, p. 51.


Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.


Time and the Order of Abstract Events in Distributed Computations, p. 29.


A Hierarchical Diagrammatic Representation of Class Structure, p. 22.

Process Algebra with Partial Choice, p. 16.
94/10 T. Verhoeff
94/11 J. Peleska
C. Huizing
C. Petersohn
94/12 T. Kloks
D. Kratsch
H. Müller
94/13 R. Seljé
94/14 W. Peremans
94/15 R.J.M. Vaessens
E.H.L. Aarts
J.K. Lenstra
94/16 R.C. Backhouse
H. Doornbos
94/17 S. Mauw
M.A. Reniers
94/18 F. Kamareddine
R. Nederpelt
94/19 B.W. Watson
94/20 R. Bloo
F. Kamareddine
R. Nederpelt
94/21 B.W. Watson
94/22 B.W. Watson
94/23 S. Mauw and M.A. Reniers
94/24 D. Dams
O. Grumberg
R. Gerth
94/25 T. Kloks
94/26 R.R. Hoogerwoord
94/27 S. Mauw and H. Mulder
94/28 C.W.A.M. van Overveld
M. Verhoeven
94/29 J. Hooman
94/30 J.C.M. Baeten
J.A. Bergstra
Gh. Ştefănescu
94/31 B.W. Watson
R.E. Watson
94/32 J.J. Vereijken
94/33 T. Laan
94/34 R. Bloo
F. Kamareddine
R. Nederpelt
94/35 J.C.M. Baeten
S. Mauw
94/36 F. Kamareddine
R. Nederpelt
94/37 T. Basten
R. Bol
M. Voorhoeve
94/38 A. Bijlsma
C.S. Scholten

The testing Paradigm Applied to Network Structure, p. 31.


A New Method for Integrity Constraint checking in Deductive Databases, p. 34.

Ups and Downs of Type Theory, p. 9.

Job Shop Scheduling by Local Search, p. 21.

Mathematical Induction Made Calculational, p. 36.

An Algebraic Semantics of Basic Message Sequence Charts, p. 9.

Refining Reduction in the Lambda Calculus, p. 15.

The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

Beyond β-Reduction in Church's λ→, p. 22.


The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.

An algebraic semantics of Message Sequence Charts, p. 43.

Abstract Interpretation of Reactive Systems: Abstractions Preserving VCTL*, ECTL* and CTL*, p. 28.

K₁,₂-free and W₄-free graphs, p. 10.

On the foundations of functional programming: a programmer's point of view, p. 54.


Correctness of Real Time Systems by Construction, p. 22.

Process Algebra with Feedback, p. 22.

A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.


A formalization of the Ramified Type Theory, p.40.

The Barendregt Cube with Definitions and Generalised Reduction, p. 37.

A formalization of the Ramified Type Theory, p. 37.

Delayed choice: an operator for joining Message Sequence Charts, p. 15.

Canonical typing and Π-conversion in the Barendregt Cube, p. 19.

Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.

Point-free substitution, p. 10.
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>94/39</td>
<td>On the equivalence covering number of spligraphs, p. 4.</td>
<td></td>
</tr>
<tr>
<td>94/40</td>
<td>Distributed Consensus and Hard Real-Time Systems, p. 34.</td>
<td></td>
</tr>
<tr>
<td>94/41</td>
<td>Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.</td>
<td></td>
</tr>
<tr>
<td>94/42</td>
<td>Concatenation of Graphs, p. 7.</td>
<td></td>
</tr>
<tr>
<td>94/43</td>
<td>Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.</td>
<td></td>
</tr>
<tr>
<td>94/44</td>
<td>Verifying Sequentially Consistent Memory, p. 160</td>
<td></td>
</tr>
<tr>
<td>94/45</td>
<td>Tutorial voor de ExSpect-bibliotheek voor &quot;Administratieve Logistiek&quot;, p. 43.</td>
<td></td>
</tr>
<tr>
<td>94/46</td>
<td>The 1-cube with classes of terms modulo conversion, p. 16.</td>
<td></td>
</tr>
<tr>
<td>94/47</td>
<td>On Π-conversion in Type Theory, p. 12.</td>
<td></td>
</tr>
<tr>
<td>94/48</td>
<td>Fixed-Point Calculus, p. 11.</td>
<td></td>
</tr>
<tr>
<td>94/50</td>
<td>A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.</td>
<td></td>
</tr>
<tr>
<td>94/51</td>
<td>Listing simplicial vertices and recognizing diamond-free graphs, p. 4.</td>
<td></td>
</tr>
<tr>
<td>94/52</td>
<td>Traces and Logic, p. 81</td>
<td></td>
</tr>
<tr>
<td>95/01</td>
<td>The Construction of a small Communication Library, p. 16.</td>
<td></td>
</tr>
<tr>
<td>95/02</td>
<td>Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.</td>
<td></td>
</tr>
<tr>
<td>95/03</td>
<td>Concrete process algebra, p. 134.</td>
<td></td>
</tr>
<tr>
<td>95/04</td>
<td>An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.</td>
<td></td>
</tr>
<tr>
<td>95/05</td>
<td>A Type Inference Algorithm for Pure Type Systems, p.20.</td>
<td></td>
</tr>
<tr>
<td>95/06</td>
<td>A Quantitative Analysis of Iterated Local Search, p.23.</td>
<td></td>
</tr>
<tr>
<td>95/07</td>
<td>Drawing Execution Graphs by Parsing, p. 10.</td>
<td></td>
</tr>
<tr>
<td>95/08</td>
<td>Preservation of Strong Normalisation for Explicit Substitution, p. 12.</td>
<td></td>
</tr>
<tr>
<td>95/09</td>
<td>Discrete Time Process Algebra, p. 20.</td>
<td></td>
</tr>
<tr>
<td>95/10</td>
<td>Mathpad: A System for On-Line Preparation of Mathematical Documents, p. 15</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>95/11</td>
<td>R. Seljé</td>
<td>Deductive Database Systems and integrity constraint checking, p. 36.</td>
</tr>
<tr>
<td>95/12</td>
<td>S. Mauw and M. Reniers</td>
<td>Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.</td>
</tr>
</tbody>
</table>