Eindhoven University of Technology

Department of Mathematics and Computing Science

A taxonomy of keyword pattern matching algorithms

by

B.W. Watson and G.Zwaan

92/27

Computing Science Note 92/27
Eindhoven, December 1992

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

A taxonomy of keyword pattern matching algorithms

B.W. Watson & G. Zwaan
Computing Science Note 92/27
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

December 24, 1992
Revised December 24, 1993

Abstract

This paper presents a taxonomy of keyword pattern matching algorithms, including the well-known Knuth-Morris-Pratt, Aho-Corasick, Boyer-Moore, and Commentz-Walter algorithms and a number of their variants. 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 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). Moreover, the paper provides complete derivations of the intricate precomputation algorithms, some of which either can not be found in the literature (Commentz-Walter) or are given in several different versions (Boyer-Moore).

Contents

1 Introduction and related work **1**

I The algorithm derivations 5

2 The problem and some naive solutions **5**
2.1 The (P_+) algorithms . . . 7
2.1.1 The (P_+S_+) algorithms . . . 7
2.1.2 The (P_+S_-) algorithm . . . 9
2.2 The (S_-) algorithms . . . 9
2.2.1 The (S_-P_+) algorithms . . . 9
2.2.2 The (S_-P_-) algorithm . . . 10

3 The Aho-Corasick algorithms **10**
3.1 Algorithm detail AC . . . 11
3.2 Method OPT . . . 15
3.3 Linear search . . . 16
3.3.1 The Aho-Corasick algorithm with failure function . . . 17
3.3.2 The Knuth-Morris-Pratt algorithm . . . 18

4 The Commentz-Walter algorithms **21**
4.1 Larger shifts . . . 21
4.2 Discarding the lookahead symbol . . . 22
4.3 Using the lookahead symbol . . . 23
4.4 A derivation of the Boyer-Moore algorithm . . . 24

5 The Boyer-Moore family of algorithms **25**
5.1 Larger shifts without using $Match$ information . . . 27
5.2 Making use of $Match$ information . . . 29

II Precomputation 31

6 Precomputation for the Aho-Corasick algorithms **31**

7 Precomputation for the Commentz-Walter algorithms **36**
7.1 Computation of d_1 and d_2 . . . 36
7.2 Computation of d_{no} . . . 39
7.3 Computation of d_3 . . . 39
7.4 Computation of d_{bm} and $char$ . . . 40
7.5 Precomputation of s_1, $char_1$, and $char_2$ . . . 41
7.5.1 Forward matching . . . 41
7.5.2 Backward matching . . . 42

III Conclusions 44

IV Appendices 46

A Calculating the value of a quantification **46**
A.1 A nondeterministic solution 46
A.2 A deterministic solution in the ascending direction 46
A.3 A deterministic solution in the descending direction 46
A.4 Nested quantifications 47

B Definitions and properties **47**

References **51**

1 Introduction and related work

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

Just as the variety of applications has grown, so has the diversity of the solutions. Many of the solutions require a simplification of the problem such as "the patterns are regular languages," or "the patterns are finite languages." The myriad of variations on the problem, along with differing program design methodology, leads to solutions that are difficult to compare to one another.

This report presents a taxonomy of keyword pattern matching algorithms. The main results are summarized in the taxonomy graph presented at the end of this section, and in the conclusions presented in Part III. The taxonomy strives for the following goals:

- to present algorithms in a common framework to permit comparison of algorithms; such a framework is to be an easy to comprehend abstract presentation.
- to emphasize the derivation of an algorithm as a series of refinements to either algorithms or to the problem.
- to factor out common portions of well-known algorithms to facilitate comparison of these algorithms.

This report systematically presents a number of variations of four well-known algorithms in a common framework. Two of the algorithms to be presented require that the set of patterns is a single keyword, while the other two require that the set of patterns is a finite set of keywords. The algorithms are

- the Knuth-Morris-Pratt (KMP) algorithm as presented in [KMP77]. This algorithm matches a single keyword against the input string. Originally, the algorithm was devised to find only the first match in the input string. We will consider a version that finds all occurrences within the input string.
- the Boyer-Moore (BM) algorithm as presented in [BM77]. This is also a single keyword matching algorithm. Several corrections and improvements to this algorithm have been published; a good starting point for these is the bibliographic section of [Aho90].
- the Aho-Corasick (AC) algorithm as presented in [AC75]. This algorithm can match a finite set of keywords in the input string.
- the Commentz-Walter (CW) algorithm as presented in [Com79a, Com79b]. This algorithm can also match a finite set of keywords in the input string. Few papers have been published on this algorithm, and its correctness, time complexity, and precomputation are ill-understood.

These four algorithms are also presented in the overview of [Aho90].

The algorithms will be derived from a common starting point. The derivation proceeds by adding either problem or algorithm details. As a problem detail is added (that is, the problem is made more specific) a change may be possible in the algorithm — in particular, an improvement of efficiency may be possible. This is because the more specific problem may permit some transformation not possible in the more general problem.

Algorithm details are of course added in a correctness-preserving way; they are usually made to improve the efficiency of the algorithm. They may be added to restrict nondeterminacy, or to make a change of representation; either of these changes to an algorithm gives a new algorithm meeting the same specification. A derivation should make clear the differences and similarities of these algorithms; the entire derivation can then be taken to be a taxonomy of the four algorithms (and other related algorithms).

This type of taxonomy development and program derivation has been used in the past. One of the most notable is Broy's sorting algorithm taxonomy [Bro83]. In this taxonomy, algorithm and problem details are also added, starting with a naive solution; the taxonomy arrives at all of the well-known sorting algorithms. A similar taxonomy (which predates the one of Broy) is by Darlington [Dar78]; this taxonomy also considers sorting algorithms. Our particular incarnation of the method of developing a taxonomy was developed in the thesis of Jonkers [Jon82], where it was used to give a taxonomy of garbage collection algorithms. Jonkers' method was then successfully applied to attribute evaluation algorithms by Marcelis in [Mar90].

The recent taxonomy of pattern matching algorithms presented by Hume and Sunday (in [HS91]) gives variations on the Boyer-Moore algorithm; the taxonomy concentrates on many of the practical issues, and provides data on the running time of the variations, and their respective precomputation.

Two important aims of our derivations are clarity and correctness of presentation. Towards both aims, the traditional method of using indexed strings (for the input string and patterns) has been abandoned in this paper; we use a more abstract (but equivalent) presentation. In order to easily provide correctness arguments the guarded command language of [Dij76] is used, rather than a programming language such as Pascal or C.

Part I contains the derivation of the four algorithms named above, along with several intermediate algorithms that are byproducts of the derivation.

Part II details the precomputation of functions necessary for each of the four algorithms.

Part III presents the conclusions.

Part IV contains the appendices. A program skeleton that we will often instantiate is detailed in Appendix A. Definitions and properties of operators and functions are provided in Appendix B.

The taxonomy graph that we arrive at after deriving the algorithms in Part I is shown in figure 1 on page 3. Each vertex corresponds to an algorithm. If the vertex is labeled with a number that number refers to an algorithm in this report. If it is labeled with a page number that page number refers to the page where the algorithm is first mentioned. Each edge corresponds to the addition of either a problem or algorithm detail and is labeled with the name of that detail (a list of detail names follows). Each of the algorithms will either be called by their algorithm number, by their name as found in the literature (for the well known algorithms), or by the parenthesized sequence of all labels along the path from the root to the algorithm's vertex. For example, the algorithm known as the optimized Aho-Corasick algorithm can also be called (P_+, E, AC, OPT) (it is also algorithm 3.3 in this report). All of the well known algorithms appear as leaves in the tree. Due to its labeling the graph can be used as an alternative table of contents to this report. Four algorithm details (P_+, S_+, P_-, and S_-) are actually composed of two separate algorithm details. For example, detail (P_+) is composed of details (P) and detail (+), however the second detail must always follow either detail (P) or detail (S) and so we treat them as a single detail. The edges labeled MO and SL in figure 1 represent generic algorithm details that still have to be instantiated. Possible instantiations are given by the two small trees at the bottom of figure 1. The details and a short description of each of them are as follows:

P (§ 2) Examine prefixes of a given string in any order.

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

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

S (§ 2) Examine suffixes of a given string in any order.

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

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

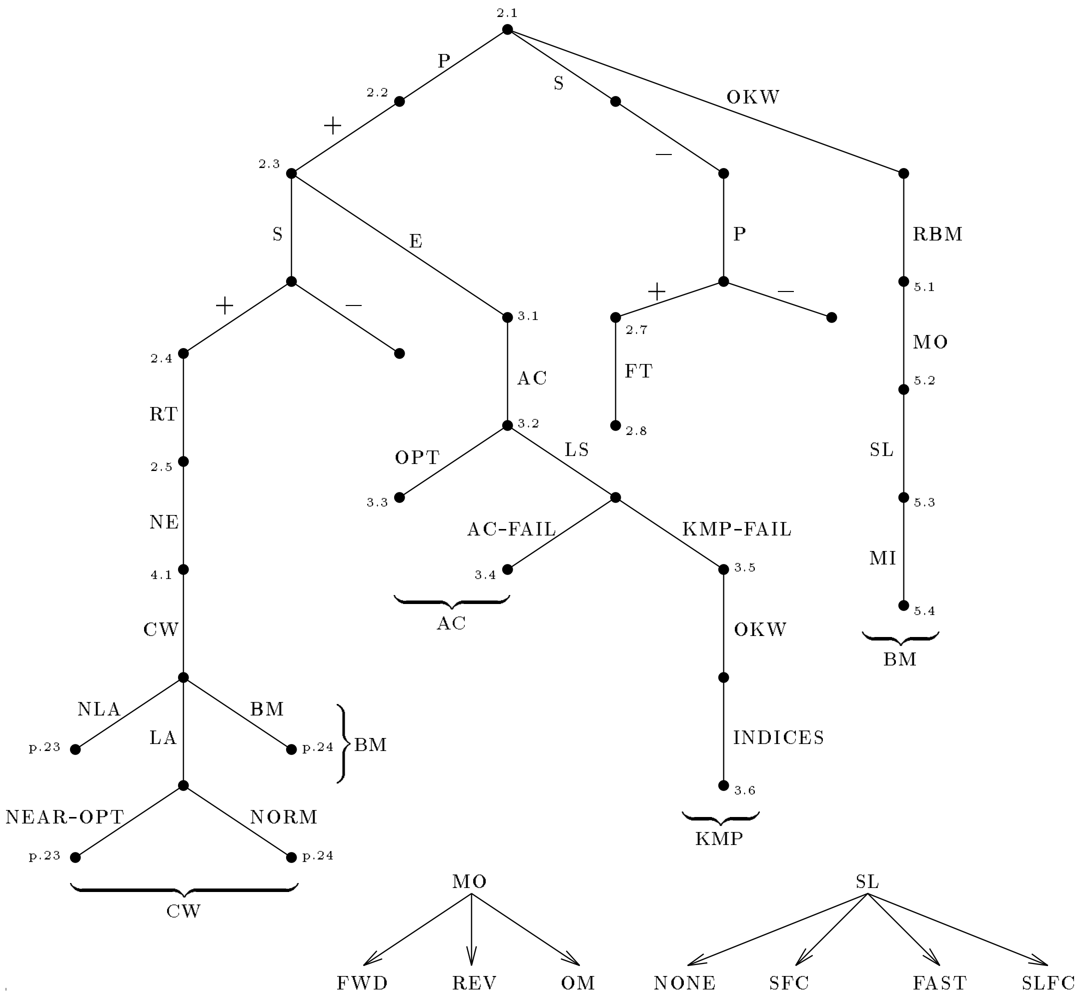

Figure 1: A taxonomy of pattern matching algorithms. An explanation of the graph and its labels is given in the text of this section. Algorithm 3.3 corresponds to the optimized Aho-Corasick algorithm ([AC75], section 6). Algorithm 3.4 corresponds to the Aho-Corasick algorithm using linear search ([AC75], section 2, algorithm 1). Algorithm 3.6 corresponds to the Knuth-Morris-Pratt algorithm ([KMP77], section 2, p.326). The algorithm of the vertex labeled p.24 and with incoming edge labeled NORM corresponds to the Commentz-Walter algorithm ([Com79a], section II, and [Com79b], sections II.1 and II.2). The algorithm of the vertex labeled p.24 and with incoming edge labeled BM corresponds to the Boyer-Moore algorithm ([BM77], section 4). Algorithm 5.4 corresponds to the Boyer-Moore algorithm as well ([BM77], sections 4 and 5).

RT (§ 2.1.1) Usage of the transition function of the reverse trie corresponding to the set of keywords to check whether a string which is a suffix of some keyword, preceded by a character is again a suffix of some keyword.

FT (§ 2.2.1) Usage of the transition function of the forward trie corresponding to the set of keywords to check whether a string which is a prefix of some keyword, followed by a character is again a prefix of some keyword.

E (§ 3) Matches are registered by their endpoint.

AC (§ 3.1) A state variable is maintained while examining prefixes of the input string. The value of the variable is the longest string from the set of all suffixes of the current prefix of the input string, which are prefixes of some keyword.

OPT (§ 3.2) A single "optimized" transition function is used to update the state variable in the Aho-Corasick algorithm.

LS (§ 3.3) Use linear search to update the state variable in the Aho-Corasick algorithm.

AC-FAIL (§ 3.3.1) Implement the linear search using the transition function of the extended forward trie and the failure function.

KMP-FAIL (§ 3.3.2) Implement the linear search using the extended failure function.

OKW (§ 3.3.2) The set of keywords contains one keyword.

INDICES (§ 3.3.2) Represent substrings by indices into the complete strings, converting a string based algorithm into an indexing based algorithm

NE (§ 4) The empty string is not a keyword.

CW (§ 4.1) Consider any shift distance that does not lead to the missing of any matches. Such shift distances are called safe.

NLA (§ 4.2) The lookahead character is not taken into account when computing a safe shift distance. The computation of a shift distance is done by using two precomputed shift functions applied to the current longest partial match.

LA (§ 4.3) The lookahead character is taken into account when computing a safe shift distance.

NEAR-OPT (§ 4.3) Compute a shift distance using a single precomputed shift function applied to the current longest partial match and the lookahead character.

NORM (§ 4.3) Compute a shift distance as in (NLA) but additionally using a third shift function applied to the lookahead character. The shift distance obtained is that of the normal Commentz-Walter algorithm.

BM (§ 4.4) Compute a shift distance using one shift function applied to the lookahead character, and another shift function applied to the current longest partial match. The shift distance obtained is that of the Boyer-Moore algorithm.

RBM (§ 5) Introduce a particular program skeleton as a starting point for the derivation of the different Boyer-Moore variants.

MO (§ 5) A match order is used to determine the order in which characters of a potential match are compared against the keyword. This is only for the one keyword case (OKW). Particular instances of match orders are

FWD (§ 5) The forward match order is used to compare the (single) keyword against a potential match in a left to right direction.

REV (§ 5) The reverse match order is used to compare the (single) keyword against a potential match in a right to left direction. This is the original Boyer-Moore match order.

OM (§ 5) The characters of the (single) keyword are compared in order of ascending probability of occurring in the input string. In this way mismatches will generally be discovered as early as possible.

SL (§ 5.1) Before an attempt at matching a candidate string and the keyword a "skip loop" is used to skip portions of the input that cannot possibly lead to a match. Particular "skips" are

NONE (§ 5.1) No "skip" loop is used.

SFC (§ 5.1 The "skip loop" compares the first character of the match candidate and the keyword; as long as they do not match, the candidate string is shifted one character to the right.

FAST (§ 5.1) As with (SFC), but the last character of the candidate and the keyword are compared, and, possibly, a larger shift distance is used.

SLFC (§ 5.1) As with (FAST), but a low frequency character of the keyword is first compared.

MI (§ 5.2) The information gathered during an attempted match is used (along with the particular match order used during the attempted match) to determine a safe shift distance.

Part I
The algorithm derivations

2 The problem and some naive solutions

The problem is to find all occurrences of any of a set of keywords 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 $P \subseteq V^*$, establish[1]

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

A trivial (but unrealistic) solution to this is

Algorithm 2.1()

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

The sequence of details describing this algorithm is the empty sequence (sequences of details are introduced in section 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". These two possibilities are considered separately below.

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

[1] 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^*.

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

The method of implementing a computation of such a quantification is detailed in Appendix A.

A simple non-deterministic[2] algorithm (the structure of which is discussed in Appendix A.1) is obtained by applying algorithm detail

Detail (P)**:** Examine prefixes of a given string in any order. □

to input string S. It results in[3]

Algorithm 2.2(P)

$W := (\cup\, u,r : ur = S : \{u\} \times \{r\}); O := \emptyset;$
for $(u,r) : (u,r) \in W$ **do**
    $O := O \cup (\cup\, l,v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})$
rof $\{R\}$

Again starting from algorithm 2.1() we can also consider "prefixes of suffixes of S" as follows:

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

Introduction of algorithm detail

Detail (S)**:** Examine suffixes of a given string in any order. □

yields the simple non-deterministic algorithm (S) which is analogous to algorithm 2.2(P). Hence, it is not presented here.

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

Each of (PS) and (SP) 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. For each of (PS) and (SP) this gives two binary choices. Along with the binary choice between (PS) and (SP) this gives a 3-cube representing the three binary choices; the cube is depicted in figure 2 on page 7 with vertices representing the eight possible algorithms for the two nested repetitions. The edges marked '=' join algorithms which are symmetrical; for example, the order in which (P_+S_-) considers S and P is mirrored (with respect to string reversal of S and P) by the order in which (S_+P_-) considers S and P. Because of this symmetry, we present only four algorithms in this section : (P_+S_+), (P_+S_-), (S_-P_-), and (S_-P_+). These algorithms were chosen because their outer repetitions examine S in left to right order. In subsection 2.1 algorithm 2.2(P) will be refined further and in subsection 2.2 algorithm (S) will be refined. In section 3 algorithm (P_+) will be developed into the Aho-Corasick and Knuth-Morris-Pratt algorithms, while in sections 4 and 5 algorithm (P_+S_+) will be developed into the Commentz-Walter and Boyer-Moore algorithms.

---

[2] An algorithm is called non-deterministic if the order in which its statements are executed is not fixed.

[3] The **for-do-rof** statement is taken from [vdE92]. Statement **for** $x : P$ **do** S **od** amounts to executing statement list S once for each value of x that satisfies P initially. The order in which the values of x are chosen is arbitrary.

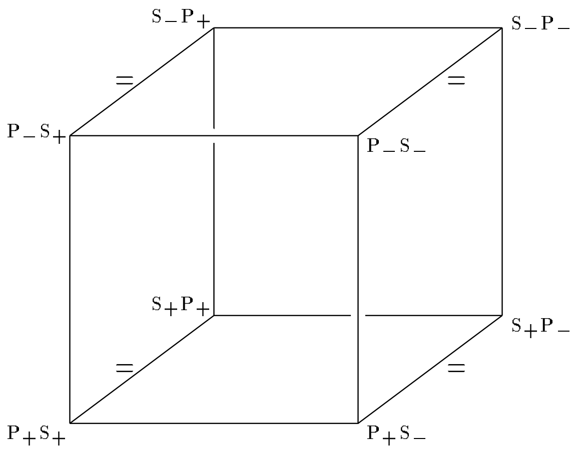

Figure 2: The 3-cube of naive pattern matching algorithms.

2.1 The (P_+) algorithms

The (P) algorithm presented in the previous section can be made deterministic by considering prefixes of S in order of increasing length. The outer union quantifier in the required value of O can be computed with a deterministic repetition. Instantiating the algorithm in Appendix A.2 with $W = V^* \times V^*$, $RANGE(u,r) \equiv ur = S$, $(u_0, r_0) \leq (u_1, r_1) \equiv u_0 \leq_p u_1$, $\oplus = \cup$, and $f(u,r) = (\cup\, l, v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})$ results in algorithm $(\mathrm{P}_+)^4$:

Algorithm 2.3(P_+)

$u, r := \varepsilon, S; O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \longrightarrow$
    $u, r := u(r{\upharpoonleft}1), r{\downharpoonleft}1;$
    $O := O \cup (\cup\, l, v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})$
od $\{R\}$

This algorithm will be used in section 3 as a starting point for the Aho-Corasick and Knuth-Morris-Pratt algorithms. The inner union quantification in the required value of O can be computed with a non-deterministic repetition as outlined in Appendix A.1. This algorithm is called ($\mathrm{P}_+\mathrm{S}$) but will not be given here.

2.1.1 The ($\mathrm{P}_+\mathrm{S}_+$) algorithms

Starting with algorithm ($\mathrm{P}_+\mathrm{S}$) we make its inner repetition deterministic by considering suffixes of u in order of increasing length. In keeping with the form in Appendix A.2, a first such algorithm is

---

[4] The operators $\upharpoonleft$, $\downharpoonleft$, $\upharpoonright$, and $\downharpoonright$ are defined in definition B.6

Algorithm 2.4(P$_+$S$_+$)

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

This algorithm has running time $\mathcal{O}(|S|^2)$, assuming that intersection with P is a $\mathcal{O}(1)$ operation. We will now improve the running time of this algorithm. Note that

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

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

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

Observe that $v \in \mathbf{suff}(P)$ is an invariant of the inner repetition. This invariant is initially established by the assignment $v := \varepsilon$. Direct evaluation of $(l{\upharpoonright}1)v \in \mathbf{suff}(P)$ is expensive. Therefore it is done using the transition function of the reverse trie [Fre60] corresponding to P $\tau_{P,r} : \mathbf{suff}(P) \times V \longrightarrow \mathbf{suff}(P) \cup \{\perp\}$ defined by

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

Since we usually refer the trie corresponding to P we will write τ_r instead of $\tau_{P,r}$. Transition function τ_r can be computed beforehand. The guard becomes $l \neq \varepsilon$ **cand** $\tau_r(v, l{\upharpoonright}1) \neq \perp$. This amounts to the introduction of algorithm detail

Detail (RT)**:** Usage of reverse trie function τ_r to implement expression $(l{\upharpoonright}1)v \in \mathbf{suff}(P)$. □

and yields

Algorithm 2.5(P$_+$S$_+$, RT)

$u, r := \varepsilon, S; O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\};$
do $r \neq \varepsilon \longrightarrow$
  $u, r := u(r{\upharpoonleft}1), r{\downharpoonright}1;$
  $l, v := u, \varepsilon; O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\};$
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l{\upharpoonright}1) \neq \perp \longrightarrow$
    $l, v := l{\downharpoonleft}1, (l{\upharpoonright}1)v;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od
  $\{v \in \mathbf{suff}(P) \wedge (l = \varepsilon$ **cor** $(l{\upharpoonright}1)v \notin \mathbf{suff}(P))\}$
od $\{R\}$

This algorithm has $\mathcal{O}(|S| \cdot (\mathbf{MAX}\, p : p \in P : |p|))$ running time. The precomputation of τ_r is similar to the precomputation of the transition function of the forward trie τ_f (defined in 2.2.1) which is discussed in Part II, section 6.

2.1.2 The ($\mathrm{P}_+\mathrm{S}_-$) algorithm

In the previous section we modified the inner repetition of algorithm ($\mathrm{P}_+\mathrm{S}$) to consider suffixes of u in order of increasing length. In this section, we will make use of an inner repetition which considers them in order of decreasing length. The general form of such a repetition is given in Appendix A.3. This gives us the following

Algorithm 2.6($\mathrm{P}_+\mathrm{S}_-$)

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

This algorithm has running time that is $\mathcal{O}(|S|^2)$.

2.2 The (S_-) algorithms

Algorithm (S) can be made deterministic by considering suffixes of S in order of decreasing length. Instantiating the algorithm in Appendix A.3 with $W = V^* \times V^*$, $RANGE(l, w) \equiv lw = S$, $(l_0, w_0) \leq (l_1, w_1) \equiv w_0 \leq_s w_1$, $\oplus = \cup$, and $f(l, w) = (\cup\, v, r : vr = w : \{l\} \times (\{v\} \cap P) \times \{r\})$ results in the deterministic algorithm (S_-) which will not be given here. Furthermore, the assignment to O in the repetition can be written as a non-deterministic repetition (see Appendix A.1 and also section 2.1) to give the algorithm ($\mathrm{S}_-\mathrm{P}$) which will not be given here.

2.2.1 The ($\mathrm{S}_-\mathrm{P}_+$) algorithms

Starting with algorithm ($\mathrm{S}_-\mathrm{P}$) we make the inner repetition deterministic by considering prefixes of each suffix of the input string in order of increasing length, in keeping with the algorithm in Appendix A.2. The algorithm is:

Algorithm 2.7($\mathrm{S}_-\mathrm{P}_+$)

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

This algorithm has $\mathcal{O}(|S|^2)$ running time like algorithm 2.4($\mathrm{P}_+\mathrm{S}_+$). In a manner similar to the introduction of the reverse trie, in algorithm 2.4($\mathrm{P}_+\mathrm{S}_+$), we can strengthen the inner repetition guard. Note that

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

So we can strengthen the guard of the inner repetition to $r \neq \varepsilon$ **cand** $v(r\upharpoonright 1) \in \mathbf{pref}(P)$. Conjunct $v \in \mathbf{pref}(P)$ can be added to the invariant of this repetition. It is initially established by the assignment $v := \varepsilon$. Efficient computation of this guard can be done by using the transition function of the forward trie corresponding to P $\tau_f : \mathbf{pref}(P) \times V \longrightarrow (\mathbf{pref}(P) \cup \{\bot\})$, defined by

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

Transition function τ_f can be computed beforehand. The guard now becomes

$$r \neq \varepsilon \ \mathbf{cand}\ \tau_f(v, r\upharpoonright 1) \neq \bot.$$

Detail (FT)**:** Usage of forward trie function τ_f to implement expression $v(r\upharpoonright 1) \in \mathbf{pref}(P)$. □

The forward trie detail (FT) is defined and used symmetrically to the reverse trie detail (RT). Introducing algorithm detail (FT) yields

Algorithm 2.8(S_P+, FT)

$l, w := \varepsilon, S; O := \varnothing;$
do $w \neq \varepsilon \longrightarrow$
  $v, r := \varepsilon, w; O := O \cup \{l\} \times (\{\varepsilon\} \cap P) \times \{w\};$
  do $r \neq \varepsilon$ **cand** $\tau_f(v, r\upharpoonright 1) \neq \bot \longrightarrow$
    $v, r := v(r\upharpoonright 1), r\downharpoonright 1;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od;
  $l, w := l(w\upharpoonright 1), w\downharpoonright 1$
od;
$O := O \cup \{S\} \times (\{\varepsilon\} \cap P) \times \{\varepsilon\}$
$\{R\}$

This algorithm has $\mathcal{O}(|S| \cdot (\mathbf{MAX}\ p : p \in P : |p|))$ running time like algorithm 2.5(P+S+, RT).

2.2.2 The (S_P_) algorithm

The inner repetition of algorithm (S_P) can also be made deterministic by considering prefixes of w in order of decreasing length, as in Appendix A.3. This yields algorithm (S_P_) which is not given here. Its running time is $\mathcal{O}(|S|^2)$.

3 The Aho-Corasick algorithms

In this section, starting with algorithm 2.3(P+), we derive the Aho-Corasick and Knuth-Morris-Pratt algorithms. First, we make a preliminary step. The triple format of O used so far has been redundant. This redundancy can be removed by registering matches in S by their end-point; that is, the first component of the triple will be dropped. This modification is known as algorithm detail (E).

Detail (E)**:** Matches are registered by their end-point. □

In the following derivation we use the symbol $\simeq$ to indicate that the problem specification has been specialized (in this case, through projection). The postcondition of algorithm 2.3(P+) can be rewritten as follows:

$$
\begin{array}{ll}
& (\cup\, u,r : ur = S : (\cup\, l,v : lv = u : \{l\} \times (\{v\} \cap P) \times \{r\})) \\
\simeq & \quad \{\text{introduction of detail } (\mathrm{E})\} \\
& (\cup\, u,r : ur = S : (\cup\, l,v : lv = u : (\{v\} \cap P) \times \{r\})) \\
= & \quad \{\text{definition of } \mathbf{suff}\text{, distributivity}\} \\
& (\cup\, u,r : ur = S : (\mathbf{suff}(u) \cap P) \times \{r\})
\end{array}
$$

This yields a new postcondition

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

which is established by a modified version of algorithm 2.3(P_+)

Algorithm 3.1(P_+, E)

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

In the following sections, algorithm details unique to the Aho-Corasick and Knuth-Morris-Pratt algorithms will be introduced.

3.1 Algorithm detail AC

In order to facilitate the update of O_e in algorithm 3.1(P_+, E) we introduce a new variable U and attempt to maintain invariant $U = \mathbf{suff}(u) \cap P$. For the update of U we derive

$$
\begin{array}{ll}
& \mathbf{suff}(ua) \cap P \\
= & \quad \{\, \mathbf{suff}(ua) = \mathbf{suff}(u)a \cup \{\varepsilon\} \,\} \\
& (\mathbf{suff}(u)a \cap P) \cup (\{\varepsilon\} \cap P) \\
= & \quad \{\, \mathbf{suff}(u)a \cap P \subseteq \mathbf{pref}(P)a \,\} \\
& ((\mathbf{suff}(u) \cap \mathbf{pref}(P))a \cap P) \cup (\{\varepsilon\} \cap P)
\end{array}
$$

Therefore, in order to calculate the new of value of U we need the set $\mathbf{suff}(u) \cap \mathbf{pref}(P)$ rather than the old value of U ($\mathbf{suff}(u) \cap P$). Formula $\mathbf{suff}(u) \cap \mathbf{pref}(P)$ can be viewed as a generalization of formula $\mathbf{suff}(u) \cap P$. Hence, we try to maintain invariant

$$P_0(u, U) \equiv (U = \mathbf{suff}(u) \cap \mathbf{pref}(P))$$

which is initially established by assignment $u, U := \varepsilon, \{\varepsilon\}$. Assuming $P_0(u, U)$ we derive

$$
\begin{array}{ll}
& \mathbf{suff}(ua) \cap \mathbf{pref}(P) \\
= & \quad \{\text{preceding derivation with } \mathbf{pref}(P) \text{ instead of } P,\ \mathbf{pref} \text{ is idempotent}^5\} \\
& ((\mathbf{suff}(u) \cap \mathbf{pref}(P))a \cap \mathbf{pref}(P)) \cup (\{\varepsilon\} \cap \mathbf{pref}(P)) \\
= & \quad \{\, P_0(u,U),\ P \neq \varnothing \,\} \\
& (Ua \cap \mathbf{pref}(P)) \cup \{\varepsilon\}\ .
\end{array}
$$

From $P_0(u, U)$ and $P \subseteq \mathbf{pref}(P)$ it follows that $\mathbf{suff}(u) \cap P = U \cap P$. This all leads to the following modification of algorithm 3.1(P_+, E):

---

[5] A function f is called idempotent if $f \circ f = f$.

$u, r := \varepsilon, S;\ U := \{\varepsilon\};\ O_e := (\{\varepsilon\} \cap P) \times \{S\};$
$\{$invariant: $P_0(u, U)\}$
do $r \neq \varepsilon \longrightarrow$
  $U := (U(r\upharpoonright 1) \cap \mathbf{pref}(P)) \cup \{\varepsilon\}; \quad \{P_0(u(r\upharpoonright 1), U)\}$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1; \quad \{P_0(u, U)\}$
  $O_e := O_e \cup (U \cap P) \times \{r\}$
od $\{R_e\}$

Since S and, therefore, u can be any string from V^* it follows from invariant $P_0(u, U)$ that the values that U can have constitute the finite set $\{\mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^*\}$. Hence, the preceding algorithm can be viewed as simulating the behavior of Moore machine[HU79] (or finite transducer) $M_0 = (Q_0, \Sigma_0, \Delta_0, \delta_0, \lambda_0, s_0)$ on input string S, where

- state set $Q_0 = \{\mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^*\}$,
- input alphabet $\Sigma_0 = V$,
- output alphabet $\Delta_0 = \mathcal{P}(P)$,
- transition function $\delta_0 : Q_0 \times V \longrightarrow Q_0$ is defined by

$$\delta_0(q, a) = (qa \cap \mathbf{pref}(P)) \cup \{\varepsilon\} \qquad (q \in Q_0, a \in \Sigma_0),$$

- output function $\lambda_0 : Q_0 \longrightarrow \Delta_0$ is defined by

$$\lambda_0(q) = q \cap P \qquad (q \in Q_0),$$

  and
- start state $s_0 = \{\varepsilon\}$.

Moore machine M_0 can be viewed as a deterministic finite automaton without final states and with an additional output alphabet Δ_0 and an additional output function λ_0. If on reading input sequence w machine M_0 goes through states $s_0, q_1, q_2, ..., q_{|w|}$ it will emit output sequence $\lambda_0(s_0)\lambda_0(q_1)\lambda_0(q_2)...\lambda_0(q_{|w|})$. The set O_e can be viewed as an encoding of the output sequence of Moore machine M_0.

The following intermezzo shows that Moore machine M_0 can be obtained in a different way.

> An interesting solution to the pattern matching problem involves using an automaton for the language V^*P. Usually, a nondeterministic finite automaton (NFA) is constructed. The automaton is then simulated, processing input string S, and considering all paths through the automaton. Whenever a final state is entered a keyword match has been found, and the match is registered; see for example Aho, Hopcroft & Ullman in [AHU74].
>
> The state graph for the NFA is simply the forward trie for P, augmented with a transition from state ε to itself on all symbols in V. The NFA is defined as $(Q_N, V, \delta_N, s_N, F_N)$, where
>
> - state set $Q_N = \mathbf{pref}(P)$,
> - input alphabet V,
> - transition function $\delta_N : Q_N \times V \longrightarrow \mathcal{P}(Q_N)$ is defined by
>
> $$\delta_N(\varepsilon, a) = \begin{cases} \{\varepsilon, a\} & \text{if } a \in \mathbf{pref}(P) \\ \{\varepsilon\} & \text{otherwise} \end{cases} \qquad (a \in V),$$
>
>   and
>
> $$\delta_N(q, a) = \begin{cases} \{qa\} & \text{if } qa \in \mathbf{pref}(P) \\ \varnothing & \text{otherwise} \end{cases} \qquad (q \in \mathbf{pref}(P) \setminus \{\varepsilon\}, a \in V).$$
>
>   and is extended to $\delta_N^* : Q_N \times V^* \longrightarrow \mathcal{P}(Q_N)$ in the obvious way,

- start state $s_N = \varepsilon$, and
- final state set $F_N = P$.

The simulation of this automaton can proceed as follows:

$$
\begin{array}{l}
u, r := \varepsilon, S; q_N := \{\varepsilon\};\\
O_e := (q_N \cap F_N) \times \{r\};\\
\{\text{invariant: } q_N = \delta_N^*(\varepsilon, u)\}\\
\mathbf{do}\ r \neq \varepsilon \longrightarrow\\
\qquad q_N := (\cup\ q : q \in q_N : \delta_N(q, r\upharpoonright 1));\\
\qquad O_e := O_e \cup (q_N \cap F_N) \times \{r\}\\
\mathbf{od}\ \{R_e\}
\end{array}
$$

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

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

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

The subset construction (see [RS59]) can be applied to the nondeterministic Moore machine, to give a deterministic Moore machine M_D. In the following paragraphs, we will prove that this deterministic Moore machine (with unreachable states removed) is equal to M_0 (presented above).

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

$$
\begin{array}{cl}
 & Q_D \\
= & \qquad \{\text{subset construction and reachability}\} \\
 & \{\delta_N^*(\varepsilon, w) \mid w \in V^*\} \\
= & \qquad \{\text{definition of } \delta_N\} \\
 & \{\{q \mid q \in \mathbf{pref}(P) \wedge w \in V^*q\} \mid w \in V^*\} \\
= & \qquad \{w \in V^*q \equiv q \in \mathbf{suff}(w)\} \\
 & \{\mathbf{suff}(w) \cap \mathbf{pref}(P) \mid w \in V^*\} \\
= & \qquad \{\text{definition of } Q_0\} \\
 & Q_0
\end{array}
$$

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

$$
\begin{array}{cl}
 & \lambda_D(q) \\
= & \qquad \{\text{subset construction}\} \\
 & \{\lambda_N(q') \mid q' \in q \wedge \lambda_N(q') \neq \perp_N\} \\
= & \qquad \{\text{definition of } \lambda_N\} \\
 & \{q' \mid q' \in q \wedge q' \in P\} \\
= & \qquad \{\text{set calculus}\} \\
 & q \cap P \\
= & \qquad \{\text{definition } \lambda_0\} \\
 & \lambda_0(q)
\end{array}
$$

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

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

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

Notice that the number of states of the Moore machine does not grow during the subset construction. Perrin mentions the AC and KMP Moore machines as examples of ones which do not suffer from exponential blowup (i.e. the number of states grows exponentially) during the subset construction [Per90].

In subsection 3.2 it is shown that Moore machine M_0 is minimal.

We proceed by observing that for each $v \in V^*$ the set $\mathbf{suff}(v) \cap \mathbf{pref}(P)$ is nonempty, finite, and linearly ordered with respect to the suffix ordering $\leq_s$ (see Definition B.4) and therefore has a maximal element $(\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(v) \cap \mathbf{pref}(P) : w)$. Since **suff** is idempotent $(\mathbf{suff}(\mathbf{suff}(u)) = \mathbf{suff}(u))$ we have by theorem B.5

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

so the states of machine M_0 can be represented by their maximal elements. We replace variable U in the algorithm by variable q and maintain invariant

$$P_0'(u, q) \equiv (q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w)).$$

Introduction of q is called algorithm detail (AC).

Detail (AC): A variable q is introduced into algorithm 3.1(P$_+$, E) such that

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

□

We now have that $\mathbf{suff}(u) \cap P = \mathbf{suff}(q) \cap P$. By introducing function $Output : \mathbf{pref}(P) \longrightarrow \mathcal{P}(P)$, defined by

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

the update of O_e can be done by assignment $O_e := O_e \cup Output(q) \times \{r\}$. The precomputation of function $Output$ is discussed in Part II, section 6.

We now have obtained algorithm

Algorithm 3.2(P$_+$, E, AC)

$u, r := \varepsilon, S; q := \varepsilon; O_e := Output(q) \times \{S\};$
$\{\text{invariant: } P_0'(u, q)\}$
do $r \neq \varepsilon \longrightarrow$
  $q := (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u(r\upharpoonright 1)) \cap \mathbf{pref}(P) : w);$ $\{P_0'(u(r\upharpoonright 1), q)\}$
  $u, r := u(r\upharpoonright 1), r\downharpoonright 1;$ $\{P_0'(u, q)\}$
  $O_e := O_e \cup Output(q) \times \{r\}$
od $\{R_e\}$

The next two sections are concerned with alternative ways of implementing assignment

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

3.2 Method OPT

Assuming $P_0'(u,q)$ we derive

$$
\begin{array}{ll}
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(ua) \cap \mathbf{pref}(P) : w) \\
= & \quad \{\, \mathbf{suff}(ua) = \mathbf{suff}(u)a \cup \{\varepsilon\},\ P \neq \varnothing \,\} \\
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u)a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{\, \text{theorem B.5} \,\} \\
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}((\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(u) \cap \mathbf{pref}(P) : w'))a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{\, P_0'(u,q) \,\} \\
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q)a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{\, \mathbf{suff}(qa) = \mathbf{suff}(q)a \cup \{\varepsilon\},\ P \neq \varnothing \,\} \\
 & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(qa) \cap \mathbf{pref}(P) : w)
\end{array}
$$

By introducing function[6] $\gamma_f : \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P)$, defined by

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

the assignment to q in algorithm 3.2(P$_+$, E, AC) can be written as $q := \gamma_f(q,a)$. This is called algorithm

Detail (OPT)**:** Usage of function γ_f to update variable q. □

and leads to algorithm (cf. [AC75], section 6)

Algorithm 3.3(P$_+$, E, AC, OPT)

$$
\begin{array}{l}
u, r := \varepsilon, S; q := \varepsilon; O_e := Output(q) \times \{S\}; \\
\{\text{invariant: } P_0'(u,q)\} \\
\mathbf{do}\ r \neq \varepsilon \longrightarrow \\
\qquad q := \gamma_f(q, r{\upharpoonright}1); \qquad \{P_0'(u(r{\upharpoonright}1), q)\} \\
\qquad u, r := u(r{\upharpoonright}1), r{\downharpoonright}1; \qquad \{P_0'(u,q)\} \\
\qquad O_e := O_e \cup Output(q) \times \{r\} \\
\mathbf{od}\ \{R_e\}
\end{array}
$$

Note that γ_f is the transition function of Moore machine $M_1 = (\mathbf{pref}(P), V, \mathcal{P}(P), \gamma_f, Output, \varepsilon)$. Machine M_1 is isomorphic with machine M_0 from section 3.1 since function $enc : Q_0 \longrightarrow \mathbf{pref}(P)$ defined by

$$enc(q) = (\mathbf{MAX}_{\leq_s}\, q' : q' \in q : q')$$

is bijective. Furthermore Moore machine M_1 corresponds to the automaton in the "optimized" version of the Aho-Corasick algorithm.

Another interesting property of the Moore machine M_1 is that it is in fact the minimal Moore machine for its language. This will be shown in the following intermezzo.

For deterministic Moore machines we use the following definition of minimality:

$$
\begin{array}{l}
Minimal(Q, V, \Sigma, \delta, \lambda, s) \equiv \\
\qquad (\forall q_0, q_1 : q_0 \neq q_1 \wedge q_0 \in Q \wedge q_1 \in Q : (\exists w : w \in V^* : \lambda(\delta^*(q_0, w)) \neq \lambda(\delta^*(q_1, w)))).
\end{array}
$$

Notice that this definition can be viewed as a generalization of the definition of minimality for deterministic finite automata (replace $\lambda(\delta^*(q,w))$ by $\delta^*(q,w) \in F$ in the definition where F is the set of final states of the finite automaton).

[6] Subscript f is used to indicate that γ_f corresponds to the forward trie transition function τ_f.

We now prove that the Moore machine M_1 is minimal by contradiction. Assume that there are

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

such that

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

Choose $w_0 : q_0 w_0 \in P$. Then $\gamma_f^*(q_0, w_0) = q_0 w_0$ and $q_0 w_0 \in Output(\gamma_f^*(q_0, w_0))$. In this case (from the assumptions)

$$\begin{array}{ll} & q_0 w_0 \in Output(\gamma_f^*(q_1, w_0)) \\ \Rightarrow & \quad \{ \text{definition of } \gamma_f \text{ and } Output \} \\ & q_0 w_0 \leq_s \gamma_f^*(q_1, w_0) \leq_s q_1 w_0 \\ \Rightarrow & \quad \{ \text{property of } \leq_s \} \\ & q_0 \leq_s q_1 \\ \Rightarrow & \quad \{ |q_0| \geq |q_1| \} \\ & q_0 = q_1 \end{array}$$

which is a contradiction. We conclude that Moore machine M_1 is minimal and end this intermezzo.

Provided evaluating $\gamma_f(q, a)$ and $Output(q)$ are $\mathcal{O}(1)$ operations (for instance, if γ_f and $Output$ are tabulated) algorithm 3.3(P$_+$, E, AC, OPT) has $\mathcal{O}(|S|)$ run time complexity. Precomputation of γ_f is discussed in Part II, section 6. It involves the so-called failure function which is introduced in the next subsection. Precomputation takes $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ time. Storage of γ_f and $Output$ takes $\mathcal{O}(|\mathbf{pref}(P)| \cdot |V|)$ space.

3.3 Linear search

In this subsection we give an alternative way of implementing assignment

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

involving linear search. We start with the following derivation, assuming $P_0'(u, q)$,

$$\begin{array}{ll} & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(ua) \cap \mathbf{pref}(P) : w) \\ = & \quad \{ \text{derivation in subsection 3.2 without last step} \} \\ & (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q)a \cap \mathbf{pref}(P) \lor w = \varepsilon : w) \\ = & \quad \{ \mathbf{suff}(q)a \cap \mathbf{pref}(P) \subseteq \mathbf{pref}(P)a \} \\ & (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(q) \cap \mathbf{pref}(P))a \cap \mathbf{pref}(P) \lor w = \varepsilon : w) \\ = & \quad \{ \text{domain split, introduction of } \bot_s \text{ with } \bot_s \,\mathbf{max}_{\leq_s}\, w = w \,\mathbf{max}_{\leq_s}\, \bot_s = w \\ & \quad\;\; \text{and } (\mathbf{MAX}_{\leq_s}\, w : w \in \varnothing : w) = \bot_s {}^{7} \} \\ & (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(q) \cap \mathbf{pref}(P))a \cap \mathbf{pref}(P) : w) \,\mathbf{max}_{\leq_s}\, \varepsilon \\ = & \quad \{ \text{change of bound variable: } w = w'a \} \\ & (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \land w'a \in \mathbf{pref}(P) : w'a) \,\mathbf{max}_{\leq_s}\, \varepsilon \\ = & \quad \{ \text{additional requirement on } \bot_s : \bot_s w = w \bot_s = \bot_s \ (\bot_s \text{ is zero of concatenation}^7) \} \\ & (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \land w'a \in \mathbf{pref}(P) : w')a \,\mathbf{max}_{\leq_s}\, \varepsilon \end{array}$$

In order to compute the value of the quantified subexpression in the last expression of the derivation we use a linear search on $\mathbf{suff}(q) \cap \mathbf{pref}(P)$. This is called algorithm detail

Detail (LS): Using linear search to update the state variable q. □

In the next two subsections we present two slightly different methods of linear search; the first leads to the standard Aho-Corasick algorithm, the second to the Knuth-Morris-Pratt algorithm.

[7] Like q representing $\mathbf{suff}(q) \cap \mathbf{pref}(P)$, $\bot_s$ can be thought of as representing the empty set, provided we extend the definition of **suff** with $\mathbf{suff}(\bot_s) = \varnothing$.

3.3.1 The Aho-Corasick algorithm with failure function

Given a linearly ordered, non-empty, and finite set W we can define predecessor function $pred : W \setminus \{\mathbf{min}(W)\} \longrightarrow W \setminus \{\mathbf{max}(W)\}$ by

$$pred(w) = (\mathbf{MAX}\, w' : w' \in W \wedge w' < w : w') \qquad (w \in W \setminus \{\mathbf{min}(W)\}).$$

Given a predicate $B : W \longrightarrow \mathbb{B}$ linear search for the maximal element of W satisfying B can proceed as follows:

$w := \mathbf{max}(W)$;
$\mathbf{do}\ w \neq \mathbf{min}(W) \wedge \neg B(w) \longrightarrow w := pred(w)\ \mathbf{od}$
$\{(w = \mathbf{min}(W) \wedge \neg(\exists w' \in W :: B(w'))) \vee w = (\mathbf{MAX}\, w' \in W : B(w') : w')\}$

Taking

- $W = \mathbf{suff}(q) \cap \mathbf{pref}(P)$ (linearly ordered under $\leq_s$, $\mathbf{max}(W) = q$ $(P_0'(u,q))$, $\mathbf{min}(W) = \varepsilon$),
- $pred = f_{f|_{\mathbf{suff}(q)\cap\mathbf{pref}(P)}}$[8] where $f_f : \mathbf{pref}(P) \setminus \{\varepsilon\} \longrightarrow \mathbf{pref}(P)$ is defined by

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

  (function f_f is the Aho-Corasick failure function corresponding to the forward trie [AC75]), and
- $B(w) \equiv wa \in \mathbf{pref}(P) \qquad (w \in \mathbf{pref}(P))$

leads to the following update of variable q

$\{P_0'(u,q)\}$
$q' := q$;
$\mathbf{do}\ q' \neq \varepsilon \wedge q'a \notin \mathbf{pref}(P) \longrightarrow q' := f_f(q')\ \mathbf{od}$;
$\{(q' = \varepsilon \wedge \neg(\exists w : w \in \mathbf{suff}(q) \cap \mathbf{pref}(P) : wa \in \mathbf{pref}(P)))$
$\ \vee q' = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge wa \in \mathbf{pref}(P) : w)\}$
$\mathbf{if}\ q' = \varepsilon \wedge a \notin \mathbf{pref}(P) \longrightarrow q := \varepsilon$
$[\!]\ q' \neq \varepsilon \vee a \in \mathbf{pref}(P) \longrightarrow q := q'a$
$\mathbf{fi}\ \{P_0'(ua,q)\}$

The second conjunct in the guard of the repetition can be evaluated using the forward trie τ_f $(q'a \notin \mathbf{pref}(P) \equiv \tau_f(q',a) = \bot)$. However, by introducing the extended forward trie $\tau_{ef} : \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P) \cup \{\bot_s\}$ defined by

$$\tau_{ef}(w,c) = \begin{cases} wc & \text{if } wc \in \mathbf{pref}(P) \\ \varepsilon & \text{if } w = \varepsilon \wedge c \notin \mathbf{pref}(P) \\ \bot_s & \text{otherwise} \end{cases} \qquad (w \in \mathbf{pref}(P), c \in V)$$

both conjuncts can be combined:

$$q' \neq \varepsilon \wedge q'a \notin \mathbf{pref}(P) \equiv \tau_{ef}(q',a) = \bot_s.$$

As a side effect of this introduction the **if-fi** statement can be replaced by the single assignment statement $q := \tau_{ef}(q',a)$.

[8] With $f_{|A}$ we denote the function that is equal to f with its domain restricted to set A.

By adding algorithm detail

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

and eliminating variable q' we obtain algorithm (cf. [AC75], section 2, algorithm 1)

Algorithm 3.4(P$_+$, E, AC, LS, AC-FAIL)

$u, r := \varepsilon, S$; $q := \varepsilon$; $O_e := Output(q) \times \{S\}$;
$\{$invariant: $P_0'(u, q)\}$
do $r \neq \varepsilon \longrightarrow$
  do $\tau_{ef}(q, r{\upharpoonright}1) = \perp_s \longrightarrow q := f_f(q)$ **od**;
  $q := \tau_{ef}(q, r{\upharpoonright}1)$; $\{P_0'(u(r{\upharpoonright}1), q)\}$
  $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1$; $\{P_0'(u, q)\}$
  $O_e := O_e \cup Output(q) \times \{r\}$
od $\{R_e\}$

This algorithm still has $\mathcal{O}(|S|)$ run time complexity [Aho90] but is less efficient than the algorithm 3.3(P$_+$, E, AC, OPT) in section 3.2. Function τ_{ef} can be stored more efficiently than function γ_f by use of a default value ($\perp_s$) requiring $\mathcal{O}(|\mathbf{pref}(P)|)$ space. Precomputation of extended forward trie τ_{ef} and failure function f_f is discussed in Part II, section 6.

3.3.2 The Knuth-Morris-Pratt algorithm

We now derive the Knuth-Morris-Pratt (KMP) algorithm, using a type of linear search different from that used for the Aho-Corasick algorithm with failure function.

As in the previous subsection we define a predecessor function on a totally ordered set W. In this case, however, we have a total predecessor function $pred_{ext} : W \longrightarrow (W \setminus \{\mathbf{max}(W)\}) \cup \{\perp_W\}$ defined by

$$pred_{ext}(w) = \begin{cases} pred(w) & \text{if } w \neq \mathbf{min}(W) \\ \perp_W & \text{if } w = \mathbf{min}(W) \end{cases} \qquad (w \in W)$$

where $\perp_W$ is such that $\perp_W \mathbf{max}_\leq w = w \,\mathbf{max}_\leq \perp_W = w$ and $(\mathbf{MAX}_\leq\, w \in W : w \in \varnothing : w) = \perp_W$.

Assuming a selection predicate B as in the previous section, linear search can proceed as follows:

$w := \mathbf{max}(W)$;
do $w \neq \perp_W$ **cand** $\neg B(w) \longrightarrow w := pred_{ext}(w)$ **od**
$\{w = (\mathbf{MAX}_\leq\, w' \in W : B(w') : w')\}$

Taking $W = \mathbf{suff}(q) \cap \mathbf{pref}(P)$, $\perp_W = \perp_s$ (remember that $\perp_s$ is also defined to be a zero of concatenation), $pred_{ext} = f_{ef}{}_{|\mathbf{suff}(q)\cap\mathbf{pref}(P)}$ where $f_{ef} : \mathbf{pref}(P) \longrightarrow \mathbf{pref}(P) \cup \{\perp_s\}$ is defined by

$$f_{ef}(w) = \begin{cases} f(w) & \text{if } w \neq \varepsilon \\ \perp_s & \text{if } w = \varepsilon \end{cases} \qquad (w \in \mathbf{pref}(P))$$

(f_{ef} is called the *extended failure function* corresponding to the forward trie), and $B(w) \equiv wa \in \mathbf{pref}(P)$ leads to the following instantiation of the linear search:

$\{P_0'(u,q)\}$
$q' := q;$
$\mathbf{do}\ q' \neq \bot_s\ \mathbf{cand}\ q'a \notin \mathbf{pref}(P) \longrightarrow q' := f_{ef}(q')\ \mathbf{od};$
$\{q' = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge wa \in \mathbf{pref}(P) : w)\}$
$q := q'a\,\mathbf{max}_{\leq_s}\,\varepsilon$
$\{q = (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(q) \cap \mathbf{pref}(P) \wedge wa \in \mathbf{pref}(P) : w)a\,\mathbf{max}_{\leq_s}\,\varepsilon\}$
$\{P_0'(ua,q)\}$

Adding the algorithm detail

Detail (KMP-FAIL)**:** The extended failure function f_{ef} is introduced to implement the linear search for the update of q. □

and eliminating variable q' leads to algorithm

Algorithm 3.5(P$_+$, E, AC, LS, KMP-FAIL)

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
$\{\text{invariant: } P_0'(u,q)\}$
$\mathbf{do}\ r \neq \varepsilon \longrightarrow$
$\quad \mathbf{do}\ q \neq \bot_s\ \mathbf{cand}\ q(r{\upharpoonright}1) \notin \mathbf{pref}(P) \longrightarrow q := f_{ef}(q)\ \mathbf{od};$
$\quad q := q(r{\upharpoonright}1)\,\mathbf{max}_{\leq_s}\,\varepsilon; \qquad \{P_0'(u(r{\upharpoonright}1), q)\}$
$\quad u, r := u(r{\upharpoonright}1), r{\downharpoonright}1; \qquad \{P_0'(u,q)\}$
$\quad O_e := O_e \cup Output(q) \times \{r\}$
$\mathbf{od}\ \{R_e\}$

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

Detail (OKW)**:** $P = \{p\}$ □

We now introduce three shadow variables, and invariants that are maintained between the shadow variables and the existing program variables. Most shadow predicates and functions will be "hatted" for easy identification. Variables i and j are so named to conform to the original publication of the algorithms.

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

We must of course define new predicates and a new predecessor function $\hat{f}_{ef}$ on these shadow variables.

- Define $\hat{f}_{ef} : [1, |p|+1] \longrightarrow [0, |p|]$ as $\hat{f}_{ef}(i) = |f_{ef}(p_1 \dots p_{i-1})| + 1$ and define $|\bot_s| = -1$.
- $\hat{P}_0'(j,i) \equiv (p_1 \dots p_{i-1} = (\mathbf{MAX}_{\leq_s}\, w \in V^* : w \in \mathbf{suff}(S_1 \dots S_{j-1}) \cap \mathbf{pref}(p) : w))$.

- $\hat{R}_e \equiv (\hat{O}_e = (\cup\, j : 1 \le j \le |S|+1 \wedge p \in \mathbf{suff}(S_1 \cdots S_{j-1}) : \{j\}))$

We can also note the following equivalences and correspondences:

- Since $q \in \mathbf{pref}(p)$ then $q(r\upharpoonright 1) \notin \mathbf{pref}(p) \equiv S_j \ne p_i$ when $i \le |p| \wedge j \le |S|$. Similarly $q \ne \perp_s \equiv 0 < i$ and $q = p \equiv i = |p|+1$.
- $q := q(r\upharpoonright 1)\,\mathbf{max}_{\le_s}\,\varepsilon$ corresponds to $i := i+1$
- $u, r := u(r\upharpoonright 1), r\downharpoonright 1$ corresponds to $j := j+1$
- $r \ne \varepsilon \equiv j \le |S|$
- $O_e := O_e \cup Output(q) \times \{r\}$ corresponds to $\mathbf{if}\ |p| < i \longrightarrow \hat{O}_e := \hat{O}_e \cup \{j\} [\!] \ i \le |p| \longrightarrow \mathbf{skip\ fi}$

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

$u, r := \varepsilon, S;\ q := \varepsilon;\ O_e := Output(q) \times \{S\};$
$i := 1; j := 1;$
$\mathbf{if}\ i = |p|+1 \longrightarrow \hat{O}_e := \{j\} [\!]\ i \ne |p|+1 \longrightarrow \hat{O}_e := \varnothing\ \mathbf{fi};$
$\{\text{invariant: } P_0'(u,q) \wedge \hat{P}_0'(j,i)\}$
$\mathbf{do}\ j \le |S| \longrightarrow$
$\quad \mathbf{do}\ 0 < i\ \mathbf{cand}\ S_j \ne p_i \longrightarrow q := f_{ef}(q); i := \hat{f}_{ef}(i)\ \mathbf{od};$
$\quad q := q(r\upharpoonright 1)\,\mathbf{max}_{\le_s}\,\varepsilon;\ i := i+1; \qquad \{P_0'(u(r\upharpoonright 1), q) \wedge \hat{P}_0'(j+1, i)\}$
$\quad u, r := u(r\upharpoonright 1), r\downharpoonright 1;\ j := j+1; \qquad \{P_0'(u,q) \wedge \hat{P}_0'(j,i)\}$
$\quad O_e := O_e \cup Output(q) \times \{r\};$
$\quad \mathbf{if}\ i = |p|+1 \longrightarrow \hat{O}_e := \hat{O}_e \cup \{j\}$
$\quad [\!]\ \ i \ne |p|+1 \longrightarrow \mathbf{skip}$
$\quad \mathbf{fi}$
$\mathbf{od}\ \{R_e \wedge \hat{R}_e\}$

We have introduced algorithm detail:

Detail (INDICES)**:** Represent substrings by indices into the complete strings. □

Removing the non-shadow variables leaves us with the classic KMP algorithm (cf. [KMP77], section 2, p.326):

Algorithm 3.6(P_+, E, AC, LS, KMP-FAIL, OKW, INDICES)

$i := 1; j := 1;$
$\mathbf{if}\ i = |p|+1 \longrightarrow \hat{O}_e := \{j\} [\!]\ i \ne |p|+1 \longrightarrow \hat{O}_e := \varnothing\ \mathbf{fi};$
$\{\text{invariant: } \hat{P}_0'(j,i)\}$
$\mathbf{do}\ j \le |S| \longrightarrow$
$\quad \mathbf{do}\ 0 < i\ \mathbf{cand}\ S_j \ne p_i \longrightarrow i := \hat{f}_{ef}(i)\ \mathbf{od};$
$\quad i := i+1; \qquad \{\hat{P}_0'(j+1, i)\}$
$\quad j := j+1; \qquad \{\hat{P}_0'(j,i)\}$
$\quad \mathbf{if}\ i = |p|+1 \longrightarrow \hat{O}_e := \hat{O}_e \cup \{j\}$
$\quad [\!]\ \ i \ne |p|+1 \longrightarrow \mathbf{skip}$
$\quad \mathbf{fi}$
$\mathbf{od}\ \{\hat{R}_e\}$

This algorithm has $\mathcal{O}(|S|)$ running time. Storage of $\hat{f}_{ef}$ requires $\mathcal{O}(|p|)$ space. Precomputation of function $\hat{f}_{ef}$ can easily be derived by converting, in a similar way, the precomputation of function f_{ef} (as discussed in Part II, section 6) into using indices.

4 The Commentz-Walter algorithms

We now consider a derivation of the Commentz-Walter algorithms starting with algorithm 2.5 (P$_+$S$_+$, RT). We will be exploring the possibility of (safely) making shifts of more than one symbol.

To present an algorithm more closely matched to the one presented by Commentz-Walter we add the problem detail

Detail (NE)**:** $\varepsilon \notin P$ □

Consequently, assignments $O := \{\varepsilon\} \times (\{\varepsilon\} \cap P) \times \{S\}$ and $O := O \cup \{u\} \times (\{\varepsilon\} \cap P) \times \{r\}$ become superfluous in algorithm 2.5(P$_+$S$_+$, RT). Our goal is to make shifts larger than one symbol in the assignment $u, r := u(r\upharpoonleft 1), r\downharpoonleft 1$. In order to do this, an attempted match should occur before this assignment. In this case, information obtained during the attempted match can be used to determine an appropriate shift. Attempted matches are performed by the inner repetition of the algorithm. A phase shift of the outer repetition will place the inner repetition immediately before the shift assignment. Such a phase shift also places an extra copy of the inner repetition after the outer repetition. Let $m = (\mathbf{MIN}\ p \in P :: |p|)$. Since $|u| < m \Rightarrow \mathbf{suff}(u) \cap P = \varnothing$ we also start with a different assignment to u, r. This phase shift and assignment to u, r are not considered algorithm details. This yields algorithm

Algorithm 4.1(P$_+$S$_+$, RT, NE)

$u, r := S\upharpoonleft m, S\downharpoonleft m; O := \varnothing;$
do $r \neq \varepsilon \longrightarrow$
  $l, v := u, \varepsilon;$
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l\upharpoonright 1) \neq \perp \longrightarrow$
    $l, v := l\downharpoonright 1, (l\upharpoonright 1)v;$
    $O := O \cup \{l\} \times (\{v\} \cap P) \times \{r\}$
  od;
  $\{v \in \mathbf{suff}(P) \wedge (l = \varepsilon\ \ \mathbf{cor}\ (l\upharpoonright 1)v \notin \mathbf{suff}(P))\}$
  $u, r := u(r\upharpoonleft 1), r\downharpoonleft 1$
od;
$l, v := S, \varepsilon;$
do $l \neq \varepsilon$ **cand** $\tau_r(v, l\upharpoonright 1) \neq \perp \longrightarrow$
  $l, v := l\downharpoonright 1, (l\upharpoonright 1)v;$
  $O := O \cup \{l\} \times (\{v\} \cap P) \times \{\varepsilon\}$
od
$\{v \in \mathbf{suff}(P) \wedge (l = \varepsilon\ \ \mathbf{cor}\ (l\upharpoonright 1)v \notin \mathbf{suff}(P))\}$
$\{R\}$

4.1 Larger shifts

We now consider larger shifts than in the assignment

$$u, r := u(r\upharpoonleft 1), r\downharpoonleft 1$$

in the previous algorithm.

Detail (CW)**:** If k is such that

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

then the assignment to u, r may be replaced by

$$u, r := u(r\upharpoonleft k), r\downharpoonleft k$$

without missing any matches. □

A number k satisfying the above condition is called a *safe shift distance*. Computing the upperbound on k (the maximal safe shift) is essentially the same as the problem that we are trying to solve. Therefore, we will aim at easier to compute approximations of the upperbound. By weakening the predicate $\mathbf{suff}(u(r\upharpoonleft n)) \cap P \neq \varnothing$ in the range of the quantified expression approximations of the upperbound from below are obtained.

This method of *predicate weakening* proves to be extremely important both in the derivation of the Commentz-Walter algorithm and the Boyer-Moore algorithm variants. In both cases the value of a quantified minimum must be computed. The range predicate in the quantifier is initially too strong, amounting to a problem of similar difficulty to the one which we are trying to solve. A weakening of this predicate will lead to a conservative approximation of the quantified minimum, with less computational effort.

In the following derivation we will assume the post-condition of the inner repetition in algorithm 4.1(P+S+, RT, NE): $lv = u \wedge v \in \mathbf{suff}(P) \wedge (l = \varepsilon \ \ \mathbf{cor}\ \ (l\upharpoonright 1)v \notin \mathbf{suff}(P))$. In fact, this post-condition can be rewritten with non-conditional disjunction in place of the conditional disjunction since $\varepsilon\upharpoonright 1 = \varepsilon$ by definition.

We now proceed to weaken the predicate, assuming $1 \leq n \leq |r|$:

$$
\begin{array}{ll}
 & \mathbf{suff}(u(r\upharpoonleft n)) \cap P \neq \varnothing \\
\equiv & \qquad \{\, u = lv \,\} \\
 & \mathbf{suff}(lv(r\upharpoonleft n)) \cap P \neq \varnothing \\
\Rightarrow & \qquad \{\, \text{split, and discard most of } l\text{, do not lookahead at } r,\ n \leq |r| \,\} \\
 & \mathbf{suff}(V^*(l\upharpoonright 1)vV^n) \cap P \neq \varnothing
\end{array}
$$

Notice that we have obtained a weaker predicate that does not depend on r. After substituting this predicate in the upperbound the restriction $n \leq |r|$ can be removed due to the second operand of the **min**-operator, $|r|$. We continue the derivation, assuming $n \geq 1$:

$$
\begin{array}{ll}
 & \mathbf{suff}(V^*(l\upharpoonright 1)vV^n) \cap P \neq \varnothing \\
\equiv & \qquad \{\, \text{property B.2} \,\} \\
 & V^*(l\upharpoonright 1)vV^n \cap V^*P \neq \varnothing \\
\equiv & \qquad \{\, V^*A \cap V^*B \neq \varnothing \equiv V^*A \cap B \neq \varnothing \vee V^*B \cap A \neq \varnothing \,\} \\
 & V^*(l\upharpoonright 1)vV^n \cap P \neq \varnothing \vee V^*P \cap (l\upharpoonright 1)vV^n \neq \varnothing \\
\Rightarrow & \qquad \{\, l = \varepsilon\text{: trivial; } l \neq \varepsilon\text{: property B.7} \,\} \\
 & V^*(l\upharpoonright 1)vV^n \cap P \neq \varnothing \vee V^*P \cap vV^n \neq \varnothing
\end{array}
$$

We now consider several further weakenings of this predicate.

4.2 Discarding the lookahead symbol

In the disjunct $V^*(l\upharpoonright 1)vV^n \cap P \neq \varnothing$ we discard $(l\upharpoonright 1)$:

$$
\begin{array}{ll}
 & V^*(l\upharpoonright 1)vV^n \cap P \neq \varnothing \vee V^*P \cap vV^n \neq \varnothing \\
\Rightarrow & \qquad \{\, \text{monotonicity of } \cap\text{: } V^*(l\upharpoonright 1) \subseteq V^* \,\} \\
 & V^*vV^n \cap P \neq \varnothing \vee V^*P \cap vV^n \neq \varnothing
\end{array}
$$

We now manipulate the **MIN** quantifier into a suitable form:

$$
\begin{array}{ll}
 & (\mathbf{MIN}\ n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\upharpoonleft n)) \cap P \neq \varnothing : n)\ \mathbf{min}\ |r| \\
\geq & \qquad \{\, \text{weakening of the range predicate using the preceding derivations} \,\} \\
 & (\mathbf{MIN}\ n : 1 \leq n \wedge (V^*vV^n \cap P \neq \varnothing \vee V^*P \cap vV^n \neq \varnothing) : n)\ \mathbf{min}\ |r| \\
= & \qquad \{\, \text{property of } \mathbf{MIN} \text{ with disjunctive range} \,\} \\
 & (\mathbf{MIN}\ n : 1 \leq n \wedge V^*vV^n \cap P \neq \varnothing : n) \\
 & \ \mathbf{min}(\mathbf{MIN}\ n : 1 \leq n \wedge V^*P \cap vV^n \neq \varnothing : n)\ \mathbf{min}\ |r|
\end{array}
$$

Since $v \in \mathbf{suff}(P)$ we can define two functions $d_1, d_2 : \mathbf{suff}(P) \longrightarrow \mathbb{N}$ by

$$\begin{array}{rcll} d_1(x) & = & (\mathbf{MIN}\, n : 1 \le n \wedge V^* x V^n \cap P \neq \varnothing : n) & (x \in \mathbf{suff}(P)) \\ d_2(x) & = & (\mathbf{MIN}\, n : 1 \le n \wedge V^* P \cap x V^n \neq \varnothing : n) & (x \in \mathbf{suff}(P)) \end{array}$$

Detail (NLA)**:** The lookahead term $l\lceil 1$ is discarded. Functions d_1 and d_2 can be precomputed and used to compute the no lookahead shift

$$k_{nla} = d_1(v)\,\mathbf{min}\,d_2(v)\,\mathbf{min}\,|r|$$

□

Using this detail gives a new algorithm (P+S+, RT, NE, CW, NLA). Precomputation of the two functions d_1 and d_2 is discussed in Part II, subsection 7.1.

4.3 Using the lookahead symbol

Instead of discarding the lookahead term $l\lceil 1$ it can also be taken into account.

Detail (LA)**:** The lookahead term $(l\lceil 1)$ is not discarded. □

$$\begin{array}{ll} & V^*(l\lceil 1)vV^n \cap P \neq \varnothing \vee V^*P \cap vV^n \neq \varnothing \\ \equiv & \quad \{ \text{monotonicity of } \cap\text{: } V^*(l\lceil 1) \subseteq V^* \} \\ & (V^*(l\lceil 1)vV^n \cap P \neq \varnothing \wedge V^*vV^n \cap P \neq \varnothing) \vee V^*P \cap vV^n \neq \varnothing \\ \Rightarrow & \quad \{ \text{monotonicity of } \cap\text{: } vV^n \subseteq V^{|v|+n} \} \\ & (V^*(l\lceil 1)V^{n+|v|} \cap P \neq \varnothing \wedge V^*vV^n \cap P \neq \varnothing) \vee V^*P \cap vV^n \neq \varnothing \end{array}$$

Detail (NEAR-OPT)**:** Define function $d_{no} : \mathbf{suff}(P) \times V \longrightarrow \mathbb{N}$ by

$$\begin{array}{l} d_{no}(x,a) = \\ \quad (\mathbf{MIN}\, n : 1 \le n \wedge (V^*aV^{n+|x|} \cap P \neq \varnothing \wedge V^*xV^n \cap P \neq \varnothing) \vee V^*P \cap xV^n \neq \varnothing : n) \end{array}$$

for $x \in \mathbf{suff}(P)$, $a \in V$, and use it to compute shift amount

$$k_{no} = \begin{cases} d_{no}(v, l\lceil 1)\,\mathbf{min}\,|r| & l\lceil 1 \neq \varepsilon \\ d_1(v)\,\mathbf{min}\,d_2(v)\,\mathbf{min}\,|r| & l\lceil 1 = \varepsilon \end{cases}$$

□

Using shift amount k_{no} yields algorithm (P+S+, RT, NE, CW, LA, NEAR-OPT). Precomputation of d_{no} is discussed in Part II, subsection 7.2.

Precomputation of d_{no} is rather expensive both in space and time. Moreover, storage of d_{no} requires $\mathcal{O}(|\mathbf{suff}(P)| \cdot |V|)$ space. Therefore, we derive another approximation, resulting in a more efficient precomputation, and less storage requirements. We derive

$$\begin{array}{ll} & d_{no}(v, (l\lceil 1))\,\mathbf{min}\,|r| \\ = & \quad \{ \text{definition of } d_{no} \text{ and } d_2\text{, disjunctive range in quantifier} \} \\ & ((\mathbf{MIN}\, n : 1 \le n \wedge V^*(l\lceil 1)V^{n+|v|} \cap P \neq \varnothing \wedge V^*vV^n \cap P \neq \varnothing : n))\,\mathbf{min}\,d_2(v)\,\mathbf{min}\,|r| \\ \ge & \quad \{ \text{conjunctive range in quantifier, definition of } d_1 \} \\ & ((\mathbf{MIN}\, n : 1 \le n \wedge V^*(l\lceil 1)V^{n+|v|} \cap P \neq \varnothing : n)\,\mathbf{max}\,d_1(v))\,\mathbf{min}\,d_2(v)\,\mathbf{min}\,|r| \\ \ge & \quad \{ \text{calculus} \} \\ & ((\mathbf{MIN}\, n : 1 \le n \wedge V^*(l\lceil 1)V^n \cap P \neq \varnothing : n - |v|)\,\mathbf{max}\,d_1(v))\,\mathbf{min}\,d_2(v)\,\mathbf{min}\,|r| \end{array}$$

Detail (NORM)**:** Define $d_3 : \mathbb{N} \times V \longrightarrow \mathbb{N}$ by

$$d_3(z, a) = (\mathbf{MIN}\ n : 1 \le n \wedge V^* a V^n \cap P \ne \varnothing : n - z) \qquad (z \in \mathbb{N}, a \in V),$$

functions d_1 and d_2 as in subsection 4.2, and use them to compute shift amount

$$k_{norm} = \begin{cases} (d_3(|v|, l\restriction 1)\ \mathbf{max}\ d_1(v))\ \mathbf{min}\ d_2(v)\ \mathbf{min}\ |r| & l\restriction 1 \ne \varepsilon \\ d_1(v)\ \mathbf{min}\ d_2(v)\ \mathbf{min}\ |r| & l\restriction 1 = \varepsilon \end{cases}$$

□

Using shift distance k_{norm} results in the normal Commentz-Walter algorithm (P_+S_+, RT, NE, CW, LA, NORM) (cf. [Com79a], section II, and [Com79b], sections II.1 and II.2). Precomputation of d_1 and d_2 is discussed in Part II, subsection 7.1, and precomputation of d_3 in Part II, subsection 7.3.

4.4 A derivation of the Boyer-Moore algorithm

We consider yet another weakening of the predicate — one that leads to a version of the regular Boyer-Moore algorithm. We derive, assuming $n \ge 1$,

$$\begin{array}{ll} & \mathbf{suff}(V^*(l\restriction 1)vV^n) \cap P \ne \varnothing \\ \equiv & \quad \{\text{property B.2}\} \\ & V^*(l\restriction 1)vV^n \cap V^*P \ne \varnothing \\ \equiv & \quad \{\text{monotonicity of } \cap\text{: } V^*(l\restriction 1) \subseteq V^*\} \\ & V^*(l\restriction 1)vV^n \cap V^*P \ne \varnothing \wedge V^*vV^n \cap V^*P \ne \varnothing \\ \Rightarrow & \quad \{\text{monotonicity of } \cap\text{: } vV^n \subseteq V^{n+|v|}\} \\ & V^*(l\restriction 1)V^{n+|v|} \cap V^*P \ne \varnothing \wedge V^*vV^n \cap V^*P \ne \varnothing \end{array}$$

We substitute this last predicate in the upperbound and derive

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \le n \wedge V^*(l\restriction 1)V^{n+|v|} \cap V^*P \ne \varnothing \wedge V^*vV^n \cap V^*P \ne \varnothing : n) \\ \ge & \quad \{(\mathbf{MIN}\ n : Q_0(n) \wedge Q_1(n) : n) \ge (\mathbf{MIN}\ n : Q_0(n) : n)\ \mathbf{max}(\mathbf{MIN}\ n : Q_1(n) : n)\} \\ & (\mathbf{MIN}\ n : 1 \le n \wedge V^*(l\restriction 1)V^{n+|v|} \cap V^*P \ne \varnothing : n) \\ & \ \mathbf{max}(\mathbf{MIN}\ n : 1 \le n \wedge V^*vV^n \cap V^*P \ne \varnothing : n) \\ \ge & \quad \{\text{changing bound variable: } n' = n + |v|\text{, enlarging range to } 1 \le n'\} \\ & (\mathbf{MIN}\ n' : 1 \le n' \wedge V^*(l\restriction 1)V^{n'} \cap V^*P \ne \varnothing : n' - |v|) \\ & \ \mathbf{max}(\mathbf{MIN}\ n : 1 \le n \wedge V^*vV^n \cap V^*P \ne \varnothing : n) \\ = & \quad \{V^*(l\restriction 1)V^m \cap V^*P \ne \varnothing\text{, where } m = (\mathbf{MIN}\ p \in P :: |p|)\} \\ & ((\mathbf{MIN}\ n : 1 \le n \wedge V^*(l\restriction 1)V^n \cap V^*P \ne \varnothing : n) - |v|) \\ & \ \mathbf{max}(\mathbf{MIN}\ n : 1 \le n \wedge V^*vV^n \cap V^*P \ne \varnothing : n) \end{array}$$

Detail (BM)**:** Define functions $char : V \longrightarrow \mathbb{N}$ and $d_{bm} : \mathbf{suff}(P) \longrightarrow \mathbb{N}$ by

$$\begin{array}{lcll} char(c) & = & (\mathbf{MIN}\ n : 1 \le n \wedge V^*cV^n \cap V^*P \ne \varnothing : n) & (c \in V) \\ d_{bm}(x) & = & (\mathbf{MIN}\ n : 1 \le n \wedge V^*xV^n \cap V^*P \ne \varnothing : n) & (x \in \mathbf{suff}(P)) \end{array}$$

and use them to compute the Boyer-Moore shift amount (cf. [BM77], section 4)

$$k_{bm} = \begin{cases} ((char(l\restriction 1) - |v|)\ \mathbf{max}\ d_{bm}(v))\ \mathbf{min}\ |r| & l\restriction 1 \ne \varepsilon \\ d_{bm}(v)\ \mathbf{min}\ |r| & l\restriction 1 = \varepsilon \end{cases}$$

□

Using shift amount k_{bm} results in the Boyer Moore algorithm[9] (P_+S_+, RT, NE, CW, BM). Precomputation of functions $char$ and d_{bm} is discussed in Part II, subsection 7.4. There it is also shown that $k_{norm} \ge k_{bm}$, meaning that the amount of shift in the normal Commentz-Walter algorithm (P_+S_+, RT, NE, CW, LA, NORM) is always at least the amount of shift in the Boyer-Moore algorithm (P_+S_+, RT, NE, CW, BM).

[9] The actual Boyer-Moore algorithm has the restriction of problem detail (OKW): $P = \{p\}$.

5 The Boyer-Moore family of algorithms

The Boyer-Moore algorithm derivation in the previous section only accounted for one method of traversing the variable u, in increasing order of v. In practice, when $P = \{p\}$ other methods of comparing v to keyword p can be used. We therefore introduce problem detail

Detail (OKW)**:** $P = \{p\}$ □

and starting with the original problem specification derive the Boyer-Moore algorithm and its variants.

We define a "perfect match" predicate

$$PM((l, v, r)) \equiv (lvr = S \wedge v = p)$$

and rewrite the postcondition into

$$R' : O = (\cup\ l, v, r : PM((l, v, r)) : \{(l, v, r)\}).$$

Define right shift function $Sh : (V^*)^3 \times \mathbb{N} \longrightarrow (V^*)^3$ by

$$Sh(l, v, r, k) = (l(vr\upharpoonright k), (v(r\upharpoonright k))\downharpoonleft k, r\downharpoonleft k).$$

By introduction of the "regular Boyer-Moore" algorithm detail

Detail (RBM)**:** Use function Sh and maintain invariant

$$\begin{array}{rcl} P_1(l, v, r) & \equiv & (lvr = S) \wedge (|v| \leq |p|) \wedge (|v| < |p| \Rightarrow r = \varepsilon) \\ & & \wedge\ (O = (\cup\ l', v', r' : PM((l', v', r')) \wedge (l'v' <_p lv) : \{(l', v', r')\})) \end{array}$$

□

we obtain a first (deterministic) solution (which is a phase shifted version of the algorithm in Appendix A.2)

Algorithm 5.1(OKW, RBM)

$l, v, r := \varepsilon, S\upharpoonright|p|, S\downharpoonleft|p|;\ O := \emptyset;$
$\{$invariant: $P_1(l, v, r)\}$
do $|v| = |p| \longrightarrow$
  if $v = p \longrightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $v \neq p \longrightarrow$ **skip**
  fi;
  $(l, v, r) := Sh(l, v, r, 1)$ $\{P_1(l, v, r)\}$
od $\{R'\}$

This algorithm does not take into account how we evaluate $v = p$. Define a "match order" to be a bijective function $mo : [1, |p|] \longrightarrow [1, |p|]$, i.e. a permutation of the integers $j : 1 \leq j \leq |p|$. This function is used to determine the order in which the individual symbols of v and p are compared. We now have

$$(v = p) \equiv (\forall i : 1 \leq i \leq |p| : v_{mo(i)} = p_{mo(i)}).$$

The match order detail is:

Detail (MO)**:** The characters of v and p are compared in a fixed order determined by a bijective function $mo : [1, |p|] \longrightarrow [1, |p|]$ (i.e. a permutation of $[1, |p|]$). □

The particular match order used in an algorithm determines the third position of the algorithm name. Three of the most common match orders, which represent particular instances of detail (MO), are

Detail (FWD)**:** The forward (or identity) match order given by $mo(i) = i$. □

Detail (REV)**:** The reverse match order given by $mo(i) = |p| - i + 1$. This is the original Boyer-Moore match order. □

Detail (OM)**:** Let $Pr : [1, |p|] \longrightarrow [0, 1]$ be the probability distribution of the symbols of p in input string S; the domain of function Pr consists of indices into p. Let an optimal mismatch match order be any permutation mo such that

$$(\forall i, j : 1 \leq i \leq |p| \wedge 1 \leq j \leq i : Pr(mo(j)) \leq Pr(mo(i))).$$

This match order is described as "optimal" because it compares characters of p in order of ascending probability of occurring in S. In this way, the least probable characters of p are compared first, so on the average one can expect to find any mismatch as early as possible. □

Comparing v and p using match order mo is done by procedure $Match$ specified by

---

$\{|v| = |p|\}$
$Match(\downarrow v, \downarrow p, \downarrow mo, \uparrow i)$
$\{P_2(v, p, mo, i) : (1 \leq i \leq |p| + 1) \wedge (i \leq |p| \Rightarrow v_{mo(i)} \neq p_{mo(i)})$
$\qquad \wedge (\forall j : 1 \leq j < i : v_{mo(j)} = p_{mo(j)})\}$

From $P_2(v, p, mo, i)$ it follows that $(v = p) \equiv (i = |p| + 1)$, and that if $i \leq |p|$ then $v_{mo(i)}$ is the first (in the given order) mismatching character. An implementation of $Match$ is

---

$i := 1$;
do $i \leq |p|$ **cand** $v_{mo(i)} = p_{mo(i)} \longrightarrow i := i + 1$ **od**

Adding mo, i, and $Match$ to the algorithm 5.1(OKW, RBM) results in

Algorithm 5.2(OKW, RBM, MO)

$l, v, r := \varepsilon, S\upharpoonleft|p|, S\downharpoonright|p|$; $O := \emptyset$;
$\{$invariant: $P_1(l, v, r)\}$
do $|v| = |p| \longrightarrow$
  $Match(v, p, mo, i)$;
  $\{P_2(v, p, mo, i)\}$
  if $i = |p| + 1 \longrightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $i \neq |p| + 1 \longrightarrow$ **skip**
  fi;
  $(l, v, r) := Sh(l, v, r, 1)$ $\{P_1(l, v, r)\}$
od $\{R'\}$

In some versions of the Boyer-Moore algorithms $Match$ is only executed after a successful comparison of a character of p which is least frequent in S, and the corresponding character of v. In the taxonomy in [HS91] this comparison is called the *guard* and the character of p involved the *guard character*. We do not consider it here since it can be viewed as additionally requiring that $p_{mo(1)}$ is a character of p with minimal frequency in S.

5.1 Larger shifts without using $Match$ information

It may be possible to make an additional shift (immediately before $Match$ is performed) providing no matches are missed. A shift of not greater than $(\mathbf{MIN}\ k : 0 \le k \wedge PM(Sh(l,v,r,k)) : k)$ will be safe. This can be done with the statement

$$
\begin{array}{l}
\{|v| = |p|\} \\
(l,v,r) := Sh(l,v,r,(\mathbf{MIN}\ k : 0 \le k \wedge PM(Sh(l,v,r,k)) : k)\,\mathbf{min}\,|r|) \\
\{|v| = |p| \wedge (r = \varepsilon \vee PM(l,v,r))\}
\end{array}
$$

The $\mathbf{min}\,|r|$ is used to ensure that $|v| = |p|$ is maintained. Another implementation of the shift is

$$
\begin{array}{l}
\{|v| = |p|\} \\
\mathbf{do}\ 1 \le |r| \wedge \neg PM(l,v,r) \longrightarrow \\
\qquad (l,v,r) := Sh(l,v,r,(\mathbf{MIN}\ k : 1 \le k \wedge PM(Sh(l,v,r,k)) : k)\,\mathbf{min}\,|r|) \\
\mathbf{od} \\
\{|v| = |p| \wedge (r = \varepsilon \vee PM(l,v,r))\}
\end{array}
$$

This could have been implemented with an **if-fi** construct, however, the **do-od** construct will prove to be more useful when the shift distance is approximated from below. The **do-od** version is known as a *skip loop* in the taxonomy of Hume and Sunday [HS91].

Calculating the **MIN** quantification is essentially as difficult as the problem we are trying to solve. Since any smaller shift length suffices, we consider weakenings of predicate PM. Some weakenings are: $Q_0((l,v,r)) \equiv true$, $Q_1((l,v,r)) \equiv (v_1 = p_1)$, $Q_2((l,v,r)) \equiv (v_{|p|} = p_{|p|})$, and $Q_3((l,v,r)) \equiv (v_j = p_j)$ (for some $j : 1 \le j \le |p|$); the predicates Q_1, Q_2 and Q_3 require that $p \ne \varepsilon$. Predicates Q_1 and Q_2 are special cases of Q_3. We can of course take the conjunction of any of these weakenings and still have a weakening of PM.

For each weakening of PM, we consider the shift length as calculated with the quantified **MIN** . In the case of Q_0, the entire skip loop is equivalent to **skip**.

We consider the shift length for Q_3 before returning to Q_1 and Q_2 as special cases. We need to compute

$$(\mathbf{MIN}\ k : 1 \le k \wedge PM(Sh(l,v,r,k)) : k)$$

In order to easily compute this we will weaken the range predicate, removing lookahead. Additionally, it is known (from the **do-od** guard) that $\neg Q_3((l,v,r))$ holds. The derivation proceeds as follows (assuming $1 \le k \le |r|$, $\neg Q_3((l,v,r))$ and fixed $j : 1 \le j \le |p|$):

$$
\begin{array}{ll}
 & PM(Sh(l,v,r,k)) \\
\equiv & \qquad \{\text{ definition of } Sh \,\} \\
 & PM((l(vr\upharpoonright k),(v(r\upharpoonright k))\downharpoonright k, r\downharpoonright k)) \\
\equiv & \qquad \{\text{ definition of } PM \,\} \\
 & (v(r\upharpoonright k))\downharpoonright k = p \\
\equiv & \qquad \{\text{ definition of} = \text{on strings} \,\} \\
 & (\forall i : 1 \le i \le |p| : ((v(r\upharpoonright k))\downharpoonright k)_i = p_i) \\
\equiv & \qquad \{\text{ rewrite } \downharpoonright \text{ into indexing} \,\} \\
 & (\forall i : 1 \le i \le |p| : (v(r\upharpoonright k))_{i+k} = p_i) \\
\Rightarrow & \qquad \{\text{ discard lookahead at } r,\ |v| = |p| \,\} \\
 & (\forall i : 1 \le i \le |p| - k : v_{i+k} = p_i) \\
\equiv & \qquad \{\text{ change of bound variable: } i' = i + k.\ \neg Q_3((l,v,r)) \,\}
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall i' : 1+k \le i' \le |p| : v'_i = p_{i'-k}) \wedge v_j \ne p_j \\
\Rightarrow & \quad \{\text{ one point rule }\} \\
 & 1+k \le j \Rightarrow v_j = p_{j-k} \wedge v_j \ne p_j \\
\equiv & \quad \{\text{ transitivity of } = \} \\
 & 1+k \le j \Rightarrow v_j = p_{j-k} \wedge p_j \ne p_{j-k}
\end{array}
$$

The final predicate is free of r, and so the upperbound of $|r|$ on k can be dropped.

Given $j : 1 \le j \le |p|$ we can define a function and a constant

$$
\begin{array}{lcl}
sl_1(a) & = & (\mathbf{MIN}\ k : 1 \le k \wedge (1+k \le j \Rightarrow a = p_{j-k}) : k) \\
sl_2 & = & (\mathbf{MIN}\ k : 1 \le k \wedge (1+k \le j \Rightarrow p_j \ne p_{j-k}) : k)
\end{array}
$$

Functions sl_1 and sl_2 can be combined to give a shift of $(sl_1(v_j)\,\mathbf{max}\,sl_2)\,\mathbf{min}\,|r|$. In practice sl_1 and sl_2 are frequently combined into one function. In section 5.2 we will show how sl_1 and sl_2 can be obtained from two functions computed for a different purpose. If a conjunct of any of Q_0, Q_1, Q_2, or Q_3 is used as a weakening of PM, the appropriate skip length can be approximated as the **max** of the individual skip lengths. A particularly interesting skip length is that arising from predicate Q_1. In this case, $sl_1(a) = 1$ and $sl_2 = 1$ and a skip length of 1 is used.

Assuming Q is a weakening of PM we introduce program detail

Detail (SL)**:** Comparison of v and p is preceded bye a *skip loop* based upon weakening Q of PM and some appropriate skip length. □

Assuming some fixed $j : 1 \le j \le |p|$ we use Q_3 as an example of a weakening of PM in

Algorithm 5.3(OKW, RBM, MO, SL)

$$
\begin{array}{l}
l, v, r := \varepsilon, S\!\upharpoonleft\!|p|, S\!\downharpoonleft\!|p|;\ O := \varnothing; \\
\{\text{invariant: } P_1(l,v,r)\} \\
\mathbf{do}\ |v| = |p| \longrightarrow \\
\quad \{|v| = |p|\} \\
\quad \mathbf{do}\ 1 \le |r| \wedge \neg Q_3((l,v,r)) \longrightarrow (l,v,r) := Sh(l,v,r,(sl_1(v_j)\,\mathbf{max}\,sl_2)\,\mathbf{min}\,|r|)\ \mathbf{od}; \\
\quad \{|v| = |p| \wedge (Q_3((l,v,r)) \vee r = \varepsilon)\} \\
\quad Match(v,p,mo,i); \qquad \{P_2(v,p,mo,i)\} \\
\quad \mathbf{if}\ i = |p|+1 \longrightarrow O := O \cup \{(l,v,r)\} \\
\quad [\!]\ \ i \ne |p|+1 \longrightarrow \mathbf{skip} \\
\quad \mathbf{fi}; \\
\quad (l,v,r) := Sh(l,v,r,1) \qquad \{P_1(l,v,r)\} \\
\mathbf{od}\ \{R'\}
\end{array}
$$

We proceed by presenting four instances of detail (SL) (each based on a weakening of PM)[10]:

Detail (NONE)**:** The predicate Q_0 ($true$) is used as the weakening of PM in the skip loop. Notice that in this case the skip loop is equivalent to statement **skip**. □

Detail (SFC[11])**:** The predicate Q_1 is used as the weakening of PM in the skip loop. □

Detail (FAST)**:** The predicate Q_2 is used as the weakening of PM in the skip loop. □

Detail (SLFC[12])**:** Let p_j be a character of p with minimal frequency in S. Predicate Q_3, defined by

$$Q_3((l,v,r)) \equiv v_j = p_j\,,$$

is used as the weakening of PM in the skip loop. □

[10] names are taken from the taxonomy in [HS91]
[11] search first character
[12] search least frequent character

5.2 Making use of *Match* information

Up to now information from previous matching attempts was not used in the computation of the shift distance (in fact there was no computation and the shift distance defaulted to 1). In this subsection we will take into account the information from the immediately preceding matching attempt.

With a shift of k symbols, p will be compared against $(vr\lfloor k)\lceil|p|$. Ideally, we would like to select our shift k such that it is the smallest k satisfying $1 \le k \le |r|$ and

$$(\forall j : 1 \le j \le |p| : (vr\lfloor k)_j = p_j).$$

Again, we apply the technique of weakening such a predicate, thereby obtaining approximations of the optimal shift distance from below. The weakening of the predicate should, amongst others, include the removal of any reference to r (no lookahead).

In the following calculations we assume $k \le |r| \wedge |v| = |p|$ and the postcondition of *Match*, namely $P_2(v, p, mo, i)$. We derive

$$\begin{array}{ll}
& (\forall j : 1 \le j \le |p| : (vr\lfloor k)_j = p_j) \\
\equiv & \quad \{\, |v| = |p|,\ k \le |r|, \text{ hence } (vr\lfloor k)_j = (vr)_{j+k} \,\} \\
& (\forall j : 1 \le j \le |p| : (vr)_{j+k} = p_j) \\
\equiv & \quad \{\, \text{change of bound variable: } j' = j + k \,\} \\
& (\forall j' : 1 + k \le j' \le |p| + k : (vr)_{j'} = p_{j'-k}) \\
\Rightarrow & \quad \{\, |v| = |p|, \text{ remove references to characters of } r \,\} \\
& (\forall j' : 1 + k \le j' \le |p| : v_{j'} = p_{j'-k}) \\
\equiv & \quad \{\, \text{change of bound variable: } j' = mo(j) \,\} \\
& (\forall j : 1 \le j \le |p| \wedge 1 + k \le mo(j) : v_{mo(j)} = p_{mo(j)-k}) \\
\equiv & \quad \{\, P_2(v, p, mo, i) \,\} \\
& (\forall j : 1 \le j \le |p| \wedge 1 + k \le mo(j) : v_{mo(j)} = p_{mo(j)-k}) \\
& \wedge (i \le |p| \Rightarrow v_{mo(i)} \ne p_{mo(i)}) \wedge (\forall j : 1 \le j < i : v_{mo(j)} = p_{mo(j)}) \\
\Rightarrow & \quad \{\, \text{combine } \forall \text{ quantifiers, with restricted range, since } 1 \le i \le |p| + 1 \,\} \\
& (\forall j : 1 \le j < i \wedge 1 + k \le mo(j) : p_{mo(j)} = p_{mo(j)-k}) \\
& \wedge (i \le |p| \textbf{ cand } 1 + k \le mo(i) \Rightarrow v_{mo(i)} = p_{mo(i)-k} \wedge p_{mo(i)} \ne p_{mo(i)-k})
\end{array}$$

The last predicate in the preceding derivation will be denoted by $P_3(v, i, k)$ (here we have chosen to make parameters mo and p implicit). We now define the shift distance k based on previous match information by

$$k = (\textbf{MIN}\ j : 1 \le j \wedge P_3(v, i, j) : j)\,\textbf{min}(|r| + 1).$$

Notice that this shift distance still depends on implicit parameter mo. The predicate P_3 is frequently weakened further (most often the conjunct $p_{mo(i)} \ne p_{mo(i)-k}$ is discarded). In much of the literature, P_3 is broken up into

$$\begin{array}{lcl}
P_3'(i, k) & \equiv & (\forall j : 1 \le j < i \wedge 1 + k \le mo(j) : p_{mo(j)} = p_{mo(j)-k}) \\
P_3''(v, i, k) & \equiv & (i \le |p| \textbf{ cand } 1 + k \le mo(i) \Rightarrow v_{mo(i)} = p_{mo(i)-k}) \\
P_3'''(i, k) & \equiv & (i \le |p| \textbf{ cand } 1 + k \le mo(i) \Rightarrow p_{mo(i)} \ne p_{mo(i)-k})
\end{array}$$

This leads to three functions $s_1 : \mathbb{N} \longrightarrow \mathbb{N}$, $char_1 : V^{|p|} \times \mathbb{N} \longrightarrow \mathbb{N}$, and $char_2 : \mathbb{N} \longrightarrow \mathbb{N}$ defined by

$$\begin{array}{lcll}
s_1(i) & = & (\textbf{MIN}\ k : 1 \le k \wedge P_3'(i, k) : k) & (i \in \mathbb{N}) \\
char_1(v, i) & = & (\textbf{MIN}\ k : 1 \le k \wedge P_3''(v, i, k) : k) & (v \in V^{|p|}, i \in \mathbb{N}) \\
char_2(i) & = & (\textbf{MIN}\ k : 1 \le k \wedge P_3'''(i, k) : k) & (i \in \mathbb{N})
\end{array}$$

Applying these functions yields a new, possibly smaller, shift distance

$$k = (s_1(i)\,\textbf{max}\,char_1(v, i)\,\textbf{max}\,char_2(i))\,\textbf{min}(|r| + 1).$$

This is known as the *match information* detail

Detail (MI)**:** Use information from the preceding match attempt by computing the shift distance using functions s_1, $char_1$, and $char_2$. □

Adding this detail results in the following Boyer-Moore algorithm skeleton (details (MO) and (SL) still have to be instantiated), for weakening Q_3 of PM (cf. [HS91], section 4, p.1224):

Algorithm 5.4(OKW, RBM, MO, SL, MI)

$l, v, r := \varepsilon, S\upharpoonleft|p|, S\downharpoonleft|p|$; $O := \emptyset$;
{invariant: $P_1(l, v, r)$}
do $|v| = |p| \longrightarrow$
  $\{|v| = |p|\}$
  do $1 \leq |r| \wedge \neg Q_3((l, v, r)) \longrightarrow (l, v, r) := Sh(l, v, r, (sl_1(v_j)\, \mathbf{max}\, sl_2)\, \mathbf{min}\, |r|)$ **od**;
  $\{|v| = |p| \wedge (Q_3((l, v, r)) \vee r = \varepsilon)\}$
  $Match(v, p, mo, i)$; $\{P_2(v, p, mo, i)\}$
  if $i = |p| + 1 \longrightarrow O := O \cup \{(l, v, r)\}$
  $[\!]$ $i \neq |p| + 1 \longrightarrow$ **skip**
  fi;
  $k := (s_1(i)\, \mathbf{max}\, char_1(v, i)\, \mathbf{max}\, char_2(i))\, \mathbf{min}(|r| + 1)$;
  $(l, v, r) := Sh(l, v, r, k)$ $\{P_1(l, v, r)\}$
od $\{R'\}$

Precomputation of functions s_1, $char_1$, and $char_2$ is discussed in Part II, subsection 7.5 for instantiations (FWD) and (REV) of algorithm detail (MO).

Given fixed $j : 1 \leq j \leq |p|$ we can easily compute the function sl_1 and constant sl_2 from section 5.1. This can be done for any particular mo. The functions are

$$\begin{array}{lcl} sl_1(v_j) & = & char_1(v, mo^{-1}(j)) \\ sl_2 & = & char_2(mo^{-1}(j)) \end{array}$$

Part II
Precomputation

In this part we derive algorithms for the precomputation of the functions used in the pattern matching algorithms in Part I. 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 Commentz-Walter family of algorithms. They can, amongst others, be specialized to a correct precomputation algorithm for the single keyword Boyer-Moore algorithm.

6 Precomputation for the Aho-Corasick algorithms

First, we consider the transition function of the forward trie corresponding to P $\tau_{P,f} : \mathbf{pref}(P) \times V \longrightarrow (\mathbf{pref}(P) \cup \{\perp\})$ defined by

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

Since $\mathbf{pref}$ is idempotent and the definition of $\tau_{P,f}$ only depends on $\mathbf{pref}(P)$, we have $\tau_{P,f} = \tau_{\mathbf{pref}(P),f}$. Set P being nonempty we also have $\mathbf{pref}(P) = \{\varepsilon\} \cup \mathbf{pref}(P)$ and, hence, $\tau_{\mathbf{pref}(P),f} = \tau_{\{\varepsilon\}\cup\mathbf{pref}(P),f}$. These observations lead to the following algorithm to compute $\tau_{P,f}$ in which variable tau is used to calculate and store $\tau_{P,f}$ (cf. [AC75], section 3, algorithm 2):

$\{tau = \tau_{\varnothing,f}\}$
for $a : a \in V$ **do** $tau(\varepsilon, a) := \perp$ **rof**;
$\{tau = \tau_{\{\varepsilon\},f}\}$
$P_d, P_r := \varnothing, P$;
$\{\text{invariant: } P_d \cup P_r = P \wedge P_d \cap P_r = \varnothing \wedge tau = \tau_{\{\varepsilon\}\cup\mathbf{pref}(P_d),f}\}$
do $P_r \neq \varnothing \longrightarrow$
  $p : p \in P_r$;
  $u, v := \varepsilon, p$;
  $\{\text{invariant: } uv = p \wedge tau = \tau_{\{\varepsilon\}\cup\mathbf{pref}(P_d)\cup\mathbf{pref}(u),f}\}$
  do $v \neq \varepsilon \longrightarrow$
    if $tau(u, v\upharpoonright 1) = \perp \longrightarrow$
      $tau(u, v\upharpoonright 1) := u(v\upharpoonright 1)$;
      for $a : a \in V$ **do** $tau(u(v\upharpoonright 1), a) := \perp$ **rof**
    $[\!]$ $tau(u, v\upharpoonright 1) \neq \perp \longrightarrow$ **skip**
    fi;
    $u, v := u(u\upharpoonright 1), v\downharpoonright 1$
  od;
  $P_d, P_r := P_d + \{p\}, P_r - \{p\}$
od $\{tau = \tau_{P,f}\}$

Notice that the algorithm does a depth first traversal of the forward 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 forward 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.

The extended forward trie corresponding to P $\tau_{P,ef} : \mathbf{pref}(P) \times V \longrightarrow (\mathbf{pref}(P) \cup \{\bot\})$ is defined by

$$\tau_{P,ef}(u,a) = \begin{cases} ua & \text{if } ua \in \mathbf{pref}(P) \\ \varepsilon & \text{if } u = \varepsilon \wedge a \notin \mathbf{pref}(P) \\ \bot & \text{if } u \neq \varepsilon \wedge ua \notin \mathbf{pref}(P) \end{cases} \qquad (u \in \mathbf{pref}(P), a \in V).$$

It can be computed by the algorithm obtained by adding statement

---

for $a : tau(\varepsilon, a) = \bot$ **do** $tau(\varepsilon, a) := \varepsilon$ **rof**

---

to the end of the algorithm computing $\tau_{P,f}$.

Next, we focus on the computation of function $\gamma_f : \mathbf{pref}(P) \times V \longrightarrow \mathbf{pref}(P)$, defined by

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

and $f_f : \mathbf{pref}(P) \setminus \{\varepsilon\} \longrightarrow \mathbf{pref}(P)$, defined by

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

In order to arrive at an algorithm computing both γ_f and f_f we first derive (mutually) recursive definitions of γ_f and f_f.

i. Let $a \in V$. We derive

$$\begin{aligned}
& \gamma_f(\varepsilon, a) \\
= & \quad \{ \text{definition of } \gamma_f \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(a) \cap \mathbf{pref}(P) : w) \\
= & \quad \{ \text{case analysis} \} \\
& \begin{cases} a & \text{if } a \in \mathbf{pref}(P) \\ \varepsilon & \text{if } a \notin \mathbf{pref}(P) \end{cases}
\end{aligned}$$

ii. Let $u \in \mathbf{pref}(P) \setminus \{\varepsilon\}$ and $a \in V$. We distinguish two cases.

a. Assume $ua \in \mathbf{pref}(P)$. Then by definition of γ_f we have $\gamma_f(u,a) = ua$.

b. Assume $ua \notin \mathbf{pref}(P)$. Let $u = bu_0$ where $b \in V$ and $u_0 \in V^*$. We derive

$$\begin{aligned}
& \gamma_f(u,a) \\
= & \quad \{ \text{definition of } \gamma_f \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(ua) \cap \mathbf{pref}(P) : w) \\
= & \quad \{ ua \notin \mathbf{pref}(P),\ \mathbf{suff}(ua) \setminus \{ua\} = (\mathbf{suff}(u) \setminus \{u\})a \cup \{\varepsilon\},\ P \neq \varnothing \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(u) \setminus \{u\})a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{ u = bu_0,\ \mathbf{suff}(u) \setminus \{u\} = \mathbf{suff}(u_0) \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u_0)a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= & \quad \{ \mathbf{suff} \text{ is idempotent, Theorem B.5} \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}((\mathbf{MAX}_{\leq_s}\, v : v \in \mathbf{suff}(u_0) \cap \mathbf{pref}(P) : v))a \cap \mathbf{pref}(P) \\
& \qquad \vee w = \varepsilon : w) \\
= & \quad \{ u = bu_0,\ \mathbf{suff}(u_0) = \mathbf{suff}(u) \setminus \{u\} \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}((\mathbf{MAX}_{\leq_s}\, v : v \in (\mathbf{suff}(u) \setminus \{u\}) \cap \mathbf{pref}(P) : v))a \cap \mathbf{pref}(P) \\
& \qquad \vee w = \varepsilon : w) \\
= & \quad \{ \text{definition of } f_f \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(f_f(u))a \cap \mathbf{pref}(P) \vee w = \varepsilon : w)
\end{aligned}$$

$$
\begin{aligned}
= &\quad \{\, \mathbf{suff}(f_f(u)a) = \mathbf{suff}(f_f(u))a \cup \{\varepsilon\},\ P \neq \varnothing \,\} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(f_f(u)a) \cap \mathbf{pref}(P) : w) \\
= &\quad \{\, \text{definition of } \gamma_f \,\} \\
& \gamma_f(f_f(u), a)
\end{aligned}
$$

Observe that the need for function f_f arises naturally in this derivation.

iii. Let $a \in V$ such that $a \in \mathbf{pref}(P)$. We derive

$$
\begin{aligned}
& f_f(a) \\
= &\quad \{\, \text{definition of } f_f \,\} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(a) \setminus \{a\}) \cap \mathbf{pref}(P) : w) \\
= &\quad \{\, \mathbf{suff}(a) = \{\varepsilon, a\},\ P \neq \varnothing \,\} \\
& \varepsilon
\end{aligned}
$$

iv. Let $u \in V^* \setminus \{\varepsilon\}$ and $a \in V$ such that $ua \in \mathbf{pref}(P)$. We derive

$$
\begin{aligned}
& f_f(ua) \\
= &\quad \{\, \text{definition of } f_f \,\} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(ua) \setminus \{ua\}) \cap \mathbf{pref}(P) : w) \\
= &\quad \{\, \mathbf{suff}(ua) \setminus \{ua\} = (\mathbf{suff}(u) \setminus \{u\})a \cup \{\varepsilon\},\ P \neq \varnothing \,\} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(u) \setminus \{u\})a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= &\quad \{\, \text{see derivation in } \mathbf{ii.b.} \,\} \\
& \gamma_f(f_f(u), a)
\end{aligned}
$$

Summarizing, we have

$$
\begin{aligned}
\gamma_f(\varepsilon, a) &= \begin{cases} \varepsilon & \text{if } a \notin \mathbf{pref}(P) \\ a & \text{if } a \in \mathbf{pref}(P) \end{cases} && (a \in V) \\
\gamma_f(u, a) &= \begin{cases} ua & \text{if } ua \in \mathbf{pref}(P) \\ \gamma_f(f_f(u), a) & \text{if } ua \notin \mathbf{pref}(P) \end{cases} && (u \in \mathbf{pref}(P) \setminus \{\varepsilon\}, a \in V) \\
f_f(a) &= \varepsilon && (a \in V, a \in \mathbf{pref}(P)) \\
f_f(ua) &= \gamma_f(f_f(u), a) && (u \in \mathbf{pref}(P) \setminus \{\varepsilon\}, a \in V, ua \in \mathbf{pref}(P))
\end{aligned}
$$

Since $(\forall u : u \in \mathbf{pref}(P) \setminus \{\varepsilon\} : |f_f(u)| < |u|)$ the functions γ_f and f_f can be computed by the following algorithm that is based upon the preceding recursive definitions (notice the layer wise or breadth first traversal of $\mathbf{pref}(P)$; algorithm variables *gf* and *ff* are used to calculate and store γ_f and f_f, respectively; cf. [AC75], a combination of algorithm 3 from section 3 and algorithm 4 from section 6):

```
for a : a ∈ V do
    if a ∈ pref(P) ⟶ gf(ε, a) := a; ff(a) := ε
    [] a ∉ pref(P) ⟶ gf(ε, a) := ε
    fi
rof;
n := 1;
{invariant:
        (∀u, a : u ∈ pref(P) ∧ |u| < n ∧ a ∈ V : gf(u, a) = γ_f(u, a))
      ∧ (∀u : u ∈ pref(P) \ {ε} ∧ |u| ≤ n : ff(u) = f_f(u))}
do pref(P) ∩ V^n ≠ ∅ ⟶
    for u : u ∈ pref(P) ∩ V^n do
        for a : a ∈ V do
            if ua ∈ pref(P) ⟶ gf(u, a) := ua; ff(ua) := gf(ff(u), a)
            [] ua ∉ pref(P) ⟶ gf(u, a) := gf(ff(u), a)
            fi
        rof
    rof;
    n := n + 1
od
```

If the forward trie τ_f has already been computed and represented by tau, then the guard "$ua \in$ **pref**(P)" in the preceding algorithm can be replaced by "$tau(u, a) \neq \perp$".

Next, we show how to compute failure function f_f without function γ_f using linear search. For $u \in \mathbf{pref}(P) \setminus \{\varepsilon\}$, $a \in V$, and $ua \in \mathbf{pref}(P)$ we derive

$$
\begin{aligned}
& f_f(ua) \\
= \quad & \{ \text{see derivation } \mathbf{iv.} \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(u) \setminus \{u\})a \cap \mathbf{pref}(P) \vee w = \varepsilon : w) \\
= \quad & \{ \text{domain split, } (\mathbf{suff}(u) \setminus \{u\})a \cap \mathbf{pref}(P) \subseteq \mathbf{pref}(P)a \} \\
& (\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(u) \setminus \{u\} \cap \mathbf{pref}(P))a \cap \mathbf{pref}(P) : w)\, \mathbf{max}_{\leq_s}\, \varepsilon \\
= \quad & \{ \text{change of bound variable: } w = w'a, \text{ properties of } \perp_s \} \\
& (\mathbf{MAX}_{\leq_s}\, w' : w' \in \mathbf{suff}(u) \setminus \{u\} \cap \mathbf{pref}(P) \wedge w'a \in \mathbf{pref}(P) : w')a\, \mathbf{max}_{\leq_s}\, \varepsilon
\end{aligned}
$$

As in 3.3.1 this expression can be computed using a linear search

```
{(∀v : v ∈ pref(P) ∧ v <_s u : ff(v) = f_f(v))}
u' := ff(u);
do τ_ef(u', a) = ⊥_s ⟶ u' := ff(u') od;
ff(ua) := τ_ef(u', a)
```

This leads to the following algorithm computing failure function f_f (notice the breadth first traversal of $\mathbf{pref}(P) \setminus \{\varepsilon\}$; cf. [AC75], section 3, algorithm 3):

```
{tau = τ_ef}
for a : a ∈ V do
      if a ∈ pref(P) ⟶ ff(a) := ε
      [] a ∉ pref(P) ⟶ skip
      fi
rof;
n := 1;
{invariant: (∀u : u ∈ pref(P) \ {ε} ∧ |u| ≤ n : ff(u) = f_f(u))}
do pref(P) ∩ V^n ≠ ∅ ⟶
      for u : u ∈ pref(P) ∩ V^n do
            for a : a ∈ V do
                  if ua ∈ pref(P) ⟶
                        u' := ff(u);
                        do tau(u', a) = ⊥_s ⟶ u' := ff(u') od;
                        ff(ua) := tau(u', a)
                  [] ua ∉ pref(P) ⟶ skip
                  fi
            rof
      rof;
      n := n + 1
od
```

Finally, we consider the precomputation of function $Output : \mathbf{pref}(P) \longrightarrow \mathcal{P}(P)$ defined by $Output(u) = \mathbf{suff}(u) \cap P$. A recursive definition of $Output$ is derived as follows:

i. By definition we have $Output(\varepsilon) = \{\varepsilon\} \cap P$.

ii. Let $u \in \mathbf{pref}(P) \setminus \{\varepsilon\}$. Let $u = bu_0$ where $b \in V$ and $u_0 \in V^*$. We derive

$$
\begin{array}{ll}
 & Output(u) \\
= & \quad \{ \text{ definition of } Output \,\} \\
 & \mathbf{suff}(u) \cap P \\
= & \quad \{ \mathbf{suff}(u) = (\mathbf{suff}(u) \setminus \{u\}) \cup \{u\} \,\} \\
 & ((\mathbf{suff}(u) \setminus \{u\}) \cap P) \cup (\{u\} \cap P) \\
= & \quad \{ u = bu_0,\ \mathbf{suff}(u) \setminus \{u\} = \mathbf{suff}(u_0),\ P \subseteq \mathbf{pref}(P) \,\} \\
 & (\mathbf{suff}(u_0) \cap \mathbf{pref}(P) \cap P) \cup (\{u\} \cap P) \\
= & \quad \{ \mathbf{suff} \text{ is idempotent, Theorem B.5} \,\} \\
 & (\mathbf{suff}((\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(u_0) \cap \mathbf{pref}(P) : w)) \cap \mathbf{pref}(P) \cap P) \cup (\{u\} \cap P) \\
= & \quad \{ P \subseteq \mathbf{pref}(P),\ u = bu_0,\ \mathbf{suff}(u_0) = \mathbf{suff}(u) \setminus \{u\} \,\} \\
 & (\mathbf{suff}((\mathbf{MAX}_{\leq_s}\, w : w \in (\mathbf{suff}(u) \setminus \{u\}) \cap \mathbf{pref}(P) : w)) \cap P) \cup (\{u\} \cap P) \\
= & \quad \{ \text{ definition of } f_f,\ u \in \mathbf{pref}(P) \setminus \{\varepsilon\}, \text{ definition of } Output \,\} \\
 & Output(f_f(u)) \cup (\{u\} \cap P)
\end{array}
$$

By preceding the algorithm on page 34 with assignment "$out(\varepsilon) := \{\varepsilon\} \cap P$", and by adding assignment "$out(a) := out(\varepsilon) \cup (\{a\} \cap P)$" to the end of the first alternative of its first **if -fi** statement and assignment "$out(ua) := out(\mathit{ff}(ua)) \cup (\{ua\} \cap P)$" to the end of the first alternative of its second **if -fi** statement one obtains an algorithm computing function $Output$ as well.

7 Precomputation for the Commentz-Walter algorithms

In this section we will be using the reverse trie corresponding to P $\tau_r : \mathbf{suff}(P) \times V \longrightarrow \mathbf{suff}(P) \cup \{\bot\}$ defined by

$$\tau_r(u,a) = \begin{cases} au & \text{if } au \in \mathbf{suff}(P) \\ \bot & \text{if } au \notin \mathbf{suff}(P) \end{cases} \qquad (u \in \mathbf{suff}(P), a \in V),$$

its optimal transition function $\gamma_r : \mathbf{suff}(P) \times V \longrightarrow \mathbf{suff}(P)$ defined by

$$\gamma_r(q,a) = (\mathbf{MAX}_{\leq_p}\, w : w \in \mathbf{pref}(qa) \cap \mathbf{suff}(P) : w) \quad (q \in \mathbf{suff}(P), a \in V),$$

and its failure function $f_r : \mathbf{suff}(P) \setminus \{\varepsilon\} \longrightarrow \mathbf{suff}(P)$ defined by

$$f_r(q) = (\mathbf{MAX}_{\leq_p}\, w : w \in (\mathbf{pref}(q) \setminus \{q\}) \cap \mathbf{suff}(P) : w) \quad (q \in \mathbf{suff}(P) \setminus \{\varepsilon\}).$$

These functions are the mirror image of the functions corresponding to the forward trie and can be computed by algorithms that are the mirror images of the algorithms in the previous section.

7.1 Computation of d_1 and d_2

Next, we consider the computation of function $d_1 : \mathbf{suff}(P) \longrightarrow \mathbb{N}$ defined by

$$d_1(x) = (\mathbf{MIN}\, n : n \geq 1 \wedge V^* x V^n \cap P \neq \varnothing : n) \qquad (x \in \mathbf{suff}(P)).$$

Let $x \in \mathbf{suff}(P)$. We derive

$$\begin{array}{ll}
 & (\mathbf{MIN}\, n : n \geq 1 \wedge V^* x V^n \cap P \neq \varnothing : n) \\
= & \quad \{\text{property B.2}\} \\
 & (\mathbf{MIN}\, n : n \geq 1 \wedge (xV^n) \cap \mathbf{suff}(P) \neq \varnothing : n) \\
= & \quad \{\text{change of bound variable: } n = |s|\} \\
 & (\mathbf{MIN}\, s : s \in V^+ \wedge xs \in \mathbf{suff}(P) : |s|) \\
= & \quad \{\text{change of bound variable: } t = xs\} \\
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p t : |t| - |x|) \\
= & \quad \{x, t \in \mathbf{suff}(P),\ t \neq \varepsilon, \text{ lemma B.8}\} \\
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x \leq_p f_r(t) : |t| - |x|) \\
= & \quad \{\text{domain split}\} \\
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x = f_r(t) : |t| - |x|) \\
 & \mathbf{min}(\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p f_r(t) : |t| - |x|) \\
= & \quad \{\text{see following note}\} \\
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x = f_r(t) : |t| - |x|)
\end{array}$$

Note

In order to show that the second operand of the **min**-operator can be omitted we distinguish two cases:

i. Assume $\neg(\exists t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} : x <_p f_r(t))$. The second operand now equals the unity of the **min**-operator.

ii. Assume $(\exists t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} : x <_p f_r(t))$. We derive

$$\begin{array}{ll}
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p f_r(t) : |t| - |x|) \\
> & \quad \{(\exists t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} : x <_p f_r(t)),\ t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \Rightarrow |f_r(t)| < |t|\} \\
 & (\mathbf{MIN}\, t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p f_r(t) : |f_r(t)| - |x|)
\end{array}$$

$$
\begin{array}{ll}
= & \{\, f_r(t) \in \mathbf{suff}(P),\ x <_p f_r(t) \Rightarrow f_r(t) \neq \varepsilon \,\} \\
 & (\mathbf{MIN}\ t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge f_r(t) \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p f_r(t) : |f_r(t)| - |x|) \\
\geq & \{\, \text{omitting first predicate in domain, change of bound variable: } t' = f_r(t) \,\} \\
 & (\mathbf{MIN}\ t' : t' \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x <_p t' : |t'| - |x|) \\
= & \{\, \text{see first part of the previous derivation} \,\} \\
 & d_1(x)
\end{array}
$$

Since $a = b \,\mathbf{min}\, c \wedge c > a \Rightarrow a = b$ the second operand of the **min**-operator can be omitted in this case as well.

(End of Note)

Summarizing, we have

$$d_1(x) = (\mathbf{MIN}\ t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge x = f_r(t) : |t| - |x|) \qquad (x \in \mathbf{suff}(P)).$$

(cf. [Com79a], sections I and III, and [Com79b], sections II.1 and III, functions `shift1`, `set1`, and `set1'`). Function d_1 can be computed during the computation of γ_r and f_r without having to compute the (generalized) inverse of f_r explicitly.

Before giving an algorithm demonstrating this we will first deal with the computation of function $d_2 : \mathbf{suff}(P) \longrightarrow \mathbb{N}$ defined by

$$d_2(x) = (\mathbf{MIN}\ n : n \geq 1 \wedge V^*P \cap xV^n \neq \varnothing : n) \qquad (x \in \mathbf{suff}(P)).$$

We will show that d_2 can also be expressed in terms of f_r. We distinguish two cases::

i. Let $x = \varepsilon$. We derive

$$
\begin{array}{ll}
 & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*P \cap xV^n \neq \varnothing : n) \\
= & \{\, x = \varepsilon, \text{ property B.2} \,\} \\
 & (\mathbf{MIN}\ n : n \geq 1 \wedge P \cap \mathbf{suff}(V^n) \neq \varnothing : n) \\
= & \{\, \varepsilon \notin P\ (\text{NE}),\ n \geq 1 \Rightarrow \mathbf{suff}(V^n) \setminus \mathbf{suff}(V^{n-1}) = V^n \,\} \\
 & (\mathbf{MIN}\ n : n \geq 1 \wedge P \cap V^n \neq \varnothing : n) \\
= & \{\, \varepsilon \notin P \,\} \\
 & (\mathbf{MIN}\ p : p \in P : |p|)
\end{array}
$$

ii. Let $x \in \mathbf{suff}(P) \setminus \{\varepsilon\}$. We derive

$$
\begin{array}{ll}
 & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*P \cap xV^n \neq \varnothing : n) \\
= & \{\, \text{property B.2} \,\} \\
 & (\mathbf{MIN}\ n : n \geq 1 \wedge P \cap \mathbf{suff}(xV^n) \neq \varnothing : n) \\
= & \{\, x \neq \varepsilon,\ \mathbf{suff}(xV^n) = xV^n + \mathbf{suff}((x\lfloor 1)V^n), \text{ domain split} \,\} \\
 & (\mathbf{MIN}\ n : n \geq 1 \wedge P \cap xV^n \neq \varnothing : n) \\
 & \mathbf{min}(\mathbf{MIN}\ n : n \geq 1 \wedge P \cap \mathbf{suff}((x\lfloor 1)V^n) \neq \varnothing : n) \\
= & \{\, \text{change of bound variable: } n = |s|, \text{ definition of } d_2 \,\} \\
 & (\mathbf{MIN}\ s : s \in V^+ \wedge xs \in P : |s|) \,\mathbf{min}\, d_2(x\lfloor 1) \\
= & \{\, \text{change of bound variable: } p = xs \,\} \\
 & (\mathbf{MIN}\ p : p \in P \wedge x <_p p : |p| - |x|) \,\mathbf{min}\, d_2(x\lfloor 1) \\
= & \{\, \varepsilon \notin P, \text{ hence } p \in \mathbf{suff}(P) \setminus \{\varepsilon\},\ x \in \mathbf{suff}(P), \text{ definition B.9, corollary B.12} \,\} \\
 & (\mathbf{MIN}\ p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : x = f_r^i(p)) : |p| - |x|) \,\mathbf{min}\, d_2(x\lfloor 1)
\end{array}
$$

The result in case **i.** can be made to look more like the result in case **ii.**:

$$\begin{array}{ll} & (\mathbf{MIN}\, p : p \in P : |p|) \\ = & \{ f_r^{\nu(p)}(p) = \varepsilon,\ \varepsilon \notin P, \text{ hence } \nu(p) > 0,\ |\varepsilon| = 0 \} \\ & (\mathbf{MIN}\, p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : \varepsilon = f_r^i(p)) : |p| - |\varepsilon|) \end{array}$$

Summarizing, we have, for $x \in \mathbf{suff}(P) \setminus \{\varepsilon\}$,

$$\begin{array}{lll} d_2(\varepsilon) & = & (\mathbf{MIN}\, p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : \varepsilon = f_r^i(p)) : |p| - |\varepsilon|) \\ d_2(x) & = & (\mathbf{MIN}\, p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : x = f_r^i(p)) : |p| - |x|)\, \mathbf{min}\, d_2(x\lfloor 1). \end{array}$$

(cf. [Com79a], sections I and III, and [Com79b], sections II.1 and III, functions `shift2`, `set2`, and `set2'`; the restriction from `set2` to `set2'` for the computation of `shift2` is not explained there and seems, in view of our results for d_2, to be incorrect).

The expressions derived for d_1 and d_2 lead to the following algorithm computing γ_r, f_r, d_1, and d_2 in program variables gr, fr, $d1$, and $d2$, respectively:

```
for u : u ∈ suff(P) do d1(u), d2(u) := +inf, +inf rof;
for a : a ∈ V do
    if a ∈ suff(P) ⟶
        gr(ε, a) := a;
        fr(a) := ε;
        d1(ε) := d1(ε) min 1;
        if a ∈ P ⟶ d2(ε) := d2(ε) min 1
        [] a ∉ P ⟶ skip
        fi
    [] a ∉ suff(P) ⟶ gr(ε, a) := ε
    fi
rof;
n := 1;
{ invariant:
    (∀u, a : u ∈ suff(P) ∧ |u| < n ∧ a ∈ V : gr(u, a) = γ_r(u, a))
  ∧ (∀u : u ∈ suff(P) \ {ε} ∧ |u| ≤ n : fr(u) = f_r(u))
  ∧ (∀u : u ∈ suff(P) : d1(u) = (MIN t : t ∈ suff(P) \ {ε} ∧ |t| ≤ n ∧ u = f_r(t) : |t| − |u|))
  ∧ (∀u : u ∈ suff(P) : d2(u) = (MIN p : p ∈ P ∧ |p| ≤ n ∧ (∃i : 0 < i ≤ ν(p) : u = f_r^i(p)) : |p| − |u|))
}
do suff(P) ∩ V^n ≠ ∅ ⟶
    for u : u ∈ suff(P) ∩ V^n do
        for a : a ∈ V do
            if au ∈ suff(P) ⟶
                gr(u, a) := au;
                fr(au) := gr(fr(u), a);
                d1(fr(au)) := d1(fr(au)) min(|au| − |fr(au)|);
                if au ∈ P ⟶
                    v := fr(au); i := 1;
                    { invariant: v = fr^i(au) ∧ 0 < i ≤ ν(au) }
                    do v ≠ ε ⟶
                        d2(v) := d2(v) min(|au| − |v|);
                        v := fr(v); i := i + 1
                    od;
                    d2(ε) := d2(ε) min |au|
                [] au ∉ P ⟶ skip
                fi
            [] au ∉ suff(P) ⟶
                gr(u, a) := gr(fr(u), a)
            fi
```

$\quad\quad\quad$ **rof**
$\quad\quad$ **rof**;
$\quad\quad$ $n := n + 1$
od;
$\{ \quad (\forall u, a : u \in \mathbf{suff}(P) \wedge a \in V : gr(u,a) = \gamma_r(u,a))$
$\quad \wedge (\forall u : u \in \mathbf{suff}(P) \setminus \{\varepsilon\} : fr(u) = f_r(u))$
$\quad \wedge (\forall u : u \in \mathbf{suff}(P) : d1(u) = d_1(u))$
$\quad \wedge (\forall u : u \in \mathbf{suff}(P) : d2(u) = (\mathbf{MIN}\ p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : u = f_r^i(p)) : |p| - |u|))$
$\}$
$n := 1$;
$\{$ invariant:
$\quad (\forall u : u \in \mathbf{suff}(P) \wedge |u| < n : d2(u) = d_2(u))$
$\quad \wedge (\forall u : u \in \mathbf{suff}(P) \wedge |u| \geq n : d2(u) = (\mathbf{MIN}\ p : p \in P \wedge (\exists i : 0 < i \leq \nu(p) : u = f_r^i(p)) : |p| - |u|))$
$\}$
do $\mathbf{suff}(P) \cap V^n \neq \varnothing \longrightarrow$
$\quad\quad$ **for** $u : u \in \mathbf{suff}(P) \cap V^n$ **do** $d2(u) := d2(u) \,\mathbf{min}\, d2(u\rfloor 1)$ **rof**;
$\quad\quad$ $n := n + 1$
od

7.2 Computation of d_{no}

Let $x \in \mathbf{suff}(P)$ and $a \in V$. We derive

$$
\begin{array}{ll}
& d_{no}(x,a) \\
= & \quad \{\text{definition of } d_{no}\} \\
& (\mathbf{MIN}\ n : n \geq 1 \wedge ((V^* a V^{n+|x|} \cap P \neq \varnothing \wedge V^* x V^n \cap P \neq \varnothing) \vee V^* P \cap (xV^n) \neq \varnothing) : n) \\
= & \quad \{\text{domain split, definition of } d_2\} \\
& (\mathbf{MIN}\ n : n \geq 1 \wedge V^* a V^{n+|x|} \cap P \neq \varnothing \wedge V^* x V^n \cap P \neq \varnothing : n) \,\mathbf{min}\, d_2(x) \\
= & \quad \{\text{property B.2}\} \\
& (\mathbf{MIN}\ n : n \geq 1 \wedge a V^{n+|x|} \cap \mathbf{suff}(P) \neq \varnothing \wedge x V^n \cap \mathbf{suff}(P) \neq \varnothing : n) \,\mathbf{min}\, d_2(x) \\
= & \quad \{\text{change of bound variable: } |s| = n\} \\
& (\mathbf{MIN}\ s : s \in V^+ \wedge a V^{|xs|} \cap \mathbf{suff}(P) \neq \varnothing \wedge xs \in \mathbf{suff}(P) : |s|) \,\mathbf{min}\, d_2(x) \\
= & \quad \{\text{change of bound variable: } t = xs\} \\
& (\mathbf{MIN}\ t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge a V^{|t|} \cap \mathbf{suff}(P) \neq \varnothing \wedge x <_p t : |t| - |x|) \,\mathbf{min}\, d_2(x) \\
= & \quad \{x, t \in \mathbf{suff}(P),\ t \neq \varepsilon, \text{ corollary B.12, definition of } occ_r \text{ (after derivation)}\} \\
& (\mathbf{MIN}\ t : t \in \mathbf{suff}(P) \setminus \{\varepsilon\} \wedge |t| \in occ_r(a) \wedge (\exists i : 0 < i \leq \nu(t) : x = f_r^i(t)) : |t| - |x|) \\
& \mathbf{min}\, d_2(x)
\end{array}
$$

where $occ_r : V \longrightarrow \mathcal{P}(\mathbb{N})$ is defined by

$$occ_r(a) = \{\, n \mid n \in \mathbb{N} \wedge a V^n \cap \mathbf{suff}(P) \neq \varnothing \} \qquad (a \in V).$$

Observe that occ_r can easily be computed beforehand, e.g. during the computation of τ_r. Thereafter, the computation of the first operand of the **min**-operator is similar to the first part of the computation of d_2. Finally, function d_{no} can be computed during the second and final part of the computation of d_2. We do not give an algorithm here since with these observations the reader may easily adapt the preceding algorithm to also compute d_{no}.

7.3 Computation of d_3

Function $d_3 : \mathbb{N} \times V \longrightarrow \mathbb{N}$ can be expressed in terms of function $\overline{d}_3 : V \longrightarrow \mathbb{N}$, defined by

$$\overline{d}_3(a) = (\mathbf{MIN}\ n : n \geq 1 \wedge V^* a V^n \cap P \neq \varnothing : n) \qquad (a \in V),$$

as follows

$$d_3(z,a) = \begin{cases} +\mathtt{inf} & \text{if } \overline{d}_3(a) = +\mathtt{inf} \\ \overline{d}_3(a) - z & \text{if } \overline{d}_3(a) \neq +\mathtt{inf} \end{cases} \qquad (z \in \mathbb{N}, a \in V).$$

Let $a \in V$. We derive

$$\begin{array}{ll} & \overline{d}_3(a) \\ = & \quad \{\text{definition of } \overline{d}_3\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*aV^n \cap P \neq \varnothing : n) \\ = & \quad \{\text{property B.2}\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge aV^n \cap \mathbf{suff}(P) \neq \varnothing : n) \\ = & \quad \{\text{definition of } occ_r\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge n \in occ_r(a) : n). \end{array}$$

This derivation shows that $\overline{d}_3$ can be computed at the same time as occ_r.

7.4 Computation of d_{bm} and $char$

Let $x \in \mathbf{suff}(P)$. We derive

$$\begin{array}{ll} & d_{bm}(x) \\ = & \quad \{\text{definition of } d_{bm}\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*xV^n \cap V^*P \neq \varnothing : n) \\ = & \quad \{V^*A \cap V^*B \neq \varnothing \equiv V^*A \cap B \neq \varnothing \vee A \cap V^*B \neq \varnothing, \text{ domain split}\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*xV^n \cap P \neq \varnothing : n)\, \mathbf{min} (\mathbf{MIN}\ n : n \geq 1 \wedge xV^n \cap V^*P \neq \varnothing : n) \\ = & \quad \{\text{definition of } d_1 \text{ and } d_2\} \\ & d_1(x)\, \mathbf{min}\, d_2(x). \end{array}$$

Hence, we have

$$d_{bm}(x) = d_1(x)\, \mathbf{min}\, d_2(x) \qquad (x \in \mathbf{suff}(P)),$$

showing that d_{bm} can be computed from d_1 and d_2.

Let $a \in V$. We derive

$$\begin{array}{ll} & char(a) \\ = & \quad \{\text{definition of } char\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*aV^n \cap V^*P \neq \varnothing : n) \\ = & \quad \{V^*A \cap V^*B \neq \varnothing \equiv V^*A \cap B \neq \varnothing \vee A \cap V^+B \neq \varnothing, \text{ domain split}\} \\ & (\mathbf{MIN}\ n : n \geq 1 \wedge V^*aV^n \cap P \neq \varnothing : n)\, \mathbf{min} (\mathbf{MIN}\ n : n \geq 1 \wedge aV^n \cap V^+P \neq \varnothing : n) \\ = & \quad \{\text{definition of } \overline{d}_3,\ P \neq \varnothing,\ \varepsilon \notin P,\ aV^n \cap V^+P \neq \varnothing \equiv n \geq (\mathbf{MIN}\ p : p \in P : |p|)\} \\ & \overline{d}_3(a)\, \mathbf{min} (\mathbf{MIN}\ p : p \in P : |p|) \end{array}$$

Defining

$$m_P = (\mathbf{MIN}\ p : p \in P : |p|)$$

we have

$$char(a) = \overline{d}_3(a)\, \mathbf{min}\, m_P \qquad (a \in V),$$

showing that $char$ can be computed from $\overline{d}_3$.

Having derived expressions for d_{bm} and $char$ in terms of d_1, d_2, and $\overline{d}_3$ we are able to compare the amount of shift for the normal Commentz-Walter algorithm, k_{norm}, to the amount of shift for the Boyer-Moore algorithm, k_{bm}. First, we derive

$$
\begin{array}{ll}
 & char(l\restriction 1) - |v| \\
= & \quad \{ \text{preceding derivation} \} \\
 & (\overline{d}_3(l\restriction 1)\ \mathbf{min}\ m_P) - |v| \\
= & \quad \{ \text{case analysis, } +\mathtt{inf} \text{ unity of } \mathbf{min}\text{, distributivity} \} \\
 & \mathbf{if}\ \overline{d}_3(l\restriction 1) = +\mathtt{inf} \longrightarrow m_P - |v| \\
 & [\!]\ \ \overline{d}_3(l\restriction 1) \neq +\mathtt{inf} \longrightarrow (\overline{d}_3(l\restriction 1) - |v|)\ \mathbf{min}(m_P - |v|) \\
 & \mathbf{fi} \\
= & \quad \{ \text{relation between } \overline{d}_3 \text{ and } d_3 \} \\
 & d_3(|v|, l\restriction 1)\ \mathbf{min}(m_P - |v|).
\end{array}
$$

Next, we derive

$$
\begin{array}{ll}
 & k_{bm} \\
= & \quad \{ \text{definition of } k_{bm} \} \\
 & ((char(l\restriction 1) - |v|)\ \mathbf{max}\ d_{bm}(v))\ \mathbf{min}\ |r| \\
= & \quad \{ \text{preceding derivation, } d_{bm} \text{ expressed in } d_1 \text{ and } d_2 \} \\
 & ((d_3(|v|, l\restriction 1)\ \mathbf{min}(m_P - |v|))\ \mathbf{max}(d_1(v)\ \mathbf{min}\ d_2(v)))\ \mathbf{min}\ |r| \\
= & \quad \{ \text{distributivity} \} \\
 & ((d_3(|v|, l\restriction 1)\ \mathbf{min}(m_P - |v|))\ \mathbf{max}\ d_1(v)) \\
 & \mathbf{min}((d_3(|v|, l\restriction 1)\ \mathbf{min}(m_P - |v|))\ \mathbf{max}\ d_2(v))\ \mathbf{min}\ |r| \\
= & \quad \{ (\forall n : 1 \leq n < m_P - |v| : V^*P \cap vV^n = \varnothing)\text{, definition of } d_2\text{, hence } m_P - |v| \leq d_2(v) \} \\
 & ((d_3(|v|, l\restriction 1)\ \mathbf{min}(m_P - |v|))\ \mathbf{max}\ d_1(v))\ \mathbf{min}\ d_2(v)\ \mathbf{min}\ |r| \\
\leq & \quad \{ \text{calculus} \} \\
 & (d_3(|v|, l\restriction 1)\ \mathbf{max}\ d_1(v))\ \mathbf{min}\ d_2(v)\ \mathbf{min}\ |r| \\
= & \quad \{ \text{definition of } k_{norm} \} \\
 & k_{norm},
\end{array}
$$

showing that the amount of shift in the normal Commentz-Walter algorithm is at least the amount of shift in the Boyer-Moore algorithm.

7.5 Precomputation of s_1, $char_1$, and $char_2$

Here we discuss the precomputation of functions s_1, $char_1$, and $char_2$ for the variants of the one keyword Boyer-Moore algorithm obtained by instantiating detail (MO) by (FWD) and (REV), respectively.

7.5.1 Forward matching

In the forward matching scheme (algorithm detail (FWD)) we have $mo(i) = i$. In this case P_3 can be manipulated further:

$$
\begin{array}{ll}
 & P_3(v, i, k) \\
\equiv & \quad \{ \text{definition of } P_3 \text{ and } mo \} \\
 & (\forall j : 1 \leq j < i \wedge 1 + k \leq j : p_j = p_{j-k}) \\
 & \wedge\ (i \leq |p| \wedge 1 + k \leq i \Rightarrow v_i = p_{i-k} \wedge p_i \neq p_{i-k}) \\
\equiv & \quad \{ \text{simplifying ranges, } 1 \leq i \leq |p| + 1 \} \\
 & (\forall j : 1 + k \leq j < i : p_j = p_{j-k}) \\
 & \wedge\ (1 + k \leq i \leq |p| \Rightarrow v_i = p_{i-k} \wedge p_i \neq p_{i-k})
\end{array}
$$

We continue with only the first conjunct, assuming $1 + k \leq i$:

$$
\begin{array}{ll}
 & (\forall j : 1+k \leq j < i : p_j = p_{j-k}) \\
\equiv & \qquad \{ \text{rewrite using } \upharpoonleft \text{ and } \downharpoonleft \} \\
 & (p\upharpoonleft(i-1))\downharpoonleft k = p\upharpoonleft(i-1-k) \\
\equiv & \qquad \{ \text{set calculus} \} \\
 & \{(p\upharpoonleft(i-1))\downharpoonleft k\} \cap \{p\upharpoonleft(i-1-k)\} \neq \varnothing \\
\equiv & \qquad \{ \{x\downharpoonleft j\} \cap Y \neq \varnothing \equiv \{x\} \cap V^j Y \neq \varnothing \ (0 \leq j \leq |x|),\ k \leq i-1 \} \\
 & \{p\upharpoonleft(i-1)\} \cap V^k(p\upharpoonleft(i-1-k)) \neq \varnothing \\
\equiv & \qquad \{ X \cap Y(x\upharpoonleft j) \neq \varnothing \equiv XV^{|x|-j} \cap Yx \neq \varnothing \ (0 \leq j \leq |x|),\ k \leq i-1,\ i \leq |p|+1 \} \\
 & (p\upharpoonleft(i-1))V^{|p|+k-i+1} \cap V^k p \neq \varnothing \\
\equiv & \qquad \{ (\forall x, y : x \in X \wedge y \in Y : |x| = |y|) \Rightarrow (X \cap Y \neq \varnothing \equiv V^* X \cap V^* Y \neq \varnothing) \} \\
 & V^*(p\upharpoonleft(i-1))V^{|p|+k-i+1} \cap V^* p \neq \varnothing
\end{array}
$$

Notice that this predicate is similar to the predicate in the definition of function d_{bm} (see subsection 4.4). Precomputation of functions s_1, $char_1$, and $char_2$ is similar to the precomputation for the Boyer-Moore variant derived from the Commentz-Walter algorithm (Part II, subsection 7.4). For this reason we do not elaborate the precomputation any further.

7.5.2 Backward matching

With backward matching (algorithm detail (REV)), p is compared to v from right to left, i.e. we have $mo(i) = |p| - i + 1$, the reverse permutation of the integers from 1 to $|p|$. Predicate P_3 can be manipulated further. We have

$$
\begin{array}{ll}
 & P_3(v, i, k) \\
\equiv & \qquad \{ \text{definition of } P_3 \text{ and } mo \} \\
 & (\forall j : 1 \leq j < i \wedge j \leq |p| - k : p_{|p|-j+1} = p_{|p|-j-k+1}) \\
 & \wedge\, (i \leq |p| - k \Rightarrow v_{|p|-i+1} = p_{|p|-i-k+1} \wedge p_{|p|-i+1} \neq p_{|p|-i-k+1})
\end{array}
$$

We concentrate on the first conjunct and distinguish three cases. If $i \leq |p| - k$ the first conjunct becomes

$$
\begin{array}{ll}
 & (\forall j : 1 \leq j < i : p_{|p|-j+1} = p_{|p|-j-k+1}) \\
\equiv & \qquad \{ \text{rewrite using } \upharpoonright \text{ and } \downharpoonright \} \\
 & p\upharpoonright(i-1) = (p\upharpoonright(k+i-1))\downharpoonright k \\
\equiv & \qquad \{ \text{set calculus} \} \\
 & \{p\upharpoonright(i-1)\} \cap \{(p\upharpoonright(k+i-1))\downharpoonright k\} \neq \varnothing \\
\equiv & \qquad \{ X \cap \{y\downharpoonright j\} \neq \varnothing \equiv XV^j \cap \{y\} \neq \varnothing \ (0 \leq j \leq |y|),\ i+k \leq |p| \} \\
 & (p\upharpoonright(i-1))V^k \cap \{p\upharpoonright(k+i-1)\} \neq \varnothing \\
\equiv & \qquad \{ X \cap \{y\upharpoonright j\} \neq \varnothing \equiv V^{|y|-j} X \cap \{y\} \neq \varnothing \ (0 \leq j \leq |y|),\ k+i-1 < |p| \} \\
 & V^{|p|-k-i+1}(p\upharpoonright(i-1))V^k \cap \{p\} \neq \varnothing \\
\equiv & \qquad \{ (\forall x, y : x \in X \wedge y \in Y : |x| = |y|) \Rightarrow (X \cap Y \neq \varnothing \equiv V^* X \cap V^* Y \neq \varnothing) \} \\
 & V^*(p\upharpoonright(i-1))V^k \cap V^* p \neq \varnothing
\end{array}
$$

If $i > |p| - k$ and $k \leq |p|$ the first conjunct becomes

$$
\begin{array}{ll}
 & (\forall j : 1 \leq j \leq |p| - k : p_{|p|-j+1} = p_{|p|-j-k+1}) \\
\equiv & \qquad \{ \text{rewrite using } \downharpoonleft \text{ and } \downharpoonright \} \\
 & p\downharpoonleft k = p\downharpoonright k \\
\equiv & \qquad \{ \text{set calculus} \} \\
 & \{p\downharpoonleft k\} \cap \{p\downharpoonright k\} \neq \varnothing
\end{array}
$$

$$\equiv \quad \{\, \{x{\downarrow}j\} \cap Y \neq \varnothing \equiv \{x\} \cap V^j Y \neq \varnothing \ (0 \le j \le |x|),\ k \le |p| \,\}$$
$$\{p\} \cap V^k (p{\downarrow}k) \neq \varnothing$$
$$\equiv \quad \{\, X \cap Y(y{\downarrow}j) \neq \varnothing \equiv XV^j \cap Yy \neq \varnothing \ (0 \le j \le |y|),\ k \le |p| \,\}$$
$$pV^k \cap V^k p \neq \varnothing$$
$$\equiv \quad \{\, (\forall x, y : x \in X \wedge y \in Y : |x| = |y|) \Rightarrow (X \cap Y \neq \varnothing \equiv V^* X \cap V^* Y \neq \varnothing) \,\}$$
$$V^* p V^k \cap V^* p \neq \varnothing$$

If $i > |p| - k$ and $k > |p|$ the first conjunct holds by definition. Notice that in this case $V^* p V^k \cap V^* p \neq \varnothing$ holds as well, so the last two cases can combined. From these derivations and the definition of d_{bm} (see subsection 4.4, $P = \{p\}$) it follows that $s_1(i) = d_{bm}(p{\upharpoonright}(i{-}1))$ for $i \ge 1$. Notice that $p{\upharpoonright}(i-1) \in \mathbf{suff}(P)$. Precomputation of function s_1 is therefore equal to the precomputation of d_{bm} (see Part II, subsection 7.4).

In a similar way one can derive

$$char_1(v, i) = (\mathbf{MIN}\ k : i \le k \wedge V^* v_{|p|-i+1} V^k \cap V^* p \neq \varnothing : k) - (i-1)$$

in which the quantified expression can approximated from below by $char(v_{|p|-i+1})$ (see subsection 4.4, $P = \{p\}$) by enlarging the range to $1 \le k$. Precomputation of $char_1$ is similar to the precomputation of $char$ (see Part II, subsection 7.4).

The expression for $char_2(i)$ becomes

$$(\mathbf{MIN}\ k : i \le k \le |p| - 1 \wedge V^* p_{|p|-i+1} V^k \cap V^* p = \varnothing : k - i + 1)\, \mathbf{min}(|p| - i + 1)$$

Equivalence

$$V^* p_{|p|-i+1} V^k \cap V^* p = \varnothing \equiv V^*(V \setminus \{p_{|p|-i+1}\}) V^k \cap V^* p \neq \varnothing$$

indicates that the precomputation of $char_2$ is analogous to the precomputation of $char_1$ and $char$.

Part III
Conclusions

The taxonomy presented in Parts I and II has achieved the goals set out in the introduction. The highlights of this taxonomy fall into two categories: general results of the derivation method and specific results of the taxonomy. The general results can be summarized as:

- The method of refinement used in each of the derivations presented the algorithms in an abstract, easily digested format. This presentation allows a correctness proof of an algorithm to be developed simultaneously with the algorithm itself.

- The presentation method proves to be more than just a method of deriving algorithms: the derivations themselves serve in the classification (in the taxonomy) of the algorithms. This is accomplished by dividing the derivation at points which involve the introduction of either problem or algorithm details. A sequence of such details serves to identify an algorithm. By prefix-factoring these sequences, common parts of two algorithm derivations can be presented simultaneously.

- The taxonomy of all algorithms considered can be depicted as a graph (in our particular case a tree); the root represents the original solution $O := (\cup\, l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\})$, edges represent the addition of a detail, and the internal vertices and leaves represent derived algorithms. This graph is shown in Figure 1. The utility of this graph is that it can be used as an "alternative table of contents" to the taxonomy. Being interested in only a subset of the algorithms, for example the Aho-Corasick (AC) algorithms, does not necessitate reading all of the derivations; only the root-leaf paths that lead to the AC algorithms need to be read for a complete view of these algorithms.

- The presentation was also more than just a taxonomy. Instead of using completed derivations of known algorithms, which are possibly in different styles of derivation, all of the algorithms were derived in a common framework. This made it easy to see what the algorithms have in common (or where they differ) for the purposes of classifying them.

- The pattern matching overview presented in [Aho90] is an excellent introduction to many of the algorithms presented in this paper. Unfortunately, it does not present all variants of the algorithms, or present them in a fashion that allows one to contrast the algorithms with one another. Our taxonomy accomplished precisely this goal, of presenting algorithms in one framework for comparison. In deriving the algorithms for this taxonomy every attempt was made to thoroughly explore all of the possible variants. Our taxonomy should be a thorough introduction to all variants of the four principal pattern matching algorithms presented in [Aho90].

Results concerning particular algorithms can be summarized as follows:

- As stated in [AC75], the AC algorithm is intended to be a generalization of the original Knuth-Morris-Pratt (KMP) algorithm — making use of automata theory. The classical derivations of the two (using automata and indices, respectively) do not serve to highlight their similarities, or differences.

  When derived in the same framework, it becomes apparent that the AC algorithm cannot be specialized to arrive at KMP; this can be seen from the derivation of the AC algorithm subtree of the taxonomy tree. The linear search (introduced in subsection 3.3) used in the failure function AC algorithm (algorithm 3.4) is quite different from the linear search used in the abstract KMP algorithm (algorithm 3.5). Indices could have been introduced in algorithm 3.4, although this does not yield the classically presented KMP algorithm. The AC-KMP relationship is in fact that they have a common ancestor algorithm (P_+, E, AC, LS).

- The abstract intermediate KMP algorithm (algorithm 3.5) is in fact a new algorithm, albeit a variant of the AC algorithm. The running time of this new algorithm does not appear to be any better than algorithm 3.4. The transformation (by adding indices) of algorithm 3.5 into the classically presented KMP algorithm (algorithm 3.6) was demonstrated to be straightforward.

- The original Aho-Corasick article [AC75] presented the "optimal" version of the algorithm after the failure function version of the algorithm. The optimal algorithm was explained as using a transition function γ_f which is a composition of the extended forward trie τ_{ef} and failure function f_f. While this is indeed the case, our derivation proceeded much more smoothly by deriving an algorithm which is a common ancestor of both the optimal and the failure function algorithms.

- "Predicate weakening" (of sections 4 and 5) was instrumental in deriving various algorithms (and their correctness proofs) from the Commentz-Walter (CW) algorithm, in particular the Boyer-Moore (BM) algorithm. The CW algorithm has not emerged as a popular string pattern matching algorithm partly due to the difficulty in understanding it. The derivation presented in Part I arrives at the CW algorithm through a series of small transformations, starting with a naive (quadratic running time) algorithm. This derivation makes the CW algorithm considerably easier to understand. Predicate weakening was also heavily used in deriving the "match-order" variant of the BM algorithm.

- Commentz-Walter's intention was to combine the BM algorithm with automata theory, to produce an algorithm dealing with multiple keywords. The relationship between the two algorithms has previously remained obscured by the styles of presentation of the two algorithms (indices in BM, and automata in CW). As seen from section 4 the BM algorithm can indeed be arrived at in the same framework (as the CW algorithm) as a special case. The publication of the Hume-Sunday taxonomy [HS91] motivated us to also derive the BM algorithm in an entirely different manner — making use of the concept of "match-orders".

- In both papers by Commentz-Walter describing her algorithm (in particular [Com79a]), the differences between methods of determining a safe shift amount were not made explicit. Indeed, that some of these shift functions were distinct was not mentioned in all cases. Our derivation of the CW algorithm clearly defines the differences between the shift functions. The (NEAR-OPT) shift function was only mentioned in passing in the original paper; this derivation provides a definition of this function; Part II provides the only full derivation of a precomputation algorithm for this function.

- In the BM algorithm the functions contributing to a shift have been presented in several separate papers since the introduction of the original algorithm. Until the publication of the taxonomy by [HS91] it was difficult to examine the contribution of each shift function. Both section 5 and [HS91] present a shift as consisting of components that can be readily replaced by an equivalent component, for example: the "skip" loops, or the "match-orders". [HS91] emphasized effects on running-time of each component. Our taxonomy has emphasized the derivation of each of these components from a common specification.

- The precomputation of the BM shift functions has been troublesome; many solutions were published, corrected, and re-published (for a good bibliography of these see [Aho90]). The precomputation presented in Part II provides an understandable derivation of a correct precomputation algorithm.

Part IV
Appendices

A Calculating the value of a quantification

The problem is, given an associative, commutative operator $\oplus$ on set U with unit $e_\oplus$, a set W, a range predicate $RANGE : W \longrightarrow \mathbb{B}$, and a function $f : W \longrightarrow U$ on W, calculate:

$$w = (\oplus x \in W : RANGE(x) : f(x))$$

We now present three solutions.

A.1 A nondeterministic solution

This can be done with the following nondeterministic repetition:

$RW := \{x \mid x \in W \land RANGE(x)\}$; $w := e_\oplus$;
for $x : x \in RW$ **do** $w := w \oplus f(x)$ **rof**
$\{w = (\oplus x \in W : RANGE(x) : f(x))\}$

A.2 A deterministic solution in the ascending direction

Given the set $RW = \{x \mid x \in W \land RANGE(x)\}$ and a linear order $\leq$ on RW we can define a function $next : RW \longrightarrow (RW \cup \{\top\})$ as:

$$next(v) = (\mathbf{MIN}_\leq\, x \in RW : v < x : x)$$

Function $next$ is extended to map the maximum element of RW to fictitious element $\top$ (to make $next$ total). Assume $RW \neq \emptyset$.

This allows us to implement a deterministic algorithm which processes RW in $\leq$-ascending order:

$v := (\mathbf{MIN}_\leq\, x \in RW :: x)$; $w := f(v)$;
{invariant: $w \oplus (\oplus x \in RW : v < x : f(x)) = (\oplus x \in W : RANGE(x) : f(x))$ }
do $next(v) \neq \top \longrightarrow$
    $v := next(v)$;
    $w := w \oplus f(v)$
od
$\{w = (\oplus x \in W : RANGE(x) : f(x))\}$

A.3 A deterministic solution in the descending direction

Given the set RW defined above in Appendix A.2, we define a function $prev : RW \longrightarrow (RW \cup \{\perp\})$ as:

$$prev(v) = (\mathbf{MAX}_\leq x \in RW : x < v : x)$$

extended to map the minimum element in RW to $\perp$. We can now implement a deterministic algorithm which processes RW in $\leq$-descending order. Assume $RW \neq \emptyset$. The following algorithm is symmetrical to that presented above in Appendix A.2, with the exception that the repetition is phase shifted, leaving an additional assignment after the repetition:

$v := (\mathbf{MAX}_{\leq}\ x \in RW :: x);\ w := e_{\oplus};$
$\{\text{invariant: } w \oplus (\oplus x \in RW : x \leq v : f(x)) = (\oplus x \in W : RANGE(x) : f(x))\ \}$
do $prev(v) \neq \bot \longrightarrow$
  $w := w \oplus f(v);$
  $v := prev(v)$
od;
$w := w \oplus f(v)$
$\{w = (\oplus x \in W : RANGE(x) : f(x))\}$

A.4 Nested quantifications

Nested quantifications can similarly be dealt with using nested repetitions. When two operators of nested quantifications are in fact the same, the accumulation variable (in the above programs w) of the two corresponding nested repetitions can be identified. This is useful in our case, where most of the quantifications will consist of two nested union quantifications.

For example, given the requirement to compute:

$$w = (\oplus x \in W : RANGE(x) : (\oplus y \in W' : RANGE'(x, y) : f(y)))$$

we can make the following first nondeterministic solution[13]:

$RW := \{x \mid x \in W \wedge RANGE(x)\};\ w := e_{\oplus};$
for $x : x \in RW$ **do**
  $RW' := \{y \mid y \in W' \wedge RANGE'(x, y)\};\ w' := e_{\oplus};$
  for $y : y \in RW'$ **do** $w' := w' \oplus f(y)$ **rof**;
  $\{w' = (\oplus y \in W' : RANGE'(x, y) : f(y))\}$
  $w := w \oplus w'$
rof
$\{w = (\oplus x \in W : RANGE(x) : (\oplus y \in W' : RANGE'(x, y) : f(y)))\}$

The program variable w' in the inner repetition is not needed, and w can instead be updated directly. The (slightly) shortened version is now:

$RW := \{x \mid x \in W \wedge RANGE(x)\};\ w := e_{\oplus};$
for $x : x \in RW$ **do**
  $RW' := \{y \mid y \in W' \wedge RANGE'(x, y)\};$
  for $y : y \in RW'$ **do** $w := w \oplus f(y)$ **rof**
rof
$\{w = (\oplus x \in W : RANGE(x) : (\oplus y \in W' : RANGE'(x, y) : f(y)))\}$

B Definitions and properties

This section provides a series of definitions and properties which are used throughout this paper.

For any language L, we take L^R to denote the reversal of the language. For a string $w \in V^*$, we take w^R to denote the reversal of w.

Definition B.1 *Let V be an alphabet. Define* $\mathbf{pref} : \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ *and* $\mathbf{suff} : \mathcal{P}(V^*) \longrightarrow \mathcal{P}(V^*)$ *as* $\mathbf{pref}(L) = \{w \mid w \in V^* \wedge (\exists x : x \in V^* : wx \in L)\}$ *and* $\mathbf{suff}(L) = (\mathbf{pref}(L^R))^R$. □

[13] The deterministic solution follows similarly.

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

Property B.2 *Let* $A, B \subseteq V^*$. *Then* $\mathbf{pref}(A) \cap B \neq \emptyset \equiv A \cap BV^* \neq \emptyset$ *and* $\mathbf{suff}(A) \cap B \neq \emptyset \equiv A \cap V^*B \neq \emptyset$. □

The following two theorems are used in the derivation of the Aho-Corasick precomputation algorithm.

Theorem B.3 *Let* V *be an alphabet,* $A, B, C \subseteq V^*$, *and* $V^*C \cap B = BC \cap B$. *Then*

$$\mathbf{suff}(A)C \cap B = \mathbf{suff}(\mathbf{suff}(A) \cap B)C \cap B.$$

Proof

$$\begin{array}{ll}
 & \mathbf{suff}(A)C \cap B \\
= & \quad \{\, \mathbf{suff}(A) \subseteq V^*, \text{ distributivity} \,\} \\
 & \mathbf{suff}(A)C \cap V^*C \cap B \\
= & \quad \{\, V^*C \cap B = BC \cap B \,\} \\
 & \mathbf{suff}(A)C \cap BC \cap B \\
= & \quad \{\, \text{distributivity} \,\} \\
 & (\mathbf{suff}(A) \cap B)C \cap B \\
\subseteq & \quad \{\, X \subseteq \mathbf{suff}(X) \text{ for all } X \subseteq V^*, \text{ monotonicity} \,\} \\
 & \mathbf{suff}(\mathbf{suff}(A) \cap B)C \cap B \\
\subseteq & \quad \{\, \mathbf{suff}(A) \cap B \subseteq \mathbf{suff}(A), \text{ monotonicity} \,\} \\
 & \mathbf{suff}(\mathbf{suff}(A))C \cap B \\
= & \quad \{\, \mathbf{suff} \text{ is idempotent, since } \leq_s \text{ is transitive} \,\} \\
 & \mathbf{suff}(A)C \cap B
\end{array}$$

□

If $C = \{\varepsilon\}$ or $B = \mathbf{pref}(B)$ then condition $V^*C \cap B = BC \cap B$ is satisfied.

Definition B.4 *Define the relations* $\leq_p$ *and* $\leq_s$ *over* $V^* \times V^*$ *as* $u \leq_p v \equiv u \in \mathbf{pref}(v)$ *and* $u \leq_s v \equiv u \in \mathbf{suff}(v)$. □

Theorem B.5 *Let* V *be an alphabet,* $A, B, C \subseteq V^*$, $V^*C \cap B = BC \cap B$, *and* A *is nonempty, finite, and linearly ordered with respect to* $\leq_s$. *Then*

$$\mathbf{suff}(A)C \cap B = \mathbf{suff}((\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(A) \cap B : w))C \cap B.$$

Proof

$$\begin{array}{ll}
 & \mathbf{suff}(A)C \cap B \\
\supseteq & \quad \{\, (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(A) \cap B : w) \in \mathbf{suff}(A), \text{ monotonicity}, A \neq \emptyset \,\} \\
 & \mathbf{suff}((\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(A) \cap B : w))C \cap B \\
\supseteq & \quad \{\, \mathbf{suff}(A) \cap B \leq_s (\mathbf{MAX}_{\leq_s}\, w : w \in \mathbf{suff}(A) \cap B : w), \text{ monotonicity} \,\} \\
 & \mathbf{suff}(\mathbf{suff}(A) \cap B)C \cap B \\
= & \quad \{\, \text{Theorem B.3} \,\} \\
 & \mathbf{suff}(A)C \cap B
\end{array}$$

□

Definition B.6 *The infix operators* $\upharpoonleft, \downharpoonright, \upharpoonright, \downharpoonleft : V^* \times \mathbb{N} \longrightarrow V^*$ *are defined by*

$$\begin{array}{rcll}
v\upharpoonleft 0 & = & \varepsilon & (v \in V^*) \\
\varepsilon\upharpoonleft(k+1) & = & \varepsilon & (k \geq 0) \\
(aw)\upharpoonleft(k+1) & = & a(w\upharpoonleft k) & (k \geq 0, a \in V, w \in V^*) \\
v\downharpoonright 0 & = & v & (v \in V^*) \\
\varepsilon\downharpoonright(k+1) & = & \varepsilon & (k \geq 0) \\
(aw)\downharpoonright(k+1) & = & w\downharpoonright k & (k \geq 0, a \in W, w \in V^*)
\end{array}$$

Define $\upharpoonright$ *as* $v\upharpoonright k = (v^R\upharpoonleft k)^R$ *and* $\downharpoonleft$ *as* $v\downharpoonleft k = (v^R\downharpoonright k)^R$. *The operators* $\upharpoonleft, \downharpoonright, \upharpoonright$, *and* $\downharpoonleft$ *are called "left take," "left drop," "right take," and "right drop" respectively.* □

For $A \subseteq V^*$ and $k \geq 0$ we define $A\upharpoonleft k = (\cup\, w : w \in A : w\upharpoonleft k)$ and $A\downharpoonright k = (\cup\, w : w \in A : w\downharpoonright k)$, and likewise for $\upharpoonright$ and $\downharpoonleft$.

Property B.7 *Let* V *be an alphabet,* $A, B \subseteq V^*$, $A \neq \emptyset$, *and* $\varepsilon \notin A$. *Then*

$$V^*A \cap B \neq \emptyset \vee V^*B \cap A \neq \emptyset \Rightarrow V^*A \cap B \neq \emptyset \vee V^*B \cap (A\downharpoonright 1) \neq \emptyset$$

Proof

$$\begin{array}{ll}
 & V^*A \cap B \neq \emptyset \vee V^*B \cap A \neq \emptyset \\
\equiv & \quad \{\text{split second disjunct: } V^* = V^+ \cup \{\varepsilon\}\} \\
 & V^*A \cap B \neq \emptyset \vee B \cap A \neq \emptyset \vee V^+B \cap A \neq \emptyset \\
\Rightarrow & \quad \{A \subseteq (A\upharpoonleft 1)(A\downharpoonright 1);\ B \cap A \neq \emptyset \Rightarrow V^*A \cap B \neq \emptyset\} \\
 & V^*A \cap B \neq \emptyset \vee VV^*B \cap (A\upharpoonleft 1)(A\downharpoonright 1) \neq \emptyset \\
\Rightarrow & \quad \{(A\upharpoonleft 1) \subseteq V\} \\
 & V^*A \cap B \neq \emptyset \vee VV^*B \cap V(A\downharpoonright 1) \neq \emptyset \\
\equiv & \quad \{\text{left factoring of } V\} \\
 & V^*A \cap B \neq \emptyset \vee V^*B \cap (A\downharpoonright 1) \neq \emptyset
\end{array}$$

□

We continue with some properties of the failure function that are used in the derivation of the Commentz-Walter precomputation algorithm.

Lemma B.8 *For* $x, y \in \textbf{suff}(P)$ *and* $y \neq \varepsilon$ *we have*

$$x <_p y \equiv x \leq_p f_r(y).$$

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

$$\begin{array}{ll}
 & x <_p y \\
\equiv & \quad \{\text{definition of } <_p \text{ and } \textbf{pref}\} \\
 & x \in \textbf{pref}(y) \setminus \{y\} \\
\equiv & \quad \{x \in \textbf{suff}(P)\} \\
 & x \in \textbf{pref}(y) \setminus \{y\} \cap \textbf{suff}(P) \\
\Rightarrow & \quad \{\textbf{pref}(y) \setminus \{y\} \cap \textbf{suff}(P) \text{ is finite and linearly ordered w.r.t. } \leq_p\} \\
 & x \leq_p (\textbf{MAX}_{\leq_p}\, w : w \in \textbf{pref}(y) \setminus \{y\} \cap \textbf{suff}(P) : w) \\
\equiv & \quad \{y \neq \varepsilon, \text{ definition of } f_r\} \\
 & x \leq_p f_r(y) \\
\Rightarrow & \quad \{y \neq \varepsilon,\ f_r(y) <_p y \text{ (by definition of } f_r\text{), transitivity of } <_p\} \\
 & x <_p y
\end{array}$$

□

Definition B.9 *We define* $\nu : \mathbf{suff}(P) \longrightarrow \mathbb{N}$ *by*

$$\nu(\varepsilon) = 0$$

and

$$\nu(y) = \nu(f_r(y)) + 1 \qquad\qquad (y \in \mathbf{suff}(P) \setminus \{\varepsilon\}).$$

□

Property B.10 *We have for all* $y \in \mathbf{suff}(P) \setminus \{\varepsilon\}$

$$f_r^{\nu(y)}(y) = \varepsilon \wedge (\forall n : 0 \leq n < \nu(y) : f_r^n(y) \neq \varepsilon).$$

□

Lemma B.11 *For* $x, y \in \mathbf{suff}(P)$ *and* $y \neq \varepsilon$ *we have*

$$(\forall n : 0 \leq n \leq \nu(y) : x <_p y \equiv (\exists i : 0 < i \leq n : x = f_r^i(y)) \vee x <_p f_r^n(y))$$

Proof
Let $x, y \in \mathbf{suff}(P)$ and $y \neq \varepsilon$. We proceed by induction on n.

base Let $n = 0$. Observe that $\nu(y) > 0 = n$. The equivalence is satisfied trivially.

step Let $n = k + 1$ for some $k : 0 \leq k < \nu(y)$. Assume

$$x <_p y \equiv (\exists i : 0 < i \leq k : x = f_r^i(y)) \vee x <_p f_r^k(y)$$

We derive

$$\begin{array}{ll} & x <_p y \\ \equiv & \quad \{\text{ induction hypothesis }\} \\ & (\exists i : 0 < i \leq k : x = f_r^i(y)) \vee x <_p f_r^k(y) \\ \equiv & \quad \{\, 0 \leq k < \nu(y), \text{ hence by property B.10 } f_r^k(y) \neq \varepsilon, \text{ lemma B.8} \,\} \\ & (\exists i : 0 < i \leq k : x = f_r^i(y)) \vee x \leq_p f_r^{k+1}(y) \\ \equiv & \quad \{\, x \leq_p f_r^{k+1}(y) \equiv x = f_r^{k+1}(y) \vee x <_p f_r^{k+1}(y) \,\} \\ & (\exists i : 0 < i \leq k+1 : x = f_r^i(y)) \vee x <_p f_r^{k+1}(y) \end{array}$$

□

By instantiating n with $\nu(y)$ in this lemma we obtain

Corollary B.12 *For* $x, y \in \mathbf{suff}(P)$ *and* $y \neq \varepsilon$ *we have*

$$x <_p y \equiv (\exists i : 0 < i \leq \nu(y) : x = f_r^i(y))$$

□

References

[Aho90] Aho, A.V. Algorithms for finding patterns in strings, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, vol. A* (North-Holland, Amsterdam, 1990) 257–300.

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

[AHU74] Aho, A.V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1974).

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

[Bro83] Broy, M. Program construction by transformations: a family tree of sorting programs, in: A.W. Biermann and G Guiho, eds., *Computer Program Synthesis Methodologies* (1983) 1–49.

[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.

[Dar78] Darlington, J. A synthesis of several sorting algorithms. *Acta Informatica,* 11 (1978) 1–30.

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

[vdE92] van den Eijnde, J.P.H.W. Program derivation in acyclic graphs and related problems, Computing Science Notes 92/04, Eindhoven University of Technology, The Netherlands, 1992.

[Fre60] Fredkin, E. Trie memory, *Comm. ACM,* 3(9) (1960) 490–499.

[HU79] Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata, Theory, Languages, and Computation* (Addison-Wesley Publishing Company, Reading, MA, 1979).

[HS91] Hume, S.C. and D. Sunday. Fast string searching, *Software—Practice and Experience,* 21 (11) (1991) 1221–1248.

[Jon82] Jonkers, H.B.M. Abstraction, specification and implementation techniques, Dissertation, Eindhoven University of Technology, The Netherlands, 1982; also MC-Tract 166, Mathematical Center, Amsterdam, The Netherlands, 1983.

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

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

[Per90] Perrin, D. Finite Automata, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, vol. B* (North-Holland, Amsterdam, 1990) 1–57.

[RS59] Rabin, M.O. and D. Scott. Finite automata and their decision problems, *IBM Journal of Research* 3(2) (1959) 115–125.

[Ryt80] Rytter, W. A correct preprocessing algorithm for Boyer-Moore string-searching, *SIAM J. Comput.* 9(2) (1980) 509–512.