TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

A Boyer-Moore type algorithm for regular expression pattern matching

Citation for published version (APA):
Watson, B. W., & Watson, R. E. (1994). *A Boyer-Moore type algorithm for regular expression pattern matching*. (Computing science notes; Vol. 9431). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1994

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

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.

Download date: 04. Oct. 2023

Eindhoven University of Technology

Department of Mathematics and Computing Science

A Boyer-Moore type algorithm for regular expression pattern matching

by

Bruce W. Watson and Richard E. Watson

94/31

ISSN 0926-4515

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

Computing Science Report 94/31
Eindhoven, January 1995

A Boyer-Moore type algorithm for regular expression pattern matching

Bruce W. Watson
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
watson@win.tue.nl

Richard E. Watson
Dept. of Mathematics
Simon Fraser University
Burnaby
B.C., Canada
watsona@sfu.ca

January 24, 1995

Abstract

This paper presents a Boyer-Moore type algorithm for regular expression pattern matching, answering an open problem posed by A. V. Aho in 1980 [Aho80, p. 342]. The new algorithm handles patterns specified by regular expressions — a generalization of the Boyer-Moore and Commentz-Walter algorithms (which deal with patterns that are single keywords and finite sets of keywords, respectively).

Like the Boyer-Moore and Commentz-Walter algorithms, the new algorithm makes use of shift functions which can be precomputed and tabulated. The precomputation algorithms are derived, and it is shown that the required shift functions can be precomputed from Commentz-Walter's shift functions known as d_1 and d_2.

In certain cases, the Boyer-Moore (Commentz-Walter) algorithm has greatly outperformed the Knuth-Morris-Pratt (Aho-Corasick) algorithm. In testing, the algorithm presented in this paper also frequently outperforms the regular expression generalization of the Aho-Corasick algorithm.

Contents

1 **Introduction** 3

2 **Mathematical preliminaries** 4

3 **Problem specification and a simple first algorithm** 5
3.1 A more practical algorithm using a finite automaton 6

4 **Greater shift distances** 8
4.1 A more efficient algorithm by computing a greater shift 9
4.2 Deriving a practical range predicate 10

5 **Precomputation** 12
5.1 Characterizing the domains of functions d_1 and d_2 12
5.2 Precomputing function t 13
5.3 Precomputing functions d_1 and d_2 14
5.4 Precomputing function f_r 15
5.5 Precomputing sets L_q 16
5.6 Precomputing function emm 17
5.7 Precomputing function st and languages L' and $\mathbf{suff}(L')$ 18
5.8 Precomputing relation X 18
5.9 Combining the precomputation algorithms 19

6 **Specializing the pattern matching algorithm** 19

7 **Performance of the algorithm** 20

8 **Conclusions** 21

References 22

1 Introduction

The pattern matching problem is: given a non-empty language L (over an alphabet[1] V) and an input string S (also over alphabet V), find all substrings of S that are in L. Several restricted forms of this problem have been solved (all of which are discussed in detail in [Aho90] and [WZ92]):

- The Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77] algorithms solve the problem when L consists of a single word (the single keyword pattern matching problem).

- The Aho-Corasick [AC75] and Commentz-Walter [Com79a, Com79b] algorithms solve the problem when L is a finite set of (key)words (the multiple keyword pattern matching problem). The Aho-Corasick and Commentz-Walter algorithms are generalizations of the Knuth-Morris-Pratt and Boyer-Moore algorithms respectively.

- The case where L is a regular language (the regular expression pattern matching problem) can be solved as follows: a finite automaton is constructed for the language V^*L; each time the automaton enters a final state (while processing the input string S) a matching substring has been found. This algorithm is detailed in [Aho90]. It is a generalization of the Knuth-Morris-Pratt and Aho-Corasick algorithms. Most practical algorithms solving the regular expression pattern matching problem are variants of this (V^*L) algorithm.

Although the Knuth-Morris-Pratt and Aho-Corasick algorithms have better worst-case running time than the Boyer-Moore and Commentz-Walter algorithms (respectively), the latter two algorithms are known to be extremely efficient in practice [HS91, Wat94]. The single and multiple keyword pattern matching algorithms are derived (with proofs) in [WZ92]. Interestingly, to date no generalization (to the case where L is a regular language) of the Boyer-Moore and Commentz-Walter algorithms has been discovered. In [Aho80, p. 342], A.V. Aho states the following open problem:

> "It would also be interesting to know whether there exists a Boyer-Moore type algorithm for regular expression pattern matching."

In this paper, we present such an algorithm. As with the Boyer-Moore and Commentz-Walter algorithms, the new algorithm requires shift tables. The precomputation of these shift table is discussed, and shown to be related to the shift tables used by the Commentz-Walter algorithm. Finally, the new algorithm is specialized to obtain the Boyer-Moore (single keyword) algorithm — showing that it is indeed a generalization of the Boyer-Moore algorithm. The algorithm has been implemented, and in practice it frequently displays better performance than the traditional (V^*L finite automaton) algorithm.

This paper is structured as follows:

- Section 2 presents the mathematical definitions and properties required for reading this paper.

- Section 3 gives the problem specification, and a simple first algorithm.

[1] An alphabet is a finite, non-empty set of symbols. Throughout this paper we assume a fixed alphabet V.

- Section 4 presents the essential idea of greater shift distances while processing the input text, as in the Boyer-Moore algorithm.
- Section 5 derives algorithms required for the precomputation of the shift functions used in the pattern matching algorithm.
- Section 6 specializes the new pattern matching algorithm to obtain the Boyer-Moore algorithm.
- Section 7 provides some data on the performance of the new algorithm versus the generalization of the Aho-Corasick algorithm.
- Section 8 presents the conclusions of this paper.

All of the algorithms are presented in the guarded command language of Dijkstra [Dij76], using Dijkstra's style of proof and program derivation.

Acknowledgements: We would like to thank the following people (in alphabetical order) for their assistance in the preparation of this paper: Kees Hemerik, F. E. J. Kruseman Aretz, Nanette Saes, Tom Verhoeff, and Gerard Zwaan.

2 Mathematical preliminaries

We now present some required definitions (most of which are taken from [WZ92]). For any set G, $\mathcal{P}(G)$ denotes the set of all subsets of G. For $n \geq 0$, V^n denotes the set of all words over V of length n. We have the property that $V^0 = \{\epsilon\}$, where ϵ denotes the empty word. Define $V^* = (\cup\ i : 0 \leq i : V^i)$ (V^* denotes the set of all words over alphabet V, including ϵ). For any $w \in V^*$, define $|w|$ to be the length of w.

The remaining definitions and properties are divided into five groups: prefixes and suffixes of strings, language theoretic properties, string and language reversal, operators for manipulating substrings, and properties of **min** and **max**. In the following definitions assume that $A, B \subseteq V^*$ are languages over alphabet V.

1. Define functions $\mathbf{pref}, \mathbf{suff} : \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ as:

$$\begin{aligned} \mathbf{pref}(A) &= \{x : (\exists\ y : y \in V^* : xy \in A)\} \\ \mathbf{suff}(A) &= \{y : (\exists\ x : x \in V^* : xy \in A)\} \end{aligned}$$

   That is, $\mathbf{pref}(A)$ is the set of all prefixes of words in A, while $\mathbf{suff}(A)$ is the set of all suffixes of words in A. When $w \in V^*$, we will take $\mathbf{pref}(w)$ to mean $\mathbf{pref}(\{w\})$ and $\mathbf{suff}(w)$ to mean $\mathbf{suff}(\{w\})$. If $A \neq \emptyset$ we have $\epsilon \in \mathbf{pref}(A) \wedge \epsilon \in \mathbf{suff}(A)$. Note that $A \subseteq \mathbf{pref}(A)$, $A \subseteq \mathbf{suff}(A)$, $\mathbf{pref}(\mathbf{pref}(A)) = \mathbf{pref}(A)$ and $\mathbf{suff}(\mathbf{suff}(A)) = \mathbf{suff}(A)$ (i.e. **pref** and **suff** are idempotent). We also define $\leq_p$ to be a partial order on strings (known as the prefix order) as $u \leq_p v \equiv u \in \mathbf{pref}(v)$. Function **suff** has the property that:

$$\mathbf{suff}(A) \cap B \neq \emptyset \equiv A \cap V^*B \neq \emptyset \tag{1}$$

2. The following language-theoretic property will be used in the algorithm derivation:

$$(V^*A \cap V^*B \neq \emptyset) \equiv (V^*A \cap B \neq \emptyset) \vee (A \cap V^*B \neq \emptyset) \tag{2}$$

3. We use post-fix (superscript) operator R to denote the reversal of words and languages. Note the following properties of reversal:

$$(A^R)^R = A \tag{3}$$
$$A \subseteq B \equiv A^R \subseteq B^R \tag{4}$$
$$\mathbf{pref}(A)^R = \mathbf{suff}(A^R) \tag{5}$$
$$\mathbf{suff}(A)^R = \mathbf{pref}(A^R) \tag{6}$$

4. Since we wish to operate on strings at a high level (without resorting to indexing the individual symbols in strings), we define some operators on strings. For any string $w \in V^*$, we define:

   - $w\rceil k$ to be the $k \,\mathbf{min}\, |w|$ leftmost symbols of w;
   - $w\lceil k$ to be the $k \,\mathbf{min}\, |w|$ rightmost symbols of w;
   - $w\rfloor k$ to be the $(|w| - k) \,\mathbf{max}\, 0$ rightmost symbols of w;
   - $w\lfloor k$ to be the $(|w| - k) \,\mathbf{max}\, 0$ leftmost symbols of w.

   (The four operators $\rceil, \lceil, \rfloor, \lfloor$ are pronounced "left take," "right take," "left drop," and "right drop" respectively.) For example $(baab)\rceil 3 = baa$, $(baab)\lceil 5 = baab$, $(baab)\rfloor 1 = aab$, and $(baab)\lfloor 10 = \epsilon$.

5. Given that universal quantification over a finite domain is shorthand for conjunction, and existential quantification over a finite domain is shorthand for disjunction, we have the following general properties (where P is some range predicate, f is some function, and the $\forall$ and $\exists$ quantified j is over a finite domain):

$$(\mathbf{MIN}\, i : (\forall\, j :: P(i,j)) : f(i)) \geq (\mathbf{MAX}\, j :: (\mathbf{MIN}\, i : P(i,j) : f(i))) \tag{7}$$
$$(\mathbf{MIN}\, i : (\exists\, j :: P(i,j)) : f(i)) = (\mathbf{MIN}\, j :: (\mathbf{MIN}\, i : P(i,j) : f(i))) \tag{8}$$

3 Problem specification and a simple first algorithm

Formally, the regular expression pattern matching problem is: given a regular expression E (the pattern expression), regular language $L \subseteq V^*$ (the pattern language denoted by E)[2], and input string $S \in V^*$, establish postcondition R:

$$O = \{(l, v, r) : lvr = S \wedge v \in L\}$$

When R holds, variable O is a set of triples (decompositions of input string S), each denoting a pattern occurrence and its left and right context within S. An equivalent specification of the postcondition is:

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

[2]In the remainder of this paper, we will use language L instead of regular expression E in order to make the algorithm derivation more readable.

This will prove to be more useful in the algorithm derivation.

To create a practical first algorithm, the postcondition R can be rewritten as:

$$O = (\cup\, u, r : ur = S : (\cup\, l, v : lv = u : \{l\} \times (\{v\} \cap L) \times \{r\}))$$

We can now give a first algorithm, in which the prefixes (u) of S and the suffixes (v) of u are considered in order of increasing length[3] (where **cand** is conditional conjunction):

Algorithm 3.1:

$u, r, O := \epsilon, S, \{\epsilon\} \times (\{\epsilon\} \cap L) \times \{S\};$
do $r \neq \epsilon \longrightarrow$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1;$
  $l, v := u, \epsilon;$
  $O := O \cup (\{l\} \times (\{v\} \cap L) \times \{r\});$
  do $l \neq \epsilon$ **cand** $(l\upharpoonright 1)v \in \mathbf{suff}(L) \longrightarrow$
    $l, v := l\downharpoonright 1, (l\upharpoonright 1)v;$
    $O := O \cup (\{l\} \times (\{v\} \cap L) \times \{r\})$
  od
od $\{R\}$

This algorithm is taken from [WZ92, Algorithm 2.5]. The number of iterations of the inner repetition is $\mathcal{O}(|S| \cdot ((\mathbf{MAX}\, w : w \in L : |w|)\, \mathbf{min}\, |S|))$. (This is not the same as the running time, as we have not taken the cost of operations such as $\{v\} \cap L$ into account.) The implementation of guard $(l\upharpoonright 1)v \in \mathbf{suff}(L)$ and expression $\{v\} \cap L$ (in the update of variable O) remain unspecified. In order to make the algorithm more practical, we introduce a finite automaton.

3.1 A more practical algorithm using a finite automaton

Since L is a regular language, we construct (from E) a (possibly non-deterministic) ϵ-transition-free finite automaton $M = (Q, V, \delta, I, F)$ accepting L^R (the reverse language[4] of L), where:

- Q is the set of states of M.
- V is our fixed alphabet.
- $\delta : \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(Q)$ is the transition function. (In the case of a deterministic automaton, the (possibly partial) transition function would be $\delta : Q \times V \not\longrightarrow Q$.) Function $\delta^* : \mathcal{P}(Q) \times V^* \longrightarrow \mathcal{P}(Q)$ is the usual Kleene closure of function δ, defined inductively as $\delta^*(H, \epsilon) = H$ and (for $a \in V, w \in V^*$) $\delta^*(H, aw) = \delta^*(\delta(H, a), w)$. (The signatures of transition functions δ and δ^* are slightly different from the ones usually found in textbooks. The signatures used in this paper are notational conveniences which shorten some of the derivations, and simplify the extension of function δ to function δ^*.)

[3] Other orders of evaluation can also be used. This order is only chosen so as to arrive at an algorithm generally resembling the Boyer-Moore algorithm.

[4] The reverse is used, since we will be using automaton M to consider the symbols of substring v in right-to-left order instead of left-to-right order.

- $I \subseteq Q$ is the set of initial (i.e. start) states.
- $F \subseteq Q$ is the set of final states.

(Such an automaton would normally be constructed from regular expression E; algorithms doing this are, for instance, presented in [Wat93a, Constrs. 4.32, 4.39, 4.45, 4.50, 5.34, 5.69, 5.75, 5.82].) Since M is ϵ-transition-free, we have the property that $\epsilon \in L \equiv I \cap F \neq \emptyset$. Finite automata with ϵ-transitions could have been used; they are only excluded in order to simplify the definitions given here.

To give the invariant in the new algorithm, we define a function $\mathcal{L} : Q \longrightarrow \mathcal{P}(V^*)$ as

$$\mathcal{L}(q) = \{w : w \in V^* \land q \in \delta^*(I, w)\}$$

Function $\mathcal{L}$ maps each state q to the set of all words (in V^*) taking M from an initial state to the state q. (Note, by definition: $L^R = (\cup\ f : f \in F : \mathcal{L}(f))$.) For each state q, we also define constant m_q to be the length of a shortest word in $\mathcal{L}(q)$. Define m to be the length of a shortest word in L.

The new algorithm using finite automaton M is (C is a new variable ranging over $\mathcal{P}(Q)$):

Algorithm 3.2:

$u, r, O := \epsilon, S,$ **if** $I \cap F \neq \emptyset$ **then** $\{(\epsilon, \epsilon, S)\}$ **else** $\emptyset$ **fi**;
do $r \neq \epsilon \longrightarrow$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1;$
  $l, v, C := u, \epsilon, I;$
  $O := O \cup$ **if** $C \cap F \neq \emptyset$ **then** $\{(l, v, r)\}$ **else** $\emptyset$ **fi**;
  $\{$invariant: $C = \{q : q \in Q \land v^R \in \mathcal{L}(q)\} \land u = lv\}$
  do $l \neq \epsilon$ **cand** $\delta(C, l{\upharpoonright}1) \neq \emptyset \longrightarrow$
    $l, v, C := l{\downharpoonright}1, (l{\upharpoonright}1)v, \delta(C, l{\upharpoonright}1);$
    $\{C \cap F \neq \emptyset \equiv v \in L\}$
    $O := O \cup$ **if** $C \cap F \neq \emptyset$ **then** $\{(l, v, r)\}$ **else** $\emptyset$ **fi**
  od
od $\{R\}$

String v is reversed in the inner repetition invariant conjunct $C = \{q : q \in Q \land v^R \in \mathcal{L}(q)\}$ since v is processed in reverse. Given this conjunct, the conditional conjunct of the inner repetition guard is now $\delta(C, l{\upharpoonright}1) \neq \emptyset$, since $\delta(C, l{\upharpoonright}1) \neq \emptyset \equiv (l{\upharpoonright}1)v \in \mathbf{suff}(L)$.

There are a number of choices in the implementation of the finite automaton M. In particular, if a deterministic finite automaton is used then the algorithm variable C would always be a singleton set (and the algorithm would be modified so that C ranges over Q instead of $\mathcal{P}(Q)$). The use of a deterministic automaton requires more costly precomputation (of the automaton), but enables the algorithm to process input string S faster. A non-deterministic automaton would involve cheaper precomputation, but the input string would be processed more slowly as all paths in the automaton are simulated. A hybrid solution is to begin with a non-deterministic automaton, and then construct (and tabulate) a deterministic automaton on-the-fly, as the non-deterministic automaton is simulated. In this paper, we continue to use a possibly non-deterministic finite automaton.

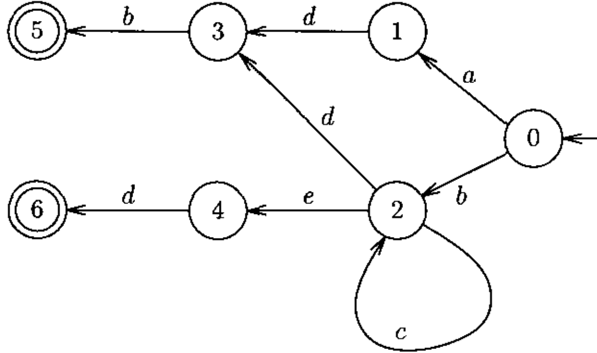

Figure 1: A finite automaton accepting the regular language $L^R = \{b\}\{c\}^*\{db, ed\} \cup \{adb\}$. The states are shown as circles (final states are depicted as concentric circles, and the single initial state is the rightmost state).

Example: As an example of a regular language pattern, and a corresponding finite automaton, consider the language $L = \{bd, de\}\{c\}^*\{b\} \cup \{bda\}$ (over alphabet $V = \{a, b, c, d, e\}$). In this case, the automaton M (which is shown in Figure 1) accepts the language $L^R = \{b\}\{c\}^*\{db, ed\} \cup \{adb\}$. Coincidentally, automaton M is deterministic. Within examples, we will use names (such as L, V, and M) to refer to the concrete objects defined above, as opposed to the abstract objects used elsewhere in the paper. The languages of each of the states (for the automaton in Figure 1) are as follows:

$$
\begin{aligned}
\mathcal{L}(0) &= \{\epsilon\} \\
\mathcal{L}(1) &= \{a\} \\
\mathcal{L}(2) &= \{b\}\{c\}^* \\
\mathcal{L}(3) &= \{ad\} \cup \{b\}\{c\}^*\{d\} \\
\mathcal{L}(4) &= \{b\}\{c\}^*\{e\} \\
\mathcal{L}(5) &= \{adb\} \cup \{b\}\{c\}^*\{db\} \\
\mathcal{L}(6) &= \{b\}\{c\}^*\{ed\}
\end{aligned}
$$

Additionally, $m = 3$, $m_0 = 0$, $m_1 = m_2 = 1$, $m_3 = m_4 = 2$, and $m_5 = m_6 = 3$. Language L and automaton M will be used as our running example throughout the paper. (End of example.)

4 Greater shift distances

Upon termination of the inner repetition, we know (by the invariant of the inner repetition) that $C = \{q : q \in Q \wedge v^R \in \mathcal{L}(q)\}$. This implies $(\forall\ q : q \in C : v^R \in \mathcal{L}(q))$, and equivalently

$$(\forall\ q : q \in C : v \in \mathcal{L}(q)^R)$$

In a manner analogous to the Boyer-Moore algorithm, this information can be used on a subsequent iteration of the outer repetition to make a shift of more than one symbol in the

assignment:

$$u, r := u(r\upharpoonright 1), r\downharpoonright 1$$

In order to make use of this information (which relates v and C) on the first iteration of the outer repetition, we make the invariant of the inner repetition an invariant of the outer repetition as well, by adding the (redundant) initialization $l, v, C := u, \epsilon, I$ before the outer repetition[5]:

Algorithm 4.1:

$u, r, O := \epsilon, S,$ **if** $I \cap F \neq \emptyset$ **then** $\{(\epsilon, \epsilon, S)\}$ **else** $\emptyset$ **fi**;
$l, v, C := u, \epsilon, I$;
$\{$invariant: $C = \{q : q \in Q \wedge v^R \in \mathcal{L}(q)\} \wedge u = lv\}$
do $r \neq \epsilon \longrightarrow$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1$;
  $l, v, C := u, \epsilon, I$;
  $O := O \cup$ **if** $C \cap F \neq \emptyset$ **then** $\{(l, v, r)\}$ **else** $\emptyset$ **fi**;
  $\{$invariant: $C = \{q : q \in Q \wedge v^R \in \mathcal{L}(q)\} \wedge u = lv\}$
  do $l \neq \epsilon$ **cand** $\delta(C, l\upharpoonright 1) \neq \emptyset \longrightarrow$
    $l, v, C := l\downharpoonleft 1, (l\upharpoonleft 1)v, \delta(C, l\upharpoonleft 1)$;
    $\{C \cap F \neq \emptyset \equiv v \in L\}$
    $O := O \cup$ **if** $C \cap F \neq \emptyset$ **then** $\{(l, v, r)\}$ **else** $\emptyset$ **fi**
  od
od $\{R\}$

4.1 A more efficient algorithm by computing a greater shift

We wish to use a greater shift distance in the assignment $u, r := u(r\upharpoonright 1), r\downharpoonright 1$. Ideally, we require the shift distance to the nearest match to the right (in input string S). Formally, this distance is given by: $(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\upharpoonright n)) \cap L \neq \emptyset : n)$. Computing this shift is as difficult as the problem that we are trying to solve. Fortunately, we can settle for any shift approximation k satisfying

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

(Note that a **MIN** quantification with an empty range has the value $+\infty$.) The assignment of u, r then becomes $u, r := u(r\upharpoonright k), r\downharpoonright k$. Consider the range predicate of the **MIN** quantification (the ideal shift); any weakening of the range predicate's second conjunct, $\mathbf{suff}(u(r\upharpoonright n)) \cap L \neq \emptyset$, will give a valid approximation. (Note that, by using the weakest predicate (*true*) we trivially obtain the algorithm above, with a constant shift distance of 1.) We begin by finding a more effective weakening; later we will show that it is practical to precompute the resulting approximation.

Assuming $1 \leq n \leq |r|$ and the (implied) invariant $(\forall\, q : q \in C : v \in \mathcal{L}(q)^R) \wedge u = lv$, we begin with the range predicate:

[5]This does not change the nature of the algorithm, other than creating a new outer repetition invariant.

$$
\begin{array}{ll}
& \mathbf{suff}(u(r{\upharpoonright}n)) \cap L \neq \emptyset \\
\equiv & \quad \{\text{invariant: } u = lv\} \\
& \mathbf{suff}(lv(r{\upharpoonright}n)) \cap L \neq \emptyset \\
\Rightarrow & \quad \{\text{domain of } l, r \text{ and } n\text{: } l \in V^* \text{ and } n \leq |r|, \text{ so } (r{\upharpoonright}n) \in V^n\} \\
& \mathbf{suff}(V^* v V^n) \cap L \neq \emptyset \\
\equiv & \quad \{\text{property (1) of } \mathbf{suff}\} \\
& V^* v V^n \cap V^* L \neq \emptyset \\
\Rightarrow & \quad \{\text{invariant: } (\forall\, q : q \in C : v \in \mathcal{L}(q)^R)\} \\
& (\forall\, q : q \in C : V^* \mathcal{L}(q)^R V^n \cap V^* L \neq \emptyset)
\end{array} \tag{9}
$$

The predicate is now free of l, v, r and S and depends only on state set C, automaton M, and language L. We will continue this derivation from the last line.

The fact that the language L and the languages $\mathcal{L}(q)$ can be infinite (for a given $q \in Q$) makes evaluation of this predicate difficult. In the following subsection, we derive a more practical range predicate.

4.2 Deriving a practical range predicate

We aim at a finite language L_q (corresponding to $q \in Q$) such that $V^*\mathcal{L}(q)^R \subseteq V^*L_q$ and a finite language L' such that $V^*L \subseteq V^*L'$.

Possible definitions of such languages are:

$$
\begin{array}{rcl}
L_q & = & \mathbf{suff}(\mathcal{L}(q)^R) \cap V^{(m_q \,\mathbf{min}\, m)} \\
L' & = & \mathbf{suff}(L) \cap V^m
\end{array}
$$

(The definitions given here were chosen for their simplicity; other definitions are possible, but these particular ones lead to a generalization of the Boyer-Moore algorithm.) In the following intermezzo, we show that these definitions of L_q and L' satisfy the required properties:

> We can see that the definition of L_q satisfies the required property by considering a particular word w:
>
> $$
> \begin{array}{ll}
> & w \in \mathcal{L}(q)^R \\
> \Rightarrow & \quad \{\text{definition of } m_q\text{: } |w| \geq m_q \geq m_q \,\mathbf{min}\, m\} \\
> & (\exists\, x, y : w = xy : y \in \mathbf{suff}(\mathcal{L}(q)^R) \wedge |y| = m_q \,\mathbf{min}\, m) \\
> \equiv & \quad \{\text{definitions of concatenation and intersection of languages}\} \\
> & w \in V^*(\mathbf{suff}(\mathcal{L}(q)^R) \cap V^{(m_q \,\mathbf{min}\, m)}) \\
> \equiv & \quad \{\text{definition of } L_q\} \\
> & w \in V^* L_q
> \end{array}
> $$
>
> We conclude that $\mathcal{L}(q)^R \subseteq V^*L_q$. It follows that $V^*\mathcal{L}(q)^R \subseteq V^*V^*L_q$, and (since $V^*V^* = V^*$) $V^*\mathcal{L}(q)^R \subseteq V^*L_q$. A similar proof applies to the L, L' case.

Example: Given our running example, we can see that $L' = \{bda, bdb, deb, dcb, ecb, ccb\}$ and (for all states $0, \ldots, 6$ in finite automaton M):

$$
L_0 = \{\epsilon\}
$$

$$\begin{aligned} L_1 &= \{a\} \\ L_2 &= \{b\} \\ L_3 &= \{da, db, cb\} \\ L_4 &= \{eb, cb\} \\ L_5 &= \{bda, bdb, dcb, ccb\} \\ L_6 &= \{deb, ecb, ccb\} \end{aligned}$$

(End of example.)

We can continue our previous derivation of a useable range predicate, from (9):

$$\begin{aligned} & (\forall\, q : q \in C : V^*\mathcal{L}(q)^R V^n \cap V^*L \neq \emptyset) \\ \Rightarrow \quad & \{\text{ property: } V^*\mathcal{L}(q)^R \subseteq V^*L_q;\ V^*L \subseteq V^*L' \,\} \\ & (\forall\, q : q \in C : V^*L_q V^n \cap V^*L' \neq \emptyset) \\ \equiv \quad & \{\text{ existentially quantify over all } w \in L_q \,\} \\ & (\forall\, q : q \in C : (\exists\, w : w \in L_q : V^*wV^n \cap V^*L' \neq \emptyset)) \end{aligned}$$

We now have a weakening of the range predicate of the ideal shift distance.

Recalling the properties of **MIN** quantification given in (7) and (8), we can now proceed with our derivation (of an approximation), beginning with the ideal shift distance:

$$\begin{aligned} & (\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r{\upharpoonright}n)) \cap L \neq \emptyset : n) \\ \geq \quad & \{\text{ weakening of range predicate (see derivation above) }\} \\ & (\mathbf{MIN}\, n : 1 \leq n \wedge (\forall\, q : q \in C : (\exists\, w : w \in L_q : V^*wV^n \cap V^*L' \neq \emptyset)) : n) \\ \geq \quad & \{\text{ conjunctive } (\forall) \textbf{ MIN } \text{range predicate — property (7); } |C| \text{ is finite }\} \\ & (\mathbf{MAX}\, q : q \in C : (\mathbf{MIN}\, n : 1 \leq n \wedge (\exists\, w : w \in L_q : V^*wV^n \cap V^*L' \neq \emptyset) : n)) \\ = \quad & \{\text{ disjunctive } (\exists) \textbf{ MIN } \text{range predicate — property (8); } |L_q| \text{ is finite }\} \\ & (\mathbf{MAX}\, q : q \in C : (\mathbf{MIN}\, w : w \in L_q : (\mathbf{MIN}\, n : 1 \leq n \wedge V^*wV^n \cap V^*L' \neq \emptyset : n))) \end{aligned}$$

Recall property (2); this property is also used in the derivation of the Commentz-Walter algorithm [WZ92]. We define two auxiliary functions $d_1, d_2 : V^* \longrightarrow \mathbb{N}$ as:

$$\begin{aligned} d_1(x) &= (\mathbf{MIN}\, n : 1 \leq n \wedge V^*xV^n \cap L' \neq \emptyset : n) \\ d_2(x) &= (\mathbf{MIN}\, n : 1 \leq n \wedge xV^n \cap V^*L' \neq \emptyset : n) \end{aligned}$$

We can now rewrite the inner **MIN** quantification of our shift distance:

$$\begin{aligned} & (\mathbf{MIN}\, n : 1 \leq n \wedge V^*wV^n \cap V^*L' \neq \emptyset : n) \\ = \quad & \{\text{ property (2) }\} \\ & (\mathbf{MIN}\, n : 1 \leq n \wedge (V^*wV^n \cap L' \neq \emptyset \vee wV^n \cap V^*L' \neq \emptyset) : n) \\ = \quad & \{\text{ disjunctive range predicate }\} \\ & (\mathbf{MIN}\, n : 1 \leq n \wedge V^*wV^n \cap L' \neq \emptyset : n)\, \mathbf{min} (\mathbf{MIN}\, n : 1 \leq n \wedge wV^n \cap V^*L' \neq \emptyset : n) \\ = \quad & \{\text{ definitions of } d_1, d_2 \,\} \\ & d_1(w)\, \mathbf{min}\, d_2(w) \end{aligned}$$

The approximation of the ideal shift distance is:

$$(\mathbf{MAX}\, q : q \in C : (\mathbf{MIN}\, w : w \in L_q : d_1(w)\,\mathbf{min}\, d_2(w)))$$

For readability, we define auxiliary function $t : Q \longrightarrow \mathbb{N}$ as

$$t(q) = (\mathbf{MIN}\, w : w \in L_q : d_1(w)\,\mathbf{min}\, d_2(w))$$

Functions d_1, d_2 and t are easily precomputed as discussed in Section 5. The final algorithm (using function t and introducing variable *distance* for readability) is:

Algorithm 4.2:

$u, r, O := \epsilon, S, \mathbf{if}\ I \cap F \neq \emptyset\ \mathbf{then}\ \{(\epsilon, \epsilon, S)\}\ \mathbf{else}\ \emptyset\ \mathbf{fi};$
$l, v, C := u, \epsilon, I;$
$\{\text{invariant: } C = \{q : q \in Q \wedge v^R \in \mathcal{L}(q)\} \wedge u = lv\}$
$\mathbf{do}\ r \neq \epsilon \longrightarrow$
$\quad distance := (\mathbf{MAX}\, q : q \in C : t(q));$
$\quad u, r := u(r \upharpoonright distance), r \downharpoonright distance;$
$\quad l, v, C := u, \epsilon, I;$
$\quad O := O \cup \mathbf{if}\ C \cap F \neq \emptyset\ \mathbf{then}\ \{(l, v, r)\}\ \mathbf{else}\ \emptyset\ \mathbf{fi};$
$\quad \{\text{invariant: } C = \{q : q \in Q \wedge v^R \in \mathcal{L}(q)\} \wedge u = lv\}$
$\quad \mathbf{do}\ l \neq \epsilon\ \mathbf{cand}\ \delta(C, l \upharpoonright 1) \neq \emptyset \longrightarrow$
$\quad\quad l, v, C := l \downharpoonright 1, (l \upharpoonright 1)v, \delta(C, l \upharpoonright 1);$
$\quad\quad \{C \cap F \neq \emptyset \equiv v \in L\}$
$\quad\quad O := O \cup \mathbf{if}\ C \cap F \neq \emptyset\ \mathbf{then}\ \{(l, v, r)\}\ \mathbf{else}\ \emptyset\ \mathbf{fi}$
$\quad \mathbf{od}$
$\mathbf{od}\ \{R\}$

---

5 Precomputation

In this section, we consider the precomputation of languages L_q and L', and functions d_1, d_2, and t. The precomputation is presented as a series of small algorithms — each easier to understand than a single monolithic one. All algorithms are presented and derived in the reverse order of their application. In practice they would be combined into one algorithm, as is shown in Section 5.9.

5.1 Characterizing the domains of functions d_1 and d_2

Since functions d_1, d_2 are only applied to elements of L_q (for all $q \in Q$), their signatures can be taken as $d_1, d_2 : (\cup\ q : q \in Q : L_q) \longrightarrow \mathbb{N}$. In order to make the precomputation of the functions easier, we need a different characterization of their domains. To do this in a simple manner, we require the automaton M to have a simple structural property:

> For all states $q \in Q$, there exists a path (in the transition graph induced by function δ) from an initial state to a final state, such that the path passes through q. (This means that there are no useless states in M.)

The property implies that (for all $q \in Q$) $\mathcal{L}(q) \subseteq \mathbf{pref}((\cup\ f : f \in F : \mathcal{L}(f))) = \mathbf{pref}(L^R)$. In [Wat93a], several general finite automata construction algorithms are given; many of those algorithms construct automata with this property.

From the implication of the property above, and the domain of d_1, d_2, we can restrict the domains of d_1 and d_2 as follows (for all $q \in Q$):

$$
\begin{array}{ll}
 & L_q \\
= & \quad \{\text{ definition of } L_q \} \\
 & \mathbf{suff}(\mathcal{L}(q)^R) \cap V^{m_q \,\mathbf{min}\, m} \\
\subseteq & \quad \{\text{ assumption (structural property) } \mathcal{L}(q) \subseteq \mathbf{pref}(L^R)\text{; property (4); monotonicity of } \mathbf{suff} \} \\
 & \mathbf{suff}(\mathbf{pref}(L^R)^R) \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{\text{ property (5); function } R \text{ is its own inverse — property (3) } \} \\
 & \mathbf{suff}(\mathbf{suff}(L)) \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{\text{ idempotency of } \mathbf{suff} \} \\
 & \mathbf{suff}(L) \cap V^{m_q \,\mathbf{min}\, m} \\
\subseteq & \quad \{\ m_q \,\mathbf{min}\, m \leq m \text{ and } (\forall\ u : u \in \mathbf{suff}(L) \cap V^{m_q \,\mathbf{min}\, m} : (\exists\ v : v \in V^* : vu \in \mathbf{suff}(L) \cap V^m)) \} \\
 & \mathbf{suff}(\mathbf{suff}(L) \cap V^m) \\
= & \quad \{\text{ definition of } L' \} \\
 & \mathbf{suff}(L')
\end{array}
$$

Given this property (of each L_q), we can restrict the domain of functions d_1 and d_2 so that $d_1, d_2 : \mathbf{suff}(L') \longrightarrow \mathbb{N}$. Since $|L'|$ is finite, then $|\mathbf{suff}(L')|$ is finite as well.

Example: In our running example, where $L' = \{bda, bdb, deb, dcb, ecb, ccb\}$, we have

$$\mathbf{suff}(L') = \{\epsilon, a, b, da, db, eb, cb, bda, bdb, deb, dcb, ecb, ccb\}$$

Given the definitions of d_1, d_2, we can compute the two functions by hand from their definitions:

w	ϵ	a	b	da	db	eb	cb	bda	bdb	deb	dcb	ecb	ccb
$d_1(w)$	1	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$d_2(w)$	3	3	2	3	2	2	2	3	2	2	2	2	2

(End of example.)

Before precomputing d_1, d_2, we concentrate on the precomputation of function t.

5.2 Precomputing function t

Assuming that functions d_1, d_2 and sets L_q (for all $q \in Q$) have been precomputed, we can compute function t as follows (variable *tee* is used to accumulate shift function t):

Algorithm 5.1:

---

for $q : q \in Q$ **do**
 $tee(q) := +\infty$
rof;
for $q, u : q \in Q \wedge u \in \mathbf{suff}(L') \wedge u \in L_q$ **do**
 $tee(q) := tee(q) \mathbin{\mathbf{min}} d_1(u) \mathbin{\mathbf{min}} d_2(u)$
rof
$\{tee = t\}$

Notice that we impose no unnecessary order of evaluation in either of the two repetitions. An implementor of this algorithm is free to choose an order of evaluation which is most efficient for the encoding used in the implementation.

Example: In our running example, we obtain the following values for function t (given the values of L_q for all states q, and functions d_1, d_2): $t(0) = 1, t(1) = 3, t(2) = 2, t(3) = 2, t(4) = 2, t(5) = 2, t(6) = 2$. (End of example.)

5.3 Precomputing functions d_1 and d_2

With the domain of functions d_1 and d_2 restricted to $\mathbf{suff}(L')$, functions d_1 and d_2 are the Commentz-Walter precomputed functions for (finite) keyword set L' [Com79a].

We now present two algorithms, computing d_1 and d_2 respectively. The algorithms are fully derived in [WZ94], and are given here without proofs of correctness. The two precomputation algorithms presented below depend upon a function $f_r : \mathbf{suff}(L') \setminus \{\epsilon\} \longrightarrow \mathbf{suff}(L')$ (called the *reverse failure function* corresponding to keyword set L') which is defined as:

$$f_r(u) = (\mathbf{MAX}_{\leq_p}\, w : w \in \mathbf{pref}(u) \setminus \{u\} \cap \mathbf{suff}(L') : w)$$

In the following two algorithms, we assume that function f_r is precomputed:

Algorithm 5.2:

---

for $u : u \in \mathbf{suff}(L')$ **do**
 $dee1(u) := +\infty$
rof;
for $u : u \in \mathbf{suff}(L') \setminus \{\epsilon\}$ **do**
 $dee1(f_r(u)) := dee1(f_r(u)) \mathbin{\mathbf{min}} (|u| - |f_r(u)|)$
rof
$\{dee1 = d_1\}$

Again, notice that we impose no unnecessary order of evaluation in either of the two repetitions.

Algorithm 5.3:

> **for** $u : u \in \mathbf{suff}(L')$ **do**
> $dee2(u) := +\infty$
> **rof**;
> **for** $u : u \in L'$ **do**
> $v := u$;
>   **do** $v \neq \epsilon \longrightarrow$
> $v := f_r(v)$;
>   **if** $|u| - |v| < dee2(v) \longrightarrow dee2(v) := |u| - |v|$
> $[\!]$ $|u| - |v| \geq dee2(v) \longrightarrow v := \epsilon$
>   **fi**
>   **od**
> **rof**;
> $n := 1$;
> **do** $\mathbf{suff}(L') \cap V^n \neq \emptyset \longrightarrow$
>   **for** $u : u \in \mathbf{suff}(L') \cap V^n$ **do**
> $dee2(u) := dee2(u)\ \mathbf{min}\ dee2(u\rfloor 1)$
>   **rof**;
> $n := n + 1$
> **od**
> $\{dee2 = d_2\}$

Notice that the third (un-nested) repetition is a breadth-first traversal of the set $\mathbf{suff}(L')$, and the second (un-nested) repetition requires that function f_r is precomputed. By the definition of language L', the depth of the traversal is m. Precomputation using these algorithms has been found to be cheap in practice [Wat94].

5.4 Precomputing function f_r

The following algorithm (taken largely from [WZ92, Section 6, pg. 33]) computes function f_r:

Algorithm 5.4:

> **for** $a : a \in V$ **do**
>   **if** $a \in \mathbf{suff}(L') \longrightarrow fr(a) := \epsilon$
> $[\!]$ $a \notin \mathbf{suff}(L') \longrightarrow$ **skip**
>   **fi**
> **rof**;
> $n := 1$;
> $\{$invariant: $(\forall\ u : u \in \mathbf{suff}(L') \wedge |u| \leq n : fr(u) = f_r(u))\}$
> **do** $\mathbf{suff}(L') \cap V^n \neq \emptyset \longrightarrow$
>   **for** $u, a : u \in \mathbf{suff}(L') \cap V^n \wedge a \in V$ **do**
>   **if** $au \in \mathbf{suff}(L') \longrightarrow$
> $u' := fr(u)$;
>   **do** $u' \neq \epsilon \wedge au' \notin \mathbf{suff}(L') \longrightarrow$
> $u' := fr(u')$

```
                od;
                if u' = ε ∧ a ∉ suff(L') ⟶ fr(au) := ε
                ▯ u' ≠ ε ∨ a ∈ suff(L') ⟶ fr(au) := au'
                fi
            ▯ au ∉ suff(L') ⟶ skip
            fi
        rof;
        n := n + 1
    od
    {n > m}
    {fr = f_r}
```

This algorithm also makes use of a breadth-first traversal (of depth m) of the set $\mathbf{suff}(L')$.

Example: Consider the function f_r for our running example:

$w : w \in \mathbf{suff}(L') \setminus \{\epsilon\}$	a	b	da	db	eb	cb	bda	bdb	deb	dcb	ecb	ccb
$f_r(w)$	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	b	b	ϵ	ϵ	ϵ	ϵ

(End of example.)

5.5 Precomputing sets L_q

The languages L_q can be precomputed using an auxiliary relation and two auxiliary functions. The auxiliary functions are $st : \mathbf{suff}(L') \longrightarrow \mathcal{P}(Q)$ and $emm : Q \longrightarrow [0, m]$ defined as:

$$\begin{aligned} st(u) &= \{q : q \in Q \wedge u^R \in \mathcal{L}(q)\} \\ emm(q) &= m_q \,\mathbf{min}\, m \end{aligned}$$

The required relation, X, is a binary relation on states (called the *reachability relation*), defined as (for any two state p, q):

$$(p, q) \in X \equiv (\exists\, a : a \in V : q \in \delta(\{p\}, a))$$

A useful property (of any finite automaton) is that (for all states $q \in Q$):

$$\mathbf{pref}(\mathcal{L}(q)) = (\cup\, p : p \in Q \wedge (p, q) \in X^* : \mathcal{L}(p)) \tag{10}$$

This property is given, in a slightly simpler form, in [Wat93b, Property 3.2].

Given relation X and functions emm and st, we can derive an expression for L_q that is easier to compute (than the definition):

$$\begin{aligned} & L_q \\ = & \quad \{\text{ definition of } L_q \,\} \\ & \mathbf{suff}(\mathcal{L}(q)^R) \cap V^{m_q \,\mathbf{min}\, m} \\ = & \quad \{\text{ property (5) }\} \\ & \mathbf{pref}(\mathcal{L}(q))^R \cap V^{m_q \,\mathbf{min}\, m} \\ = & \quad \{\text{ property (10) }\} \end{aligned}$$

$$
\begin{array}{ll}
& (\cup\ p : p \in Q \wedge (p,q) \in X^* : \mathcal{L}(p))^R \cap V^{m_q \,\mathbf{min}\, m} \\
= & \quad \{ \cap \text{ and } R \text{ distribute over } \cup \} \\
& (\cup\ p : p \in Q \wedge (p,q) \in X^* : \mathcal{L}(p)^R \cap V^{m_q \,\mathbf{min}\, m}) \\
= & \quad \{ \text{quantify over all words } w : w \in \mathcal{L}(p)^R \cap V^{m_q \,\mathbf{min}\, m} \} \\
& (\cup\ w,p : p \in Q \wedge (p,q) \in X^* \wedge w \in V^{m_q \,\mathbf{min}\, m} \wedge w \in \mathcal{L}(p)^R : \{w\}) \\
= & \quad \{ w \in V^{m_q \,\mathbf{min}\, m} \equiv |w| = m_q \,\mathbf{min}\, m \equiv |w| = emm(q) \} \\
& (\cup\ w,p : p \in Q \wedge (p,q) \in X^* \wedge w \in \mathbf{suff}(L') \wedge |w| = emm(q) \wedge w \in \mathcal{L}(p)^R : \{w\}) \\
= & \quad \{ w \in \mathcal{L}(p)^R \equiv w^R \in \mathcal{L}(p) \equiv p \in st(w) \} \\
& (\cup\ w,p : p \in Q \wedge (p,q) \in X^* \wedge w \in \mathbf{suff}(L') \wedge |w| = emm(q) \wedge p \in st(w) : \{w\})
\end{array}
$$

Assuming that relation X and auxiliary functions *emm* and *st* are precomputed, we can now present an algorithm computing L_q (for all $q \in Q$):

Algorithm 5.5:

---

for $q : q \in Q$ **do**
    $ell(q) := \emptyset$
rof;
for $p,q,w : (p,q) \in X^* \wedge w \in \mathbf{suff}(L') \wedge |w| = emm(q) \wedge p \in st(w)$ **do**
    $ell(q) := ell(q) \cup \{w\}$
rof
$\{(\forall\ q : q \in Q : ell(q) = L_q)\}$

5.6 Precomputing function *emm*

Assuming that function *st* had already been computed, the following algorithm computes function *emm* using a breadth-first traversal of $\mathbf{suff}(L')$:

Algorithm 5.6:

---

for $q : q \in Q$ **do**
    if $q \in I \longrightarrow emm(q) := 0$
    $\|\ q \notin I \longrightarrow emm(q) := m$
    fi
rof;
$n := 1$;
do $\mathbf{suff}(L') \cap V^n \neq \emptyset \longrightarrow$
    for $u : u \in \mathbf{suff}(L') \cap V^n$ **do**
        for $q : q \in st(u)$ **do**
            $emm(q) := emm(q) \,\mathbf{min}\, n$
        rof
    rof
od
$\{(\forall\ q : q \in Q : emm(q) = m_q \,\mathbf{min}\, m)\}$

5.7 Precomputing function st and languages L' and $\mathbf{suff}(L')$

The following algorithm makes a breadth-first traversal (of depth m) of the transition graph of finite automaton M. It simultaneously computes function st, languages L' and $\mathbf{suff}(L')$, and m (the length of a shortest word in language L).

Languages L' and $\mathbf{suff}(L')$ are used in most of the precomputation algorithms already presented. While the following algorithm computes language $\mathbf{suff}(L')$, it is also an example of a breadth-first traversal of $\mathbf{suff}(L')$ without having to explicitly compute and store the language $\mathbf{suff}(L')$; instead, the algorithm traverses the transition graph of finite automaton M and implicitly performs a breadth-first traversal of $\mathbf{suff}(L')$.

Algorithm 5.7:

$st(\epsilon), current, SLprime, n, final := I, \{\epsilon\}, \{\epsilon\}, 0, (I \cap F = \emptyset);$
{invariant:
 $current = \mathbf{suff}(L') \cap V^n$
 $\wedge\ SLprime = (\cup\ i : i \le n : \mathbf{suff}(L') \cap V^i)$
 $\wedge\ 0 \le n \le m$
 $\wedge\ (final \equiv n = m)$
 $\wedge\ (\forall\ u : u \in \mathbf{suff}(L') \wedge |u| \le n : st(u) = \{q : u^R \in \mathcal{L}(q)\})\}$
do $\neg final \longrightarrow$
 $current' := \emptyset;$
 $n := n + 1;$
 for $u, a : u \in current \wedge a \in V$ **do**
  if $\delta(st(u), a) \ne \emptyset \longrightarrow$
   $\{au \in \mathbf{suff}(L') \cap V^n\}$
   $st(au) := \delta(st(u), a);$
   $\{(\forall\ q : q \in st(au) : au \in \mathcal{L}(q)^R)\}$
   $current' := current' \cup \{au\};$
   $final := final \vee (st(au) \cap F \ne \emptyset)$
  $[\!]\ \delta(st(u), a) = \emptyset \longrightarrow$ **skip**
  fi
 rof;
 $current := current';$
 $SLprime := SLprime \cup current$
od
$\{n = m\}$
$\{current = \mathbf{suff}(L') \cap V^m = L'\}$
$\{SLprime = \mathbf{suff}(L')\}$
$\{(\forall\ u : u \in \mathbf{suff}(L') : st(u) = \{q : u^R \in \mathcal{L}(q)\})\}$

5.8 Precomputing relation X

Relation X can be precomputed using a reachability algorithm which traverses the transition graph of automaton M. Relation X^* can then be precomputed by a reflexive and transitive closure algorithm. The two algorithms are combined into one below:

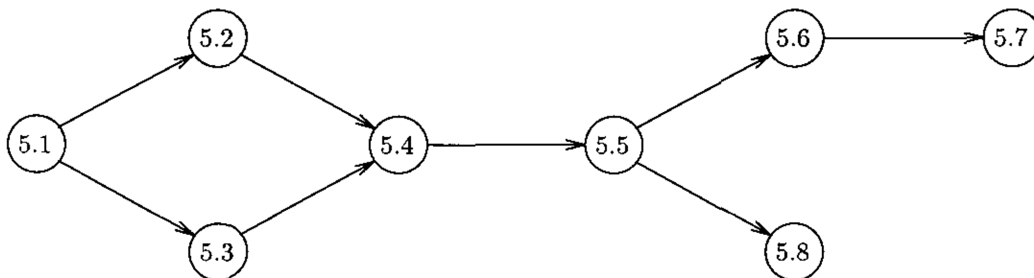

Figure 2: The dependency graph of the precomputation algorithms. An arrow from algorithm a to algorithm b indicates that algorithm b must be applied before algorithm a.

Algorithm 5.8:

$Exstar := \emptyset$;
for $q, a : a \in Q \wedge a \in V$ **do**
    $Exstar := Exstar \cup \{(q,q)\} \cup (\{q\} \times \delta(\{q\}, a))$;
rof;
$\{Exstar = X^0 \cup X^1\}$
$change := true$;
do $change \longrightarrow$
    $change := false$;
    for $p, q, r : (p,q) \in Exstar \wedge (q,r) \in Exstar$ **do**
        $change := change \vee (p,r) \notin Exstar$;
        $Exstar := Exstar \cup \{(p,r)\}$
    rof
od
$\{Exstar = X^*\}$

5.9 Combining the precomputation algorithms

The precomputation algorithms can be combined into a single monolithic algorithm. Such an algorithm is essentially the sequential concatenation of the separate precomputation algorithms. The order in which the algorithms are applied is determined by their dependency graph, which is shown in Figure 2. A possible order of execution is obtained by reversing a topological sort of the dependency graph. One such order is: (Algorithms) 5.8, 5.7, 5.6, 5.5, 5.4, 5.2, 5.3, 5.1.

6 Specializing the pattern matching algorithm

By restricting the form of the regular expression patterns, we can specialize the pattern matching algorithm to obtain the Boyer-Moore and the Commentz-Walter algorithms. In this section, we specialize to obtain the Boyer-Moore algorithm that does not use a lookahead symbol.

To obtain the single-keyword pattern matching problem, we require that L be a singleton set; that is $L = \{p\}$, a language consisting of a single keyword.

We define deterministic finite automaton $M = (\mathbf{suff}(p), V, \gamma, \{\epsilon\}, \{p\})$. The states are elements of $\mathbf{suff}(p)$. We define deterministic transition function $\gamma : \mathbf{suff}(p) \times V \longrightarrow \mathbf{suff}(p) \cup \{\bot\}$ (the special value $\bot$ denotes an undefined transition) as:

$$\gamma(w, a) = \begin{cases} aw & \text{if } aw \in \mathbf{suff}(p) \\ \bot & \text{otherwise} \end{cases}$$

Automaton M satisfies the structural property on page 12. Given function γ, we have (for every state $w \in \mathbf{suff}(p)$):

$$\mathcal{L}(w) = \{w^R\}$$

Automaton M is deterministic, and the current state-set variable (C in the algorithm) is always a singleton set; call it state $w \in \mathbf{suff}(p)$. Since $\mathcal{L}(w)$ is a singleton set and $|w| \leq |p|$, we have $m_w = |w|$ and $L_w = \mathcal{L}(w)^R = \{w\}$. Additionally, since $m = |p|$, $L' = L = \{p\}$. Clearly, we have $L_w \subseteq \mathbf{suff}(L') = \mathbf{suff}(p)$. Function t is defined as $t(w) = d_1(w) \mathbf{min}\, d_2(w)$. The shift distance will then be $d_1(w) \mathbf{min}\, d_2(w)$ in the update of variables u, r. Elements of $\mathbf{suff}(p)$ (in particular, current state variable w) can be encoded as integer indices (into string p) in the range $[0, |p|]$. By making use of this encoding, and changing the domain of the variables u, r and functions d_1, d_2 to make use of indexing in input string S, we obtain the Boyer-Moore algorithm. The Commentz-Walter algorithm can similarly be obtained as a specialization.

7 Performance of the algorithm

Empirical performance data was gathered by implementing this algorithm in a **grep** style pattern matching tool, running under UNIX (on a Sun SPARC Station 1+) and DOS (on a 20 Mhz 386).

On each run, the new algorithm was used in addition to the old (generalized Aho-Corasick) algorithm which constructs a finite automaton accepting the language V^*L. (For both the old and the new algorithms, only deterministic finite automata were used. The time required for precomputation was not measured, but for both algorithms it appeared to be negligible compared to the time required to process the input string.) In the cases where $m \geq 6$ (the length of the shortest word in L is at least 6), and $|L'| \leq 18$, this new algorithm outperforms the other algorithm. These conditions held on approximately 35% of our user-entered regular expression patterns.

In the cases where the new algorithm outperformed the traditional one, the differences in execution speed varied from a 5% improvement to a 150% improvement. In the cases where the new algorithm was outperformed, its execution speed was never less than 30% of the execution speed of the traditional algorithm.

The conditions for obtaining high performance from the new algorithm ($m \geq 6 \wedge |L'| \leq 18$) can easily be determined from automaton M. In a **grep** style pattern matching tool, the automaton M can be constructed for language L^R. If the required conditions are met, the Boyer-Moore type pattern matcher is used. If the conditions are not met, M can be reversed (so that it accepts language L), and converted to an automaton accepting V^*L. The traditional algorithm can then be used.

8 Conclusions

We have achieved our aim of deriving an efficient generalized Boyer-Moore type pattern matching algorithm for regular languages. The stepwise derivation began with a simple, intuitive first algorithm; a finite automaton was introduced to make the implementation practical. The idea of shift distances greater than one symbol (as in the Boyer-Moore and Commentz-Walter algorithms) was introduced. The use of predicate weakening was instrumental in deriving a practical approximation to the ideal shift distance.

Using a structural property of finite automata, the approximation was shown to be the composition of several functions, all but two of which are easily computed. The remaining two functions are the Commentz-Walter shift functions; an algorithm computing these functions has previously been derived with correctness arguments in [WZ92].

The Boyer-Moore algorithm was derived as a special case of our algorithm, showing our algorithm to be a truly generalized pattern matching algorithm.

References

[Aho80] AHO, A.V. Pattern matching in strings, in: R.V. Book, ed., *Formal Language Theory: Perspectives and Open Problems,* (Academic Press, New York, 1980) 325–347.

[Aho90] AHO, A.V. Algorithms for Finding Patterns in Strings, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity,* (Elsevier, Amsterdam, 1990) 256–300.

[AC75] AHO, A.V. and M.J. CORASICK. Efficient string matching: an aid to bibliographic search, *Comm. ACM,* 18(6) (1975) 333–340.

[BM77] BOYER, R.S. and J.S. MOORE. A fast string searching algorithm, *Comm. ACM,* 20(10) (1977) 62–72.

[Com79a] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer, Berlin, 1979) 118–132.

[Com79b] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, Technical report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[Dij76] DIJKSTRA, E.W. *A discipline of programming* (Prentice-Hall Inc., New Jersey, 1976).

[HS91] HUME, A. and D. SUNDAY. Fast string searching, *Software—Practice and Experience,* 21(11) (1991) 1221–1248.

[KMP77] KNUTH, D.E., J.H. MORRIS and V.R. PRATT. Fast pattern matching in strings, *SIAM J. Comput.,* 6(2) (1977) 323–350.

[WZ92] WATSON, B.W. and G. ZWAAN. A taxonomy of keyword pattern matching algorithms, Computing Science Note 92/27, Eindhoven University of Technology, The Netherlands, 1992. Available from `watson@win.tue.nl`.

[WZ94] WATSON, B.W. and G. ZWAAN. The Commentz-Walter family of keyword pattern matching algorithms, to appear as a Computing Science Note, Eindhoven University of Technology, The Netherlands, 1994. Available from `watson@win.tue.nl`.

[Wat93a] WATSON, B.W. A taxonomy of finite automata construction algorithms, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. Available from `watson@win.tue.nl`.

[Wat93b] WATSON, B.W. A taxonomy of finite automata minimization algorithms, Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993. Available from `watson@win.tue.nl`.

[Wat94] WATSON, B.W. The performance of some multiple-keyword pattern matching algorithms, Computing Science Note 94/19, Eindhoven University of Technology, The Netherlands, 1994. Available from `watson@win.tue.nl`.

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of P-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.

94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\!\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.