A new taxonomy of sublinear keyword pattern matching algorithms

A new taxonomy of sublinear keyword pattern matching algorithms

Loek Cleophas, Bruce W. Watson & Gerard Zwaan

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
loek@loekcleophas.com bruce@bruce-watson.com g.zwaan@tue.nl

April 22, 2004

Abstract

This paper presents a new taxonomy of sublinear (multiple) keyword pattern matching algorithms. Based on an earlier taxonomy by Watson and Zwaan [WZ96, WZ95], this new taxonomy includes not only suffix-based algorithms related to the Boyer-Moore, Commentz-Walter and Fan-Su algorithms, but factor- and factor oracle-based algorithms such as Backward DAWG Matching and Backward Oracle Matching as well. In particular, we show how suffix-based (Commentz-Walter like), factor- and factor oracle-based sublinear keyword pattern matching algorithms can all be seen as instantiations of a general sublinear algorithm skeleton. In addition, we show all shift functions defined for the suffix-based algorithms to be in principle reusable for factor- and factor oracle-based algorithms. The taxonomy is based on deriving the algorithms from a common starting point by adding algorithm and problem details, in order to arrive at efficient or well-known algorithms. Such a presentation provides correctness arguments for the algorithms as well as clarity on how the algorithms are related to one another. In addition, it is helpful in the construction of a toolkit of the algorithms.

Contents

1 Introduction **3**
1.1 Related work . . . 3
1.2 Taxonomy construction . . . 3
1.3 Notation used . . . 5
1.4 Overview . . . 5

2 The problem and some high-level solutions **6**
2.1 A change leading to smaller automata . . . 9
2.2 A generalized sublinear algorithm skeleton . . . 11

3 Suffix-based sublinear pattern matching **13**
3.1 No lookahead at the unscanned part of the input string . . . 14
3.2 Restriction to one symbol lookahead . . . 14
3.3 Lookahead symbol is mismatching . . . 15
3.4 The multiple keyword Boyer-Moore algorithm . . . 17
3.5 The Commentz-Walter algorithm . . . 17
3.6 Complete decoupling of recognized suffix and lookahead symbol . . . 18
3.7 Discarding the lookahead symbol . . . 18
3.8 The multiple-keyword Boyer-Moore-Horspool algorithm . . . 19

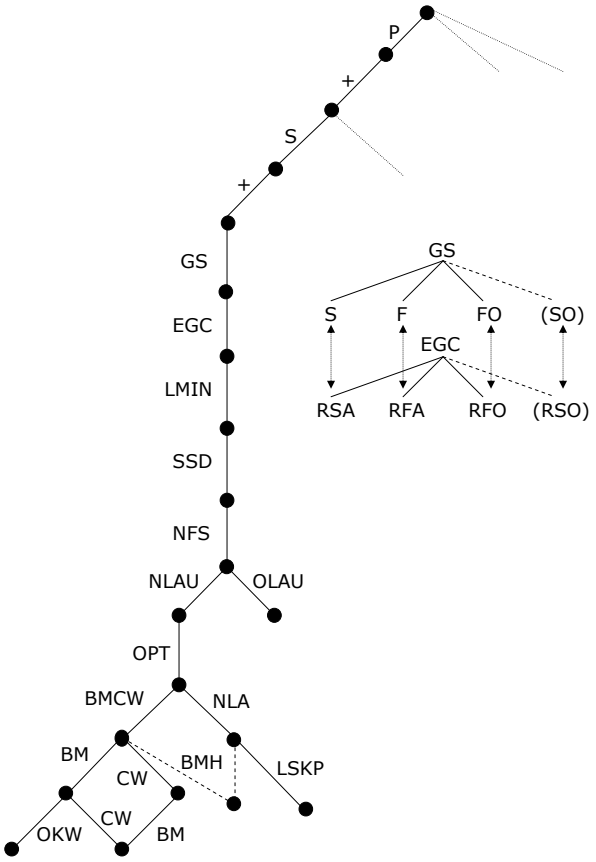

Figure 1: A new taxonomy of sublinear keyword pattern matching algorithms. Edges are directed in the top-down direction. A list of the algorithm and problem details used plus a short description of each detail is given in Appendix A.

3.9 One symbol lookahead at the unscanned part of the input string 20

4 Factor-based sublinear pattern matching **21**
4.1 The no-factor shift 22
4.2 Cheap computation of a particular shift function 24

5 Factor oracle-based sublinear pattern matching **26**

6 Final remarks **28**

A Algorithm and problem details **31**

B Definitions **32**

1 Introduction

The (exact) keyword pattern matching problem can be described as "the problem of finding all occurrences of keywords from a given set as substrings in a given string" [WZ95, WZ96]. This problem has been frequently studied in the past, and many different algorithms have been suggested for solving it. Watson and Zwaan (in [WZ95, WZ96], [Wat95, Chapter 4]) derived a set of well-known solutions to the problem—including prefix-based Knuth-Morris-Pratt [KMP77] and Aho-Corasick [AC75] and suffix-based Boyer-Moore [BM77], Commentz-Walter [CW79a, CW79b] and variants of these four algorithms—from a common starting point, factoring out their commonalities and presenting them in a common setting to better comprehend and compare them. Other overviews of keyword pattern matching are given by Crochemore & Rytter [CR03], Apostolico & Galil [AG97], and many others.

Although the taxonomy contained a large number of variations on these four basic algorithms, some efficient variants were not included. Among these are the single and multiple keyword Boyer-Moore-Horspool algorithms ([Hor80, NR02]). Most importantly however, a new category of algorithms—based on factors instead of prefixes or suffixes of keywords—has emerged in the last decade. This category includes algorithms such as (Set) Backward DAWG Matching ([CCG+94, NR00]) and (Set) Backward Oracle Matching ([ACR01, AR99]). Cleophas (in [Cle03]) extended the existing taxonomy to include these algorithms and their derivations. In this paper, we focus our attention on the part of the new taxonomy containing (multiple) keyword pattern matching algorithms that have potentially sublinear matching time; that is, the number of symbol comparisons may be sublinear in the length of the input string.

Figure 1 shows the new taxonomy of sublinear keyword pattern matching algorithms. Nodes in this taxonomy graph represent algorithms, while edges are labeled with the algorithm or problem detail they represent.

1.1 Related work

The original taxonomy of keyword pattern matching algorithms is presented in Watson's PhD thesis [Wat95, Chapter 4], while the part representing sublinear keyword pattern matching algorithms is described in Watson & Zwaan's [WZ95, WZ96]. Subsections 3.1 through 3.7 and Subsection 3.9 of this paper are modified versions of corresponding parts of those publications. They have been included here to present a complete overview of the (new) taxonomy and reduce the number of external references in this paper. A section on the precomputation of functions (including shift functions) needed by the various algorithms is included in both papers. We do not discuss such precomputation here. The additions to the original taxonomy to form the new taxonomy are described in Cleophas's MSc thesis [Cle03, Chapter 3].

An implementation of most of the algorithms in the (new) taxonomy in the form of the SPARE Time (String PAttern REcognition) toolkit has been made. The implementation is based on the algorithm representations that are part of the taxonomy. SPARE Time is discussed in more detail in [Cle03, Chapter 5]. Benchmarking and performance tuning of this toolkit is currently being performed and will be discussed in a future paper. The toolkit will be available for non-commercial use from `http://www.win.tue.nl/fastar`.

1.2 Taxonomy construction

According to the Merriam Webster's Collegiate Dictionary, a *taxonomy* is:

> [An] orderly classification of plants and animals according to their presumed natural relationships. [Mis93, p. 1208]

Although this definition is somewhat biology oriented, we can create a classification according to essential details of algorithms or data structures from a certain field as well. In our case, such a

classification takes the form of a *(directed acylic) taxonomy graph*[1].

The main goal of constructing a taxonomy is to improve our understanding of the algorithms in a problem domain, and their interrelatedness, i.e. what their commonalities and variations are.

The process of taxonomy construction is preceded by surveying the existing literature of algorithms in the problem field, in order to see what algorithms exist. Based on such a survey, one may try to bring order to the field by placing the algorithms in a taxonomy.

The various algorithms in an algorithm taxonomy are derived from a common starting point by adding details indicating the variations between different algorithms. The common starting point is a naïve algorithm whose correctness is easily shown. Associated with this abstract algorithm are requirements in the form of a pre- and postcondition, an invariant and a specification of (theoretical) running time and/or memory usage, specifying the problem under consideration. The details separating the various algorithms each belong to one of the following categories:

- *Problem details* involve minor changes to pre- and postconditions, restricting in- or output
- *Algorithm details* are used to specify variance in algorithmic structure
- *Representation details* are used to indicate variance in data structures used, internally to an algorithm or influencing the representation of in- and output as well.
- *Performance details* are about variance in running time and memory consumption.

As the goal of constructing our keyword pattern matching algorithm taxonomy is to improve our understanding of the algorithms solving the problem, problem and algorithm details are most important. They are the details forming the taxonomy graph edges. Representation and performance details on the other hand will not be considered that explicitly, although they will be considered[2].

The choice of details—including their granularity—and of how to structure a taxonomy depends on a person's understanding of the algorithms in a domain. A taxonomy therefore is *a* taxonomy of a problem field, but not *the* taxonomy of the field, and taxonomists may end up with different taxonomies for the same field, depending on their understanding of and preference for emphasizing certain details of algorithms. Taxonomy construction is often done bottom-up: initially one may start with as many single-node taxonomies as there are algorithms in the problem domain literature (each taxonomy corresponding to a single algorithm), and as one sees commonalities among the algorithms, one may find generalizations of them which allow combining multiple taxonomies into one larger one with the new generalization as the root. Once a taxonomy has been completely constructed, it is presented in a top-down fashion.

Looking at taxonomy construction as a top-down process, the addition of problem, algorithm or representation details to an algorithm results in a new algorithm solving the same or a similar problem. As a side effect of such detail additions, performance details may change as well. The goal of adding details to algorithms is to (indirectly) improve algorithm performance or to arrive at one of the well-known algorithms appearing in the literature. Associated with the addition of a detail, correctness arguments are given that show how the more detailed algorithm can be derived from its predecessor. To indicate a particular algorithm and to form a taxonomy graph, we use the sequence of details, in order of introduction. In some cases, it may be possible to derive an algorithm in multiple ways through the application of some details in a different order. This causes the taxonomy to take the form of a directed acyclic graph instead of a directed tree.

The type of taxonomy development and program derivation we use here has previously been used for garbage collection algorithms [Jon83], finite automata construction and minimization algorithms [Wat95, Wat04], graph representations [BS02] and other problem fields.

[1] One might say that our classifications are not strictly taxonomies, as the choice points or nodes in our taxonomies are not necessarily single-dimension choice points.

[2] Since the goal of taxonomy construction is to broaden our understanding of a particular domain, we are less interested in practical performance. The focus of a toolkit builder constructing a toolkit based on such a taxonomy, on the other hand, will be more on practical performance aspects.

1.3 Notation used

Since a large part of this paper consists of derivations of existing algorithms, we will often use notations corresponding to their use in existing literature on those algorithms. Nevertheless, we tried to adopt standard conventions for naming variables, functions and sets whenever possible. We use A and B for arbitrary sets, V for the alphabet and V^* for words over the alphabet, P, Q and R for predicates, M for finite automata and Q for state sets. Symbols $a, b, \ldots, e$ represent alphabet symbols from V, while $p, r, \ldots z$ represent words over alphabet V. States are represented by q, while we use δ for automata transition functions. Symbols $i, j, \ldots, n$ represent integer values. We use the symbol $\perp$ ('bottom') to denote an undefined value.

Sometimes functions, relations or predicates are used that have names longer than just a single character. Subscripts, superscripts, prime symbols etc. are sometimes used as well.

We use predicate calculus in our derivations [DF88, DS90] and present our algorithms in an extended version of Dijkstra's guarded command language [Dij76]. The extensions that we use are:

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

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

1.4 Overview

Section 2 presents a formal definition of the exact keyword pattern matching problem, as well as the first, most abstract solutions to it that appear at the top of the taxonomy graph, up to and including the general sublinear algorithm skeleton. In Section 3, suffix-based algorithms such as the Commentz-Walter and Boyer-Moore algorithms are considered. Section 4 discusses algorithms based on factors instead of suffixes, such as (Set) Backward DAWG Matching. The same is done for factor oracle-based algorithms in Section 5. Section 6 gives some concluding remarks and observations about the work reported in this paper, as well as directions for possible future work.

2 The problem and some high-level solutions

In this section, we start out with a naïve solution to the problem and derive more detailed solutions from it. By the end of this section, we arrive at the generalized algorithm skeleton for sublinear keyword pattern matching.

Formally the keyword pattern matching problem, given an alphabet V (a non-empty finite set of symbols), an input string $S \in V^*$, and a finite non-empty pattern set $P = \{p_0, p_1, \ldots p_{|P|-1}\} \subseteq V^*$, is to establish (see Appendix B for an explanation of our notation for quantification)[3]

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

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

A trivial (but unrealistic) solution to the problem is

Algorithm 2.1()

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

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

Two basic directions in which to proceed while developing naïve algorithms to solve this problem are, informally, to consider a substring of S as "suffix of a prefix of S" or as "prefix of a suffix of S". We choose the first possibility (the second leads to mirror images of the algorithms obtained this way) as this is the way that the algorithms we consider treat substrings of input string S.

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

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

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

Algorithm 2.2(P)

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

This algorithm is used as a starting point in [Cle03] and [Wat95] to derive prefix-based algorithms (such as the Aho-Corasick and Knuth-Morris-Pratt algorithms and the Shift-And/-Or algorithms). These algorithms—although very efficient for a number of situations—are not discussed in this paper, as their behaviour is not sublinear.

[3]Note that lvr is used for the concatenation of l, v and r. Also note that the problem definition is slightly different but equivalent to that used in [WZ95, WZ96, Wat95], where $R: \ O = \big(\bigcup\ l, v, r : lvr = S : \{l\} \times (\{v\} \cap P) \times \{r\}\big)$ is used. As a result, the algorithms given in this text will be slightly different in structure but equivalent in meaning to the algorithms of the same name in those texts.

The update of O in the repetition of the preceding algorithm can be computed with another non-deterministic repetition. This inner repetition would consider suffixes of u. Thus by applying "examine suffixes of a given string in any order" (algorithm detail (S)) to string u we obtain the following algorithm:

Algorithm 2.3(P, S)

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

Algorithm (P, S) consists of two nested non-deterministic repetitions. We could have combined the **as** statement with the inner **for**-loop, but do not do so in anticipation of algorithm detail choices to be made. Each repetition can be made deterministic by considering prefixes (or suffixes as the case is) in increasing (called detail (+)) or decreasing (detail (−)) order of length. Since the algorithms we want to consider in order to achieve sublinear behaviour examine string S from left to right, and the patterns in P from right to left, we focus our attention on:

Algorithm 2.4(P_+, S_+)

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

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

To arrive at a more efficient algorithm, we try to strengthen the guard of the inner loop ($l \neq \varepsilon$). In [WZ96, WZ95, Wat95], the guard was strengthened by adding **cand** $(l\uparrow 1)v \in \mathbf{suff}(P)$ (see Notation B.4 for **cand** and Definition B.9 for the definition of **suff**, **pref** and **fact**). A more general strengthening is possible however. Suppose we have a function $\mathbf{f} \in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ satisfying

$$P \subseteq \mathbf{f}(P) \wedge \mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$$

then we have (for all $w, x \in V^*$) $w \notin \mathbf{f}(P) \Rightarrow w \notin P$ (application of the left conjunct) and $w \notin \mathbf{f}(P) \Rightarrow xw \notin P$ (application of right followed by left conjunct and definition of **suff**). We may therefore strengthen the guard to $l \neq \varepsilon$ **cand** $(l\uparrow 1)v \in \mathbf{f}(P)$:

Algorithm detail 2.5. (GS). (Guard Strengthening). Strengthening the inner repetition guard $l \neq \varepsilon$ to $l \neq \varepsilon$ **cand** $(l\uparrow 1)v \in \mathbf{f}(P)$ for function $\mathbf{f} \in \mathcal{P}(V^*) \rightarrow \mathcal{P}(V^*)$ satisfying $P \subseteq \mathbf{f}(P)$ and $\mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$. □

This leads to the following generalized algorithm skeleton[4]:

[4]We use the term generalized to indicate that the skeleton can be instantiated with various functions **f**, in contrast to the algorithm skeleton used in [WZ96, Wat95] where a skeleton using function **suff** is treated.

Algorithm 2.6(P$_+$, S$_+$, GS)

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

Observe that $v \in \mathbf{f}(P)$ is now an invariant of the inner repetition. This invariant is initially established by the assignment $v := \varepsilon$ since $P \neq \varnothing$ and thus $\varepsilon \in \mathbf{f}(P)$.

Several choices for $\mathbf{f}(P)$ will turn out to be allowed, of which we mention the following:

- **suff**(P). This choice will be discussed in Section 3.
- **fact**(P). We discuss this choice in Section 4.
- **factoracle**$(P^R)^R$. Function **factoracle** returns a superset of **fact**. An implementation of this choice using a kind of automata called *factor oracles* is discussed in Section 5.
- A function that returns a superset of **suff**(P). This could be implemented using **sufforacle**, i.e. the function defining the language recognized by a *suffix oracle* [ACR01, AR99] on a set of keywords. We will not explore this option here.

Direct evaluation of $(l\restriction 1)v \in \mathbf{f}(P)$ is expensive. Instead, the transition function $\delta_{R,\mathbf{f},P}$ of a finite automaton recognizing $\mathbf{f}(P)^R$ is used—where we use the reversal operator ^{R} since suffixes of u are read in reverse—such that $\delta_{R,\mathbf{f},P}$ has the following property:

Property 2.7 (Transition function of automaton recognizing f$(P)^R$). The transition function $\delta_{R,\mathbf{f},P}$ of a (weakly deterministic) finite automaton[5] $M = \langle Q, V, \delta_{R,\mathbf{f},P}, q_0, F\rangle$ recognizing $\mathbf{f}(P)^R$ has the property that

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

and we assume $\delta^*_{R,\mathbf{f},P}(q, \varepsilon) = q$ □

Note that Property 2.7 requires **pref**$(\mathbf{f}(P)^R) \subseteq \mathbf{f}(P)^R$, i.e. **suff**$(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$. Also note that $w^R \in \mathbf{f}(P)^R \equiv w \in \mathbf{f}(P)$. Since we will always refer to the same set P in the remainder of this document, we will use $\delta_{R,\mathbf{f}}$ instead of $\delta_{R,\mathbf{f},P}$. Transition function $\delta_{R,\mathbf{f}}$ can be computed beforehand, as in [WZ95, WZ96, Section 4.1][6].

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

[5] Finite automata and δ^* are introduced in Definitions B.17 and B.18.

[6] In [WZ95, WZ96], the transition function is called τ_P. It is called $\tau_{P,r}$ in [Wat95] to distinguish it from the *forward trie* function. We generalize the function by means of a parameter **f** and make it into a transition function on automata.

Algorithm detail 2.8. (EGC). (Efficient Guard Computation). Given a finite automaton recognizing $\mathbf{f}(P)^R$ and satisfying Property 2.7, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{f}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{f}(P)$ then becomes $q \neq \bot$. □

This algorithm detail leads to the following algorithm skeleton:

Algorithm 2.9(P$_+$, S$_+$, GS, EGC)

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

The particular automaton choices for this detail will be discussed together with the corresponding choices for detail (GS) in Sections 3 through 5. Note that guard $v \in P$ can be efficiently computed, i.e. computed in $\Theta(1)$, in this and following algorithms by providing a map from states of the automaton to a boolean.[7]

2.1 A change leading to smaller automata

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

Algorithm detail 2.10. (LMIN). The automaton used in algorithm detail (EGC) is built on $\mathbf{f}(P')$ where $P' = \{w : w \in \mathbf{pref}(P) \wedge |w| = lmin_P\}$ instead of on $\mathbf{f}(P)$. □

As a result of using algorithm detail (LMIN) with Algorithm 2.9 (P$_+$, S$_+$, GS, EGC), after assignment $l, v := l\downarrow 1, (l\restriction 1)v$ in the inner loop, $v \in \mathbf{f}(P')$ holds (instead of $v \in \mathbf{f}(P)$ as before). Due to Property 2.7, in case $|v| = lmin_P$ (i.e. $v \in P'$) we need to verify any matches $v(r\restriction i) \in P$ for $i \leq lmax_P - lmin_P$ (where $lmax_P = \big(\mathbf{MAX}\, p : p \in P : |p|\big)$). Since there is a longest keyword, we do not need to increase i past the mentioned maximum value. This leads to the following algorithm skeleton (where details (GS) and (EGC) still need to be instantiated):

Remark 2.11. Note that this algorithm detail could be applied to *any* of the pattern matching algorithms in the taxonomy shown in Figure 1. □

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

Algorithm 2.12(P$_+$, S$_+$, GS, EGC, LMIN)

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

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

Remark 2.13. It is possible to improve the efficiency of the last part of this algorithm by introducing a forward trie function (as in [Wat95, Section 4.2.2]), assuming that the initialization to $\delta^*(q_0, v)$ in the forward trie is a $\Theta(1)$ operation. This can be achieved for example by having a mapping between each element of $\mathbf{f}(P')$ of length $lmin$ and the corresponding state of the forward trie. Since $|v| = lmin_P$ always holds when the forward trie is used, it might be possible to only construct the parts of the trie of depth $lmin_P$ or greater. We do not further discuss this option. □

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

- Reduced size of automaton: Since $lmin_P$ might be less than $|p_i|$ for some i, the automaton might have less states and less transitions. This gain may (partially) be offset by the time spent executing the new **as** $|v| = lmin_P \to \ldots$ **sa** statement, or by the space and time spent when introducing the forward trie as in Remark 2.13.
- Reduced maximal shift distances with detail (SSD)[8]: Since $|v|$ might be less than $|p_i|$ (for i such that $|p_i| \geq lmin_P$), there is less information from v that can be used. Hence shift distances might be smaller, leading to a larger total number of shifts.
- Reduced number of character comparisons: Let P consist of keywords p_i, p_j, such that $p_j \neq p_i$ and $p_j \in \mathbf{pref}(p_i)$. In this case, using detail (LMIN) it will take less comparisons to verify a match of both keywords. Originally, for an occurrence of p_i in S, $|p_i| + |p_j|$ comparisons are needed to detect both matches. Using detail (LMIN), $lmin + (|p_i| - lmin) = |p_i|$ comparisons are needed.

[8]To be introduced in the next subsection.

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

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

2.2 A generalized sublinear algorithm skeleton

Starting from Algorithm 2.9 (P_+, S_+, GS, EGC)[9], we derive a generalized sublinear keyword pattern matching algorithm skeleton forming the basis of a family of sublinear algorithms. The basic idea is to make shifts of more than one symbol. This is accomplished by replacing $u, r := u(r\uparrow 1), r\downarrow 1$ by $u, r := u(r\uparrow k), r\downarrow k$ for some k satisfying $1 \leq k \leq \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing : n\big)$. The upperbound is the distance to the next match, the maximal safe shift distance. Any smaller number k satisfying the equation is safe as well, and we thus define a safe shift distance as:

Definition 2.14 (Safe shift distance). A shift distance k satisfying

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

is called a *safe shift distance*. □

Algorithm detail 2.15. (SSD). Replace assignment

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

in Algorithm 2.9 (P_+, S_+, GS, EGC) by assignment

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

using a safe shift distance k. □

Since shift functions may depend on l, v and r, we will write $k(l, v, r)$.

We aim at approximating the maximal safe shift distance from below, since computing the maximum safe shift distance itself essentially amounts to solving our original problem. To do this, we weaken the predicate $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$. This results in safe shift distances that are easier to compute than the maximal safe shift distance. In the derivation of such weakening steps in Sections 3 through 5, the $u = lv \wedge v \in \mathbf{f}(P)$ part of the invariant of the inner repetition in Algorithm 2.9 will be used. By adding $l, v := \varepsilon, \varepsilon$ to the initial assignments of the algorithm, we turn this into an invariant of the outer repetition. This also turns $l = \varepsilon$ **cor** $(l\uparrow 1)v \notin \mathbf{f}(P)$—the negation of the guard of the inner repetition—into an invariant of the outer repetition. Hence, we arrive at the following algorithm skeleton:

Algorithm 2.16(P_+, S_+, GS, EGC, SSD)

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

[9] One could start from Algorithm 2.12 (P_+, S_+, GS, EGC, LMIN) as well.

```
    do l ≠ ε cand q ≠ ⊥ →
        l, v := l↓1, (l↑1)v;
        q := δ_{R,f}(q, l↑1);
        as v ∈ P → O := O ∪ {(l, v, r)} sa
    od
od{ R }
```

Using this algorithm skeleton, various sublinear algorithms may be obtained by choosing appropriate $\mathbf{f}(P)$ and shift functions $k(l, v, r)$. The next three sections consider the choice of $\mathbf{suff}(P)$, $\mathbf{fact}(P)$ and $\mathbf{factoracle}(P^R)^R$ for $\mathbf{f}(P)$ respectively.

In [Wat00], an alternative algorithm skeleton for (suffix-based) sublinear keyword pattern matching is presented, in which the update to O in the inner loop has been moved out of that loop. This requires the use of a precomputed output function, but has the potential to substantially reduce the algorithms' running time. This alternative skeleton is not considered in this paper.

3 Suffix-based sublinear pattern matching

We now derive a family of algorithms by using the set of suffixes of P, **suff**(P), for **f**(P) in Algorithm 2.9, i.e. instantiating **f** with **suff**. We introduce

Algorithm detail 3.1. (GS=S). (Guard Strengthening = Suffix). Strengthen the guard of the inner repetition by adding conjunct $(l\restriction 1)v \in \mathbf{suff}(P)$. □

As indicated in Section 2, direct evaluation of $(l\restriction 1)v \in \mathbf{suff}(P)$ is expensive and a reverse suffix automaton is used with algorithm detail (EGC): given a finite automaton recognizing $\mathbf{suff}(P)^R$ and satisfying Property 2.7, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\restriction 1)v)^R)$. The guard conjunct $(l\restriction 1)v \in \mathbf{suff}(P)$ then becomes $q \neq \bot$.[10]

Algorithm 3.2(P$_+$, S$_+$, GS=S, EGC=RSA)

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

Assuming set P and (as a result) $\big(\mathbf{MAX}\, p : p \in P : |p|\big)$ to be constant, the algorithm has $\Theta(|S|)$ running time.

We can introduce the safe shift distance, i.e. instantiating **f** with **suff** in Algorithm 2.16:

Algorithm 3.3(P$_+$, S$_+$, GS=S, EGC=RSA, SSD)

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

[10] These two algorithm details are equivalent to the introduction of algorithm detail (RT) in [WZ95, WZ96, Wat95], where the first detail (the strengthening of the guard) is introduced implicitly. We replaced it by two separate algorithm detail names here as part of the generalization of guard strengthening and efficient guard computation that was discussed in Section 2.

Based on this algorithm skeleton, various shift functions will be obtained in the remainder of this section. This will lead to the Commentz-Walter, Fu-San and multiple keyword Boyer-Moore and Boyer-Moore-Horspool algorithms[11].

3.1 No lookahead at the unscanned part of the input string

As indicated in Section 2.2, we aim at an approximation of the maximal safe shift distance that is easier to compute, but may not always result in the maximal safe shift distance (although it will never exceed that value). We first derive an approximation that does not depend on r and that will be a starting point of most of our further derivations. In terms of algorithms this means that we refrain from looking ahead at the symbols of r, the yet unscanned part of the input string (algorithm detail (NLAU) (No LookAhead at Unscanned part of the input string)). This is in accordance with most of the algorithms we are aiming at. One symbol lookahead at the unscanned part of the input string is discussed in Subsection 3.9. Taking the upperbound of Definition 2.14 as a starting point, we derive

$$
\begin{array}{ll}
 & \big(\mathbf{MIN}\, n : 1 \le n \wedge \mathbf{suff}\big(u(r{\uparrow}n)\big) \cap P \neq \varnothing : n\big) \\
= & \quad \{\, \text{domain split } n \le |r| \vee n > |r|,\ r{\uparrow}n = r \text{ if } n \ge |r| \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \le n \le |r| \wedge \mathbf{suff}\big(u(r{\uparrow}n)\big) \cap P \neq \varnothing : n\big) \\
 & \quad \mathbf{min}\, \big(\mathbf{MIN}\, n : |r| < n \wedge \mathbf{suff}(ur) \cap P \neq \varnothing : n\big) \\
\ge & \quad \{\, 1 \le n \le |r| : r{\uparrow}n \in V^n, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \le n \le |r| \wedge \mathbf{suff}(uV^n) \cap P \neq \varnothing : n\big) \\
 & \quad \mathbf{min}\, \big(\mathbf{MIN}\, n : |r| < n \wedge \mathbf{suff}(ur) \cap P \neq \varnothing : n\big) \\
= & \quad \{\, r \in V^{|r|}, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap :\ \mathbf{suff}(ur) \cap P \neq \varnothing \Rightarrow \mathbf{suff}\big(uV^{|r|}\big) \cap P \neq \varnothing \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \le n \le |r| \wedge \mathbf{suff}(uV^n) \cap P \neq \varnothing : n\big) \\
\ge & \quad \{\, \text{enlarging domain} \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \le n \wedge \mathbf{suff}(uV^n) \cap P \neq \varnothing : n\big)
\end{array}
$$

Since the last formula is to be the starting point of our further derivations we will from here on aim at shift functions k being dependent only on u, i.e. on l and v (recall $u = lv$ as in the invariant in Algorithm 3.3). We will write $k(l,v)$ instead of $k(l,v,r)$.

3.2 Restriction to one symbol lookahead

In all derivations in this and following subsections we assume

$$u = lv \wedge v \in \mathbf{suff}(P).$$

Restriction to one symbol lookahead ($l{\uparrow}1$, the last symbol of u scanned in the inner loop) leads to the algorithm by Fan and Su [FS93, FS94]. It is obtained by weakening the predicate in the domain of the approximation of the upperbound in Subsection 3.1 in the following way:

$$
\begin{array}{ll}
 & \mathbf{suff}(uV^n) \cap P \neq \varnothing \\
= & \quad \{\, u = lv \,\} \\
 & \mathbf{suff}(lvV^n) \cap P \neq \varnothing \\
\Rightarrow & \quad \{\, l = (l{\downarrow}1)(l{\uparrow}1),\ l{\downarrow}1 \in V^*, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\
 & \mathbf{suff}\big(V^*(l{\uparrow}1)vV^n\big) \cap P \neq \varnothing
\end{array}
$$

[11] With the exception of the multiple keyword Boyer-Moore-Horspool algorithm, which is described in [Cle03], these were previously described in [WZ95, WZ96] already, where the algorithm skeleton is called (P$_+$, S$_+$, RT, SSD).

$$
\begin{array}{ll}
= & \{\, \mathbf{suff}(A) \cap B \neq \varnothing \;\equiv\; A \cap V^*B \neq \varnothing \,\} \\
& V^*(l{\restriction}1)vV^n \cap V^*P \neq \varnothing \\
= & \{\, \text{property B.16} \,\} \\
& V^*(l{\restriction}1)vV^n \cap P \neq \varnothing \;\vee\; vV^n \cap V^*P \neq \varnothing
\end{array}
$$

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

$$
k_{opt}(x,y) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge \big(V^*(x{\restriction}1)yV^n \cap P \neq \varnothing \vee yV^n \cap V^*P \neq \varnothing\big) : n\big)\,.
$$

Function k_{opt} can be expressed as follows

$$
k_{opt}(x,y) = \begin{cases} d_{opt}(x{\restriction}1, y)\,\mathbf{min}\, d_{sp}(y) & x \neq \varepsilon \\ d_i(y)\,\mathbf{min}\, d_{sp}(y) & x = \varepsilon \end{cases}
$$

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

$$
d_{opt}(a,y) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge V^*ayV^n \cap P \neq \varnothing : n\big) \qquad (a \in V, y \in \mathbf{suff}(P)),
$$

$d_{sp} \in \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$
d_{sp}(y) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge yV^n \cap V^*P \neq \varnothing : n\big) \qquad (y \in \mathbf{suff}(P)),
$$

(function d_2 in [CW79a, CW79b]), and $d_i \in \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$
d_i(y) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge V^*yV^n \cap P \neq \varnothing : n\big) \qquad (y \in \mathbf{suff}(P)),
$$

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

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

3.3 Lookahead symbol is mismatching

We derive an approximation from below of d_{opt} that yields an algorithm that is the common ancestor of the multiple keyword generalization of the Boyer-Moore algorithm [BM77] and the Commentz-Walter algorithm [CW79a, CW79b]. Essentially, the resulting shift function is not based on the identity of the lookahead symbol $l{\restriction}1$ but only uses the fact that the lookahead symbol is mismatching, as is done in the Boyer-Moore shift function. In this way one might say that the recognized suffix and the (mismatching) lookahead symbol have to some extent been decoupled.

We start by weakening the predicate from d_{opt}. Assume $l \neq \varepsilon$ and $(l{\restriction}1)v \notin \mathbf{suff}(P)$. We derive

$$
\begin{array}{ll}
& V^*(l\restriction 1)vV^n \cap P \neq \varnothing \\
= & \qquad \{\, v \in V^{|v|},\ \text{monotonicity of } \cap,\ \text{duplication} \,\} \\
& V^*(l\restriction 1)V^{|v|+n} \cap P \neq \varnothing \ \ \wedge \ \ V^*(l\restriction 1)vV^n \cap P \neq \varnothing \\
\Rightarrow & \qquad \{\, (l\restriction 1)v \notin \mathbf{suff}(P),\ \text{so } l\restriction 1 \in \{\, a \mid a \in V \wedge av \notin \mathbf{suff}(P) \,\},\ \text{definition } MS \,\} \\
& V^*(l\restriction 1)V^{|v|+n} \cap P \neq \varnothing \ \ \wedge \ \ V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing
\end{array}
$$

where $MS \in \mathbf{suff}(P) \to V$ is defined by

$$MS(y) = \{\, a \mid a \in V \wedge ay \in \mathbf{suff}(P) \,\} \qquad (y \in \mathbf{suff}(P)).$$

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

$$
\begin{array}{ll}
& \big(\mathbf{MIN}\, n : n \geq 1 \ \ \wedge \ \ V^*(l\restriction 1)V^{|v|+n} \cap P \neq \varnothing \ \ \wedge \ \ V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing : n\big) \\
\geq & \qquad \{\, \mathbf{MIN} \text{ with conjunctive range} \,\} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \ \wedge \ \ V^*(l\restriction 1)V^{|v|+n} \cap P \neq \varnothing : n\big) \\
& \qquad \mathbf{max}\, \big(\mathbf{MIN}\, n : n \geq 1 \ \ \wedge \ \ V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing : n\big) \\
= & \qquad \{\, \text{change of bound variable: } m = |v| + n;\ \text{definition } d_{vi} \ \text{(after derivation)} \,\} \\
& \big(\mathbf{MIN}\, m : m \geq |v| + 1 \ \ \wedge \ \ V^*(l\restriction 1)V^m \cap P \neq \varnothing : m - |v|\big)\, \mathbf{max}\, d_{vi}(v) \\
\geq & \qquad \{\, \text{enlarging domain} \,\} \\
& \big(\mathbf{MIN}\, m : m \geq 1 \ \ \wedge \ \ V^*(l\restriction 1)V^m \cap P \neq \varnothing : m - |v|\big)\, \mathbf{max}\, d_{vi}(v) \\
= & \qquad \{\, l \neq \varepsilon,\ \text{definition of } char_{cw} \ \text{(after derivation)} \,\} \\
& char_{cw}(l\restriction 1, |v|)\, \mathbf{max}\, d_{vi}(v)
\end{array}
$$

where $d_{vi} \in \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$d_{vi}(y) = \big(\mathbf{MIN}\, n : n \geq 1 \ \ \wedge \ \ V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing : n\big) \qquad (y \in \mathbf{suff}(P))$$

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

$$char_{cw}(a, z) = \big(\mathbf{MIN}\, n : n \geq 1 \ \ \wedge \ \ V^*aV^n \cap P \neq \varnothing : n - z\big) \qquad (a \in V, z \in \mathbb{N}).$$

This results in shift distance $k_{bmcw}(l,v)$ where $k_{bmcw} \in V^* \times \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$k_{bmcw}(x, y) = \begin{cases} \big(char_{cw}(x\restriction 1, |y|)\, \mathbf{max}\, d_{vi}(y)\big)\, \mathbf{min}\, d_{sp}(y) & (x \in V^+, y \in \mathbf{suff}(P)) \\ d_i(y)\, \mathbf{min}\, d_{sp}(y) & (x = \varepsilon, y \in \mathbf{suff}(P)). \end{cases}$$

Notice that we have

$$k_{opt}(x, y) \geq k_{bmcw}(x, y) \qquad (x \in V^*, y \in \mathbf{suff}(P)).$$

Approximation from below of k_{opt} by k_{bmcw} is referred to as algorithm detail (BMCW). We chose this name to reflect that essential ideas from both the Boyer-Moore and Commentz-Walter algorithms are introduced. In the next two subsections these algorithms are derived from the algorithm presented in this subsection and characterized by detail sequence (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW).

3.4 The multiple keyword Boyer-Moore algorithm

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

$$
\begin{array}{ll}
& char_{cw}(l\restriction 1, |v|) \\
= & \quad \{ \text{ definition } char_{cw} \} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \wedge \ V^*(l\restriction 1)V^n \cap P \neq \varnothing : n - |v|\big) \\
\geq & \quad \{ V^*(l\restriction 1)V^n \cap P \neq \varnothing \Rightarrow V^*(l\restriction 1)V^n \cap V^*P \neq \varnothing \} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \wedge \ V^*(l\restriction 1)V^n \cap V^*P \neq \varnothing : n - |v|\big) \\
= & \quad \{ P \neq \varnothing,\, V^*(l\restriction 1)V^{|p|+1} \cap V^*p \neq \varnothing \text{ for all } p \in P, \text{ nonempty domain} \} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \wedge \ V^*(l\restriction 1)V^n \cap V^*P \neq \varnothing : n\big) - |v| \\
= & \quad \{ l \neq \varepsilon, \text{ definition of } char_{bm} \text{ (after derivation)} \} \\
& char_{bm}(l\restriction 1) - |v|
\end{array}
$$

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

$$char_{bm}(a) = \big(\mathbf{MIN}\, n : n \geq 1 \wedge V^*aV^n \cap V^*P \neq \varnothing : n\big) \qquad (a \in V).$$

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

$$k_{bm}(x,y) = \begin{cases} \big((char_{bm}(x\restriction 1) - |y|) \,\mathbf{max}\, d_{vi}(y)\big) \,\mathbf{min}\, d_{sp}(y) & (x \in V^+, y \in \mathbf{suff}(P)) \\ d_i(y) \,\mathbf{min}\, d_{sp}(y) & (x = \varepsilon, y \in \mathbf{suff}(P)). \end{cases}$$

Approximating k_{bmcw} from below by k_{bm} is referred to as algorithm detail (BM). It results in the multiple keyword generalization of the regular Boyer-Moore algorithm [BM77]. The algorithm is characterized by detail sequence (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BM). The regular Boyer-Moore algorithm can be obtained by restricting P to one keyword (problem detail (OKW) (One KeyWord)). Notice that we have

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

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

The formula for the Boyer-Moore shift function given here differs from but is equivalent to the ones given in [BM77] and [Aho90], as is shown in [WZ95, WZ96].

3.5 The Commentz-Walter algorithm

Instead of approximating $char_{cw}$ in k_{bmcw} from below by $char_{bm}$ we now approximate d_{vi} in k_{bmcw} from below by d_i. This results in the Commentz-Walter algorithm [CW79a, CW79b]. We derive

$$
\begin{array}{ll}
& d_{vi}(v) \\
= & \quad \{ \text{ definition } d_{vi} \} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \wedge \ V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing : n\big) \\
\geq & \quad \{ V^*AvV^n \cap P \neq \varnothing \Rightarrow V^*vV^n \cap P \neq \varnothing \} \\
& \big(\mathbf{MIN}\, n : n \geq 1 \ \wedge \ V^*vV^n \cap P \neq \varnothing : n\big) \\
= & \quad \{ \text{ definition } d_i \} \\
& d_i(v).
\end{array}
$$

This results in shift distance $k_{cw}(l,v)$ where $k_{cw} \in V^* \times \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$k_{cw}(x,y) = \begin{cases} \big(char_{cw}(x{\uparrow}1, |y|) \,\mathbf{max}\, d_i(y)\big) \,\mathbf{min}\, d_{sp}(y) & (x \in V^+, y \in \mathbf{suff}(P)) \\ d_i(y) \,\mathbf{min}\, d_{sp}(y) & (x = \varepsilon, y \in \mathbf{suff}(P)). \end{cases}$$

Approximating k_{bmcw} from below by k_{cw} is referred to as algorithm detail (CW). It results in the Commentz-Walter algorithm [CW79a, CW79b] that is characterized by detail sequence (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, CW). Notice that we have

$$k_{bmcw}(x,y) \geq k_{cw}(x,y) \qquad (x \in V^*, y \in \mathbf{suff}(P)).$$

Such a comparison can not be made between k_{bm} and k_{cw} as the following example shows.

Example 3.4 Let $V \in \{a,b,c,d\}$, $P = \{cababa\}$, and $x \in V^*$. Shift functions k_{bm} and k_{cw} are incomparable since

$$\begin{array}{rcccl} k_{opt}(xd,a) & = & +\infty \,\mathbf{min}\, 6 & = & 6 \\ k_{bmcw}(xd,a) & = & (+\infty \,\mathbf{max}\, 4) \,\mathbf{min}\, 6 & = & 6 \\ k_{bm}(xd,a) & = & \big((6-1) \,\mathbf{max}\, 4\big) \,\mathbf{min}\, 6 & = & 5 \\ k_{cw}(xd,a) & = & (+\infty \,\mathbf{max}\, 2) \,\mathbf{min}\, 6 & = & 6 \end{array}$$

and

$$\begin{array}{rcccl} k_{opt}(xa,a) & = & +\infty \,\mathbf{min}\, 6 & = & 6 \\ k_{bmcw}(xa,a) & = & \big((2-1) \,\mathbf{max}\, 4\big) \,\mathbf{min}\, 6 & = & 4 \\ k_{bm}(xa,a) & = & \big((2-1) \,\mathbf{max}\, 4\big) \,\mathbf{min}\, 6 & = & 4 \\ k_{cw}(xa,a) & = & \big((2-1) \,\mathbf{max}\, 2\big) \,\mathbf{min}\, 6 & = & 2. \end{array}$$

It also follows that in some cases k_{bmcw} is smaller than k_{opt} and that in some cases k_{bm} and k_{cw} are smaller than k_{bmcw}. □

It is not possible that both $k_{bmcw}(x,y) > k_{bm}(x,y)$ and $k_{bmcw}(x,y) > k_{cw}(x,y)$ hold for some $x \in V^+$ and $y \in \mathbf{suff}(P)$ since the first inequality implies $char_{cw}(x{\uparrow}1, |y|) = +\infty$ and this in its turn implies $k_{bmcw}(x,y) = d_{sp}(y) = k_{cw}(x,y)$.

3.6 Complete decoupling of recognized suffix and lookahead symbol

The derivations in the previous subsections effect an ever stronger decoupling of the recognized suffix v and the lookahead symbol $l{\uparrow}1$ in the subsequent shift functions. By approximating d_{vi} in k_{bm} from below by d_i or $char_{cw}$ in k_{cw} by $char_{bm}$ (or both in k_{bmcw}) we obtain a complete decoupling. It results in shift distance $k_{dsl}(l,v)$ where $k_{dsl} \in V^* \times \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$k_{dsl}(x,y) = \begin{cases} \big((char_{bm}(x{\uparrow}1) - |y|) \,\mathbf{max}\, d_i(y)\big) \,\mathbf{min}\, d_{sp}(y) & (x \in V^+, y \in \mathbf{suff}(P)) \\ d_i(y) \,\mathbf{min}\, d_{sp}(y) & (x = \varepsilon, y \in \mathbf{suff}(P)). \end{cases}$$

The algorithm can be characterized by detail sequences (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BM, CW) and (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, CW, BM).

3.7 Discarding the lookahead symbol

We weaken the predicate in the range of k_{opt} by weakening its first disjunct to $V^* v V^n \cap P \neq \varnothing$ due to $V^*(l{\uparrow}1) \subseteq V^*$ and the monotonicity of $\cap$. This weakening step is referred to as discarding the lookahead symbol $l{\uparrow}1$. The shift distance corresponding to this weakening is $k_{nla}(v)$ where $k_{nla} \in \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$k_{nla}(y) = d_i(y) \,\mathbf{min}\, d_{sp}(y) \qquad (y \in \mathbf{suff}(P)).$$

Notice that this shift function can also be viewed as an approximation from below of k_{dsl}. Precomputation of d_i and d_{sp} is discussed in [WZ95, WZ96, Section 4.2]. Approximating k_{opt} from below by k_{nla} is referred to as algorithm detail (NLA) (No LookAhead at mismatching symbol) and results in algorithm (P_+S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, NLA).

3.8 The multiple-keyword Boyer-Moore-Horspool algorithm

Here, we consider two particular weakenings of the range predicate $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$ of Definition 2.14:

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

(used in algorithm detail (NLA) discussed before) and

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

(used in the derivation at the beginning of Section 3.2). We now further weaken the first predicate, for the case $v \neq \varepsilon$:

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

Weakening the second predicate, for the case $v = \varepsilon$, we get:

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

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

$$\begin{array}{ll} & \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing : n\big) \\ \geq & \quad \{\ \text{weakening steps above}\ \} \\ & \big(\mathbf{MIN}\, n : 1 \leq n \wedge (V^*(v\uparrow 1)V^n \cap V^* P \neq \varnothing) : n\big) \\ = & \quad \{\ \text{definition of } \mathit{char}_{bm}\ \} \\ & \mathit{char}_{bm}(v\uparrow 1) \end{array}$$

To use $\mathit{char}_{bm}(l\uparrow 1)$ as a safe shift distance, we need $l \neq \varepsilon$ to hold in case $v = \varepsilon$. Note that in Algorithm 3.3 (P$_+$, S$_+$, GS=S, EGC=RSA, SSD), $l \neq \varepsilon$ does not hold initially. Assuming $\varepsilon \notin P$, we can solve this by changing the initialization to $u, r := S\uparrow lmin_P, S\downarrow lmin_P$ (where $lmin_P = \big(\mathbf{MIN}\, p : p \in P : |p|\big)$). This results in shift distance $k_{bmh}(l,v)$ where $k_{bmh} \in V^* \times \mathbf{suff}(P) \to \mathbb{N}$ is defined by

$$k_{bmh}(l,v) = \begin{cases} \mathit{char}_{bm}(v\uparrow 1) & \text{if } v \neq \varepsilon, \\ \mathit{char}_{bm}(l\uparrow 1) & \text{if } v = \varepsilon. \end{cases}$$

The use of shift function k_{bmh} yields algorithm (P$_+$, S$_+$, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BMH)[12], the Set Horspool algorithm [NR02, Section 3.3.2]. Adding problem detail (OKW) leads to the single-keyword Horspool algorithm [NR02, Section 2.3.2], [Hor80].

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

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

In this subsection we consider looking ahead at the first symbol of the unscanned part r of the input string. The first symbol of r will be taken into account independently of the other available information. In this way we obtain stronger variants of all of the shift functions derived thus far. Assuming $r \neq \varepsilon$ and taking the upperbound on k given in Definition 2.14 as a starting point, we derive

$$
\begin{array}{ll}
 & \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}\big(u(r{\uparrow}n)\big) \cap P \neq \varnothing : n\big) \\
= & \quad \{\, \text{domain split, } 1 \leq n \leq |r|\text{: } r{\uparrow}n = (r{\uparrow}1)\big((r{\downarrow}1){\uparrow}(n-1)\big),\ |r| < n\text{: } r{\uparrow}n = r \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}\big(u(r{\uparrow}1)\big((r{\downarrow}1){\uparrow}(n-1)\big)\big) \cap P \neq \varnothing : n\big) \\
 & \quad \mathbf{min}\, \big(\mathbf{MIN}\, n : |r| < n \wedge \mathbf{suff}(ur) \cap P \neq \varnothing : n\big) \\
\geq & \quad \{\, 1 \leq n \leq |r|\text{: } (r{\downarrow}1){\uparrow}(n-1) \in V^{n-1}\text{, monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}\big(u(r{\uparrow}1)V^{n-1}\big) \cap P \neq \varnothing : n\big) \\
 & \quad \mathbf{min}\, \big(\mathbf{MIN}\, n : |r| < n \wedge \mathbf{suff}(ur) \cap P \neq \varnothing : n\big) \\
= & \quad \{\, r \neq \varepsilon,\ r \in (r{\uparrow}1)V^{|r|-1},\ \mathbf{suff}(ur) \cap P \neq \varnothing \Rightarrow \mathbf{suff}\big(u(r{\uparrow}1)V^{|r|-1}\big) \cap P \neq \varnothing \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}\big(u(r{\uparrow}1)V^{n-1}\big) \cap P \neq \varnothing : n\big) \\
\geq & \quad \{\, u \in V^*\text{, monotonicity of } \mathbf{suff} \text{ and } \cap \,\} \\
 & \big(\mathbf{MIN}\, n : 1 \leq n \leq |r| \wedge \mathbf{suff}\big(V^*(r{\uparrow}1)V^{n-1}\big) \cap P \neq \varnothing : n\big) \\
\geq & \quad \{\, \text{enlarging domain, changing bound variable: } m = n-1 \,\} \\
 & \big(\mathbf{MIN}\, m : 0 \leq m \wedge \mathbf{suff}\big(V^*(r{\uparrow}1)V^m\big) \cap P \neq \varnothing : m+1\big) \\
= & \quad \{\, \mathbf{suff}(A) \cap B \neq \varnothing \equiv A \cap V^*B \neq \varnothing\text{, nonempty domain} \,\} \\
 & \big(\mathbf{MIN}\, m : 0 \leq m \wedge V^*(r{\uparrow}1)V^m \cap V^*P \neq \varnothing : m\big) + 1 \\
= & \quad \{\, \text{definition } \mathit{char}_{la} \text{ (after derivation)} \,\} \\
 & \mathit{char}_{la}(r{\uparrow}1) + 1
\end{array}
$$

where $\mathit{char}_{la} \in V \to \mathbb{N}$ is defined by

$$\mathit{char}_{la}(a) = \big(\mathbf{MIN}\, n : 0 \leq n \wedge V^*aV^n \cap V^*P \neq \varnothing : n\big) \quad (a \in V).$$

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

$$
\begin{array}{ll}
 & M(u,r) \\
= & \quad \{\, \text{property } \mathbf{max} \,\} \\
 & M(u,r)\, \mathbf{max}\, M(u,r) \\
\geq & \quad \{\, \text{derivation in subsection 3.1, preceding derivation} \,\} \\
 & N(u)\, \mathbf{max}\big(\mathit{char}_{la}(r{\uparrow}1) + 1\big)
\end{array}
$$

Since all shift functions derived in the previous subsections are approximations from below of $N(u)$ the preceding derivation shows that they all may be extended with $\mathbf{max}\big(\mathit{char}_{la}(r{\uparrow}1)+1\big)$ to form a class of stronger shift functions of signature $k(l,v,r)$ (algorithm detail (OLAU) (One symbol LookAhead at Unscanned part of the input string)). The first derivation in this subsection shows that it is also possible to couple the information on $r{\uparrow}1$ with the information on l and v ($u = lv$). We will not pursue that direction any further in this paper.

4 Factor-based sublinear pattern matching

We now derive a family of algorithms by using the set of factors of P, $\mathbf{fact}(P)$, for $\mathbf{f}(P)$ in Algorithm 2.9, i.e. instantiating $\mathbf{f}$ with $\mathbf{fact}$. We introduce

Algorithm detail 4.1. (GS=F). (Guard Strengthening = Factor). Strengthen the guard of the inner repetition by adding conjunct $(l\uparrow 1)v \in \mathbf{fact}(P)$. □

As with $(l\uparrow 1)v \in \mathbf{suff}(P)$ in Section 3, direct evaluation of $(l\uparrow 1)v \in \mathbf{fact}(P)$ is expensive. The transition function of an automaton recognizing the set $\mathbf{fact}(P)^R$ is used instead (detail choice (EGC=RFA)). Using function $\delta_{R,\mathbf{fact}}$ introduced in Section 2 and making $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\uparrow 1)v)^R)$ an invariant of the inner repetition, the guard becomes

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

Note that various automata exist whose transition functions can be used for $\delta_{R,\mathbf{fact}}$. One is the trie built on $\mathbf{fact}(P)^R$, another is the *suffix automaton* or the *dawg* (for directed acyclic word graph) on $\mathbf{fact}(P)^R$ [CH97]. The use of algorithm details (GS=F) and (EGC=RFA) leads to

Algorithm 4.2(P_+, S_+, GS=F, EGC=RFA)

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

This algorithm has $\Theta(|S|)$ running time, just as Algorithm 3.2 (P_+, S_+, GS=S, EGC=RSA).

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

- More character comparisons: In cases where $(l\uparrow 1)v \notin \mathbf{suff}(P)$ yet $(l\uparrow 1)v \in \mathbf{fact}(P)$, the guard of the inner loop will still be true, and hence the algorithm will go on extending v to the left more than strictly necessary.
- Larger shift distances with detail (SSD): When the guard of the inner loop becomes false, $(l\uparrow 1)v \notin \mathbf{fact}(P)$, which gives potentially more information to use in the shift function than $(l\uparrow 1)v \notin \mathbf{suff}(P)$. This aspect will be used in the derivations leading to algorithm detail (NFS) in Subsection 4.1.

We can introduce the notion of a safe shift distance, as was done for suffix-based algorithms in Section 3. This leads to:

Algorithm 4.3(P$_+$, S$_+$, GS=F, EGC=RFA, SSD)

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

Since $(l\restriction 1)v \notin \mathbf{fact}(P) \Rightarrow (l\restriction 1)v \notin \mathbf{suff}(P)$, we may use any of the safe shift functions derived for the suffix-based sublinear algorithm family[13]. As these have all been discussed in Section 3, we will not repeat them here.

4.1 The no-factor shift

We can do better than simply using the shift functions from Section 3, since $(l\restriction 1)v \notin \mathbf{fact}(P)$ is stronger than $(l\restriction 1)v \notin \mathbf{suff}(P)$. Taking the predicate $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$ from the domain of the approximation of the upperbound on k given in Subsection 3.1, we derive:

$\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$
$\equiv$ { $(\star)$ }
$\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing$
$\equiv$ { $\mathbf{suff}(x) \cap P \neq \varnothing \ \Rightarrow\ |x| \geq lmin_P$, $|v(r\restriction n)| = |v| + n$, duplication }
$\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P$ $(\star\star)$

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

- case $l = \varepsilon$:

  $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$
  $\equiv$ { $u = lv$, $l = \varepsilon$ }
  $\mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing$

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

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

[13]Note that in doing so, we implicitly replace the $\mathbf{suff}(P)$ part of their domain by $\mathbf{fact}(P)$. This means that precomputation of the functions changes. We do not further discuss this in this paper.

$$
\begin{aligned}
&\equiv \quad \{ \text{ Property B.7 } \} \\
&\quad \mathbf{suff}((l\lfloor 1)(l\lceil 1)v(r\restriction n)) \cap P \neq \varnothing \\
&\equiv \quad \{ \text{ property of } \mathbf{suff}\text{: } \mathbf{suff}(xay) = \mathbf{suff}(x)ay \cup \mathbf{suff}(y) \ \} \\
&\quad \Big(\mathbf{suff}(l\lfloor 1)\ (l\lceil 1)v(r\restriction n) \cup \mathbf{suff}(v(r\restriction n))\Big) \cap P \neq \varnothing \\
&\equiv \quad \{ \cap \text{ distributes over } \cup \ \} \\
&\quad \Big(\mathbf{suff}(l\lfloor 1)\ (l\lceil 1)v(r\restriction n) \cap P\Big) \cup \Big(\mathbf{suff}(v(r\restriction n)) \cap P\Big) \neq \varnothing \\
&\equiv \quad \{ \ (l\lceil 1)v \notin \mathbf{fact}(P) \equiv V^*(l\lceil 1)vV^* \cap P = \varnothing, \text{ hence } \mathbf{suff}(l\lfloor 1)\ (l\lceil 1)v(r\restriction n) \cap P = \varnothing \ \} \\
&\quad \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing
\end{aligned}
$$

Using the step marked ($\star$), we also have $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P$. Observe that the left conjunct—$\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$—is the predicate used to derive safe shift distances for suffix-based sublinear algorithms in Section 3. Let $Weakening(\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing)$ be any weakening of that predicate. Then:

$$
\begin{aligned}
&\quad \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P : n\big) \\
&\geq \quad \{ \ \} \\
&\quad \big(\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing) \wedge |v| + n \geq lmin_P : n\big) \\
&\geq \quad \{ \text{ Property B.2} \} \\
&\quad \big(\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing) : n\big) \\
&\qquad \mathbf{max}\ \big(\mathbf{MIN}\, n : 1 \leq n \wedge |v| + n \geq lmin_P : n\big) \\
&= \quad \{ \ \} \\
&\quad \big(\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing) : n\big)\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|))
\end{aligned}
$$

We may thus use any shift function

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

It is clear that the left operand of the outer **max** corresponds to any safe shift function from Section 3, represented by the various detail sequences given there. The right operand corresponds to the shift in case $(l\lceil 1)v$ is not a factor of a keyword. We introduce shift function $k_{ssd,nfs}(l, v, r)$ where $k_{ssd,nfs} \in V^* \times \mathbf{fact}(P) \times V^* \to \mathbb{N}$ is defined by

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

for any safe shift function k_{ssd} given in Section 3. We call it the no-factor shift, since it uses $(l\lceil 1)v \notin \mathbf{fact}(P)$.

In particular, we may use shift distance 1 with the no-factor shift. Using this, we derive a new shift distance:

$$
\begin{aligned}
&\quad 1\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|)) \\
&= \quad \{ \ \} \\
&\quad 1\ \mathbf{max}\ (lmin_P - |v|)
\end{aligned}
$$

This equals the shift distance used in the basic ideas for backward DAWG matching [NR02, page 27] and—combined with algorithm detail (LMIN) discussed in Section 2.1—set backward DAWG matching [NR02, page 68]. The actual Backward DAWG Matching [CCG+94], [NR02, page 28-29] and Set Backward DAWG Matching [CCG+94], [NR02, page 68] algorithms use an improvement based on a property of DAWGs. We discuss this in Subsection 4.2.

4.2 Cheap computation of a particular shift function

We now consider a different weakening of $\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing$ in the safe shift function predicate:

$$
\begin{array}{ll}
& \mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{ \text{ see } (\star) \text{ in Subsection 4.1 } \} \\
& \mathbf{suff}(v(r\restriction n)) \cap P \neq \varnothing \\
\equiv & \quad \{ \text{ introduce } last = \big(\mathbf{MAX}\, m : 0 \leq m \leq |v| \wedge v\restriction m \in \mathbf{pref}(P) : m\big) \ \} \\
& \mathbf{suff}((v\restriction last)(r\restriction n)) \cap P \neq \varnothing \\
\Rightarrow & \quad \{ \ r\restriction n \in V^n, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \ \} \\
& \mathbf{suff}((v\restriction last)V^n) \cap P \neq \varnothing
\end{array}
$$

We now derive

$$
\begin{array}{ll}
& \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}((v\restriction last)V^n) \cap P \neq \varnothing : n\big) \\
= & \quad \{ \text{ property of } \mathbf{suff}\text{: } \mathbf{suff}(A) \cap B \neq \varnothing \equiv A \cap V^*B \neq \varnothing \ \} \\
& \big(\mathbf{MIN}\, n : 1 \leq n \wedge (v\restriction last)V^n \cap V^*P \neq \varnothing : n\big) \\
\geq & \quad \{ \ last \leq |v| \text{ as result of definition } last,\ v\restriction last \in V^{last}, \text{ monotonicity of } \mathbf{suff} \text{ and } \cap \ \} \\
& \big(\mathbf{MIN}\, n : 1 \leq n \wedge V^{last+n} \cap V^*P \neq \varnothing : n\big) \\
\geq & \quad \{ \text{ Property B.2 } \} \\
& \big(\mathbf{MIN}\, n : 1 \leq n : n\big)\ \mathbf{max}\ \big(\mathbf{MIN}\, n : V^{last+n} \cap V^*P \neq \varnothing : n\big) \\
= & \quad \{ \text{ definition of } lmin_P \ \} \\
& 1\ \mathbf{max}\ (lmin_P - last)
\end{array}
$$

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

We introduce shift function k_{lskp} where $k_{lskp} \in \mathbf{fact}(P) \to \mathbb{N}$ is defined by

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

Note that this shift function does not depend on l. It can therefore be seen as a variant of algorithm detail (NLA) discussed in Subsection 3.7. The shift function does not directly depend on v either, but it indirectly depends on v due to its dependence on *last*. Calculating the shift distance using k_{lskp} is algorithm detail (LSKP). Using variable *last* and shift function k_{lskp}, the algorithm becomes:

Algorithm 4.4(P$_+$, S$_+$, GS=F, EGC=RFA, SSD, NLAU, OPT, NLA, LSKP)

$u, r := \varepsilon, S$;
if $\varepsilon \in P \to O := \{(\varepsilon, \varepsilon, S)\}$ $[\!]$ $\varepsilon \notin P \to O := \varnothing$ **fi**;
$l, v := \varepsilon, \varepsilon$;
$last := 0$;
{ invariant: $ur = S \wedge O = \big(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\big)$
$\wedge\ u = lv \wedge v \in \mathbf{fact}(P)$
$\wedge\ \big(l = \varepsilon\ \mathbf{cor}\ (l\restriction 1)v \notin \mathbf{fact}(P)\big)$ }

do $r \neq \varepsilon \rightarrow$
  $k := 1\ \mathbf{max}(lmin_P - last)$;
  $u, r := u(r\restriction k), r\downarrow k;\ l, v := u, \varepsilon$;
  $q, last := \delta_{R,\mathbf{fact}}(q_0, l\restriction 1), 0$;
  as $\varepsilon \in P \rightarrow O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
  { invariant: $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\restriction 1)v)^R)$
    $\wedge\ last = \big(\mathbf{MAX}\, m : m \leq |v| \wedge v\restriction m \in \mathbf{pref}(P) : m\big)$ }
  do $l \neq \varepsilon$ **cand** $q \neq \perp \rightarrow$
    $l, v := l\downarrow 1, (l\restriction 1)v$;
    $q := \delta_{R,\mathbf{fact}}(q, l\restriction 1)$
    as $q \in F \rightarrow last := |v|$ **sa**;
    as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
  od
od{ R }

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

Algorithm detail (NFS) is not included in either of these two detail sequences, since the no-factor shift can never be larger than the shift according to k_{lskp}:[14]

$$
\begin{array}{ll}
 & lmin_P - |v| \\
\leq & \quad \{\ last \leq |v|\ \} \\
 & lmin_P - last \\
= & \quad \{\ \text{definition } k_{lskp}\ \} \\
 & k_{lskp}
\end{array}
$$

In addition, we note that the quantification $\big(\mathbf{MIN}\, n : 1 \leq n \wedge (v\restriction last)V^n \cap V^*P \neq \varnothing : n\big)$ in the second line of the last derivation above equals $d_{sp}(v\restriction last)$ as defined in Subsection 3.2. It follows that shift function k_{lskp} gives an approximation from below of that function.

[14] We do not include algorithm detail LMIN in the detail sequence of the single-keyword Backward DAWG algorithm either, since the addition of algorihm detail LMIN does not influence the algorithm when combined with problem detail OKW.

5 Factor oracle-based sublinear pattern matching

We now derive a family of algorithms by using $\mathbf{factoracle}(P^R)$, the language of a factor oracle on P^R. A factor oracle is a deterministic finite automaton built on a set of words that recognizes at least all factors of these words, but possibly more. In addition, if a word is recognized, all suffixes of that word are recognized as well. In other words, a factor oracle recognizes a superset of $\mathbf{fact}(P^R)$ and is suffix-closed[15]. Even though the exact language recognized by a factor oracle is not yet known, we can strengthen the guard of the inner repetition by choosing $\mathbf{factoracle}(P^R)^R$ for $\mathbf{f}(P)$, since $P \subseteq \mathbf{factoracle}(P^R)^R$ and $\mathbf{suff}(\mathbf{factoracle}(P^R)^R) \subseteq \mathbf{factoracle}(P^R)^R$ both hold. The inner repetition guard now becomes

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

Since the exact definition of **factoracle** independent of the factor oracle automaton is currently unknown, direct evaluation of $(l\restriction 1)v \in \mathbf{factoracle}(P^R)^R$ is not possible. The transition function of the factor oracle (see [CZW04, CZW03], as well as [ACR01, AR99]) recognizing the set $\mathbf{factoracle}(P^R)$ is therefore used. Using function $\delta_{\mathbf{factoracle}(P^R)}$[16] and making $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l\restriction 1)v)^R)$ an invariant of the inner repetition (algorithm detail (EGC=RFO)), the guard becomes

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

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

Algorithm 5.1(P$_+$, S$_+$, GS=FO, EGC=RFO)

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

This algorithm has $\Theta(|S|)$ running time, just like Algorithm 3.2 (P$_+$, S$_+$, GS=S, EGC=RSA) and Algorithm 4.2 (P$_+$, S$_+$, GS=F, EGC=RFA).

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

- Easier construction of and more compact automata: The factor oracle recognizing the words in $\mathbf{factoracle}(P^R)$ is easier to construct and may have less states and transitions than an automaton recognizing $\mathbf{fact}(P^R)$ (see [CZW04, CZW03, ACR01, AR99]).

[15] We prove this for the single keyword factor oracle in [CZW04, CZW03], and it is proven for the multiple keyword version in [AR99]. Proofs for the single keyword version can be found in [ACR01] as well.

[16] Since it is possible that $\mathbf{factoracle}(P)^R \neq \mathbf{factoracle}(P^R)$, we cannot use $\delta_{R,\mathbf{factoracle}}$ to describe the transition function of the automaton used. We therefore introduce the notation $\delta_{\mathbf{factoracle}(P^R)}$, the transition function of the automaton recognizing $\mathbf{factoracle}(P^R)$.

- More character comparisons: When $(l\restriction 1)v \notin \mathbf{fact}(P)$ yet $(l\restriction 1)v \in \mathbf{factoracle}(P^R)^R$, the guard of the inner loop will still be true, and hence the algorithm will go on extending v to the left more than strictly necessary.

Note that the effects of using (GS=FO, EGC=RFO) instead of (GS=S, EGC=RSA) are a combination of the effects mentioned here and those described in Section 4 when comparing (GS=F, EGC=RFA) and (GS=S, EGC=RSA).

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

Algorithm 5.2(P$_+$, S$_+$, GS=FO, EGC=RFO, SSD)

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

We derive

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

Therefore any shift function may be used satisfying

$$\big(\mathbf{MIN}\, n : 1 \leq n \wedge Weakening(\mathbf{suff}(u(r\restriction n)) \cap P \neq \varnothing) : n\big)\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|))$$

as derived for factor-based algorithms in Section 4. In particular, both the safe shift functions for the suffix-based algorithms as well as the no-factor shift introduced in Section 4 may be used.

The Set Backward Oracle Matching algorithm [AR99], [NR02, pages 69-72] equals our algorithm (P$_+$, S$_+$, GS=FO, EGC=RFO, LMIN, SSD, NFS, ONE), while the single keyword Backward Oracle Matching algorithm [ACR01], [NR02, pages 34-36], [CZW04, CZW03] corresponds to (P$_+$, S$_+$, GS=FO, EGC=RFO, SSD, NFS, ONE, OKW).

6 Final remarks

We presented a revised and expanded version [Cle03] of the original taxonomy of sublinear keyword pattern matching algorithms [WZ95, WZ96, Wat95], giving new derivations for various algorithms and placing them in the taxonomy. The use of formal techniques made it relatively easy to extend and generalize the taxonomy.

In particular, we showed how suffix-, factor- and factor oracle-based sublinear keyword pattern matching algorithms can all be seen as instantiations of a general sublinear algorithm skeleton. In addition, we have shown all shift functions defined for the suffix-based algorithms to be in principle reusable for factor- and factor oracle-based algorithms.

The algorithms described here could also be described using a generalization of the alternative Commentz-Walter algorithm skeleton presented in [Wat00], in which a move of the output variable update out of a loop may substantially increase performance. In addition to changes to the algorithms in the taxonomy to accomodate this idea, benchmarking would also need to be performed to study the effects.

We have not considered precomputation of the various shift functions used in the algorithms discussed in this paper. Precomputation of these functions for suffix-based algorithms was described in [WZ95, WZ96], but extending this precomputation to factor- and factor oracle-based algorithms remains to be done.

Although we have extended the original taxonomy of keyword pattern matching algorithms by adding some algorithm variants and generalizing the suffix-based sublinear algorithm skeleton in order to include factor- and factor oracle-based sublinear algorithms as well, there are still some keyword pattern matching algorithms that we have not considered. Possible future work on the sublinear part of the taxonomy includes:

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

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

3. Deriving Sunday's variant of Boyer-Moore-Horspool [Sun90].

4. Looking into the possibility of a bit-parallel suffix-based pattern matching algorithm, i.e. a bit-parallel version of (suffix-based) Commentz-Walter.

References

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

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

[AG97] A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[Aho90] A.V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, volume A, pages 255–300. Elsevier, Amsterdam, 1990.

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

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

[BS02] Gabor Barla-Szabo. A taxonomy of graph representations. Master's thesis, Department of Computer Science, University of Pretoria, November 2002.

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

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

[Cle03] Loek G.W.A. Cleophas. Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms. Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, August 2003.

[CR03] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology - Text Algorithms*. World Scientific Publishing, 2003.

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

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

[CZW03] Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. In *Proceedings of the Prague Stringology Conference 2003*. Department of Computer Science and Engineering, Czech Technical University, Prague, 2003.

[CZW04] Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. Technical Report 04/01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2004.

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

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

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, New York, NY, 1990.

[FS93] J.-J. Fan and K.-Y. Su. An efficient algorithm for matching multiple patterns. *IEEE Trans. Knowledge and Data Eng.*, 5:339–351, 1993.

[FS94] J.-J. Fan and K.-Y. Su. An efficient algorithm for matching multiple patterns. In J. Aoe, editor, *Computer Algorithms: String Pattern Matching Strategies*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

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

[Jon83] H.B.M. Jonkers. Abstraction, specification and implementation techniques, with an application to garbage collection. Technical Report 166, Mathematisch Centrum, Amsterdam, 1983.

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

[Mis93] Frederick C. Mish, editor. *Merriam Webster's Collegiate Dictionary*. Merriam Webster, Springfield, MA, 10th edition, 1993.

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

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

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

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

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

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

[Wat04] Bruce W. Watson. *Constructing minimal acyclic deterministic finite automata*. PhD thesis, Department of Computer Science, University of Pretoria, 2004.

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

[WZ95] B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. Technical Report 95/13, Faculty of Computing Science, Technische Universiteit Eindhoven, 1995.

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

A Algorithm and problem details

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

P	Examine prefixes of a given string in any order.
P_+	Examine prefixes of a given string in order of increasing length.
S	Examine suffixes of a given string in any order.
S_+	Examine suffixes of a given string in order of increasing length.
GS=S	Use guard strengthening to increment the length of a suffix only for as long as a string which is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.
GS=F	Use guard strengthening to increment the length of a suffix only for as long as a string which is a factor of some keyword, preceded by a symbol is again a factor of some keyword.
GS=FO	Use guard strengthening to increment the length of a suffix only for as long as a string whose reverse is part of the language of the factor oracle on the reverse of the set of keywords, preceded by a symbol is again part of that language.
GS=SO	Use guard strengthening to increment the length of a suffix only for as long as a string whose reverse is part of the language of the suffix oracle on the reverse of the set of keywords, preceded by a symbol is again part of that language.
EGC=RSA	Usage of automaton recognizing the reverse of the set of suffixes of the keywords for efficient guard computation, i.e. to check whether a string which is a suffix of some keyword, preceded by a symbol is again a suffix of some keyword.
EGC=RFA	Usage of an automaton recognizing the reverse of the set of factors of the keywords for efficient guard computation, i.e. to check whether a string which is a factor of some keyword, preceded by a symbol is again a factor of some keyword.
EGC=RFO	Usage of a factor oracle on the reverse of the keywords for efficient guard computation, i.e. to check whether a string which is part of the language of the factor oracle, preceded by a symbol is again part of the language of that factor oracle.
LMIN	When using an automaton in one of the (EGC) details, construct this automaton on the prefixes of length equal to the length of the shortest keyword instead of on the complete keywords.
SSD	Consider any shift distance that does not lead to the missing of any matches. Such shift distances are called *safe*.
ONE	Use a safe shift distance of 1.
NLAU	No lookahead at the symbols of the unscanned part of the input string when computing a safe shift distance.
OLAU	One symbol lookahead at the unscanned part of the input string when computing a safe shift distance.

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

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

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

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

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

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

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

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

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

B Definitions

Notation B.1 (Quantifications). A basic understanding of the meaning of *quantifications* is assumed. We use the following notation:

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

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

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

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

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

□

Property B.2 (Conjunction and disjunction in MIN quantifications). For predicates P, Q and integer function f we have

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

□

Definition B.3 (Nondeterministic algorithm). An algorithm is called *nondeterministic* if the order in which (some of) its statements can be executed is not fixed, or if the guards in a selection statement are not mutually exclusive. □

Notation B.4 (Conditional conjunction/disjunction). We use **cand** and **cor** for *conditional conjunction* and *conditional disjunction* respectively. A conditional conjunction (disjunction) is one in which the second operand is evaluated if and only if this is necessary to determine the value of the conjunction (disjunction). □

Definition B.5 (String reversal function R). Assuming alphabet V, we define string reversal function R recursively by $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$ (for $a \in V$, $w \in V^*$). We will use R on sets of strings as well. □

Definition B.6 (String operators $\uparrow$, $\downarrow$, $\restriction$, $\downharpoonright$). Assuming alphabet V, we define four infix operators $\uparrow, \downarrow, \restriction, \downharpoonright \in V^* \times \mathbb{N} \to V^*$ as follows:

- $w{\uparrow}k$ is the $k\,\mathbf{min}\,|w|$ leftmost symbols of w
- $w{\downarrow}k$ is the $(|w| - k)\,\mathbf{max}\,0$ rightmost symbols of w
- $w{\restriction}k$ is the $k\,\mathbf{min}\,|w|$ rightmost symbols of w
- $w{\downharpoonright}k$ is the $(|w| - k)\,\mathbf{max}\,0$ leftmost symbols of w

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

Property B.7 (String operators $\uparrow$, $\downarrow$, $\restriction$, $\downharpoonright$). For string operator $\uparrow$, $\downarrow$, $\restriction$ and $\downharpoonright$,

$$(w{\uparrow}k)(w{\downarrow}k) = w$$
$$(w{\downharpoonright}k)(w{\restriction}k) = w$$

□

Example B.8 (String operators $\uparrow$, $\downarrow$, $\restriction$, $\downharpoonright$). $(hers){\uparrow}3 = her$, $(hers){\downarrow}1 = ers$, $(hers){\restriction}5 = hers$ and $(hers){\downharpoonright}10 = \varepsilon$. □

Definition B.9 (Functions pref, suff and fact). For any given alphabet V, define $\mathbf{pref} \in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$, $\mathbf{suff} \in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ and $\mathbf{fact} \in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ as

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

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

Notation B.10 (String arguments to functions pref, suff and fact). For string $w \in V^*$, we will write $\mathbf{pref}(w)$ ($\mathbf{suff}(w)$, $\mathbf{fact}(w)$) instead of $\mathbf{pref}(\{w\})$ ($\mathbf{suff}(\{w\})$, $\mathbf{fact}(\{w\})$). □

Property B.11 (Idempotence of pref, suff and fact). **pref**, **suff** and **fact** are idempotent. □

Property B.12 (Relationship between fact and suff, pref). Function **fact** can also be defined in terms of the functions **suff** and **pref**:

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

and

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

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

$y \in \mathbf{pref}(\mathbf{suff}(L))$

$=$ { definition of **pref** }

$(\exists z :: yz \in \mathbf{suff}(L))$

$=$ { property of **suff** }

$(\exists z :: (\exists x :: xyz \in L))$

$=$ { nesting }

$(\exists x, z :: xyz \in L)$

$\equiv$ { definition of **fact** }

$y \in \mathbf{fact}(L)$

□

Property B.13 (Duality of pref and suff). Functions **pref** and **suff** are each other's duals. This can be seen as follows:

$x \in \mathbf{pref}(L^R)$

$\equiv$ { property of **pref** }

$(\exists y :: xy \in L^R)$

$\equiv$ { property of operator R }

$(\exists y :: y^R x^R \in L)$

$\equiv$ { change of bound variable: $y' = y^R$ }

$(\exists y' :: y' x^R \in L)$

$\equiv$ { property of **suff** }

$x^R \in \mathbf{suff}(L)$

$\equiv$ { property of operator R }

$x \in \mathbf{suff}(L)^R$

□

Property B.14 (Symmetry of fact). Function **fact** is symmetrical. This can be seen as follows:

$\mathbf{fact}(L^R)$

$\equiv$ { Property B.12 }

$\mathbf{pref}(\mathbf{suff}(L^R))$

$\equiv$ { (dual of) Property B.13 }

$\mathbf{pref}(\mathbf{pref}(L)^R)$

$\equiv$ { Property B.13 }

$\mathbf{suff}(\mathbf{pref}(L))^R$

$\equiv$ { Property B.12 }

$\mathbf{fact}(L)^R$

□

Definition B.15 (Prefix and suffix partial orderings). Partial orders $\leq_p$, $<_p$, $\leq_s$ and $<_s$ over $V^* \times V^*$ are defined as

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

□

Property B.16 (Language intersection). If A and B are languages over alphabet V and $a \in V$, then

$$\begin{array}{lcl} V^*A \cap V^*B \neq \varnothing & \equiv & V^*A \cap B \neq \varnothing \quad \vee \quad A \cap V^*B \neq \varnothing \\ V^*aA \cap V^*B \neq \varnothing & \equiv & V^*aA \cap B \neq \varnothing \quad \vee \quad A \cap V^*B \neq \varnothing \end{array}$$

□

Definition B.17 ((Deterministic) Finite Automaton). A (deterministic) finite automaton is a 5-tuple $M = \langle Q, V, \delta, q_0, F \rangle$ where

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

□

Definition B.18 (Extending transition relation δ). We extend transition relation $\delta \in Q \times V \to Q$ to $\delta^* \in Q \times V^* \to Q$ defined by

$$\begin{aligned} \delta^*(q, \varepsilon) &= q \\ \delta^*(q, wa) &= \delta(\delta^*(q, w), a) \end{aligned}$$

□