Science of Computer Programming 75 (2010) 1095–1112

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms

Loek Cleophas [a,b,*], Bruce W. Watson [a], Gerard Zwaan [b]

[a] FASTAR Research Group, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa

[b] Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 10 April 2008
Received in revised form 23 April 2010
Accepted 25 April 2010
Available online 15 May 2010

Keywords:
Finite automata
Algorithm taxonomies
Pattern matching

ABSTRACT

A new taxonomy of sublinear (multiple) keyword pattern matching algorithms is presented. Based on an earlier taxonomy by the second and third authors, this new taxonomy includes not only suffix-based algorithms, but also factor- and factor-oracle-based algorithms. In particular, we show how suffix-based (Commentz-Walter like), factor- and factor-oracle-based sublinear keyword pattern matching algorithms can be seen as instantiations of a general sublinear algorithm skeleton. During processing, such algorithms shift or jump through the text in a forward or left-to-right direction, and read backward or right-to-left starting from positions in the text, i.e. they read suffixes of certain prefixes of the text. They use finite automata for efficient computation of string membership in a certain language. In addition, we show shift functions defined for the suffix-based algorithms to be 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, 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.

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" [38]. This problem has been frequently studied in the past, and many different algorithms have been suggested for solving it. Watson and Zwaan (in [38], [35, Chapter 4]) derived a set of well-known solutions to the problem from a common starting point, factoring out their commonalities and presenting them in a common setting to better comprehend and compare them. Their taxonomy of solutions included prefix-based Knuth–Morris–Pratt [27] and Aho–Corasick [2], suffix-based Boyer–Moore [7], Commentz-Walter [14,15], and many variants of these four algorithms. Other overviews of keyword pattern matching have recently been given by Crochemore et al. [19], Smyth [32], Crochemore and Rytter [20], Apostolico and Galil [5] and by Navarro and Raffinot [31].

Although the original taxonomy contained a large number of variations on these four algorithms, some efficient variants were not included. Among these are the single and multiple keyword Boyer–Moore–Horspool algorithms [25,31]. Most

* Corresponding author at: Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
E-mail addresses: loek@loekcleophas.com (L. Cleophas), bruce@fastar.org (B.W. Watson), g.zwaan@tue.nl (G. Zwaan).
URLs: http://www.fastar.org (L. Cleophas, B.W. Watson), http://www.win.tue.nl/set (L. Cleophas, G. Zwaan).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.04.012

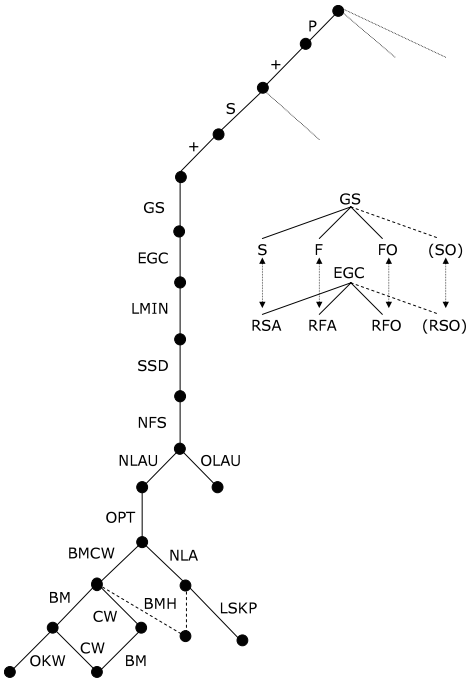

Fig. 1. A new taxonomy of sublinear keyword pattern matching algorithms. Edges are directed top-down. The algorithm and problem details used are introduced in Sections 2 through 5 as part of the running text. A complete list with a short description of each detail can also be found in Appendix.

importantly however, a new category of algorithms – based on factors instead of prefixes or suffixes of keywords – has emerged since the construction of the original taxonomy. This category includes algorithms such as (Set) Backward DAWG Matching [16,30] and (Set) Backward Oracle Matching [3,4], which were added to the existing taxonomy by Cleophas [12].

We focus our attention on the part of the new taxonomy containing (multiple) keyword pattern matching algorithms that read from right to left starting from certain positions in the text and that have potentially sublinear matching time; that is, the number of symbol comparisons may be sublinear in the length of the input string.

Fig. 1 shows this taxonomy part. Nodes 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 Ph.D. Thesis [35, Chapter 4], while the part representing sublinear keyword pattern matching algorithms is described in Watson and Zwaan's [38]. Sections 3.1 through 3.7 and Section 3.9 of this paper are based on 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 [38]. We do not discuss such precomputation here. The additions and changes to the original taxonomy to form the new taxonomy are described in Cleophas' M.Sc. Thesis [12, Chapter 3].

In [8], an earlier and less extensive version of the current paper, we presented the generalized sublinear keyword pattern matching algorithm skeleton used in this paper, albeit with a focus on automata and restricted to factor- and factor-oracle-based algorithms.

An implementation of most algorithms from 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 [12, Chapter 5]. The toolkit will be available for non-commercial use from http://www.fastar.org.

1.2. Basic algorithm and derivation principles

The pattern matching algorithms that we consider in the current paper inspect suffixes of prefixes of the input string both in order of increasing length. In considering suffixes, the algorithms increase this length as long as the suffix read is a suffix or factor of a keyword. Choosing such a basic approach we have the possibility to attain matching times that are sublinear in the length of the input string (i.e. not all symbols of the input string are inspected). It is achieved by taking steps through the input string of which the length is determined by a shift function based on the information of the last matching attempt and

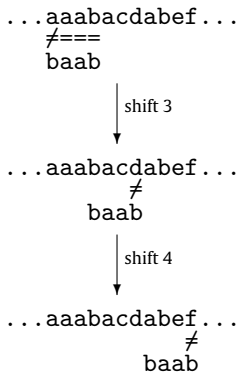

Fig. 2. Example of larger shift distances.

possibly on additional information. The example in Fig. 2 illustrates this principle. Notice that not all symbols of the input string are scanned, although some symbols of the input string are scanned more than once. The simplest shift function is the one that always yields 1. However, one can imagine larger shifts being possible. Ideally, such a shift would take us to the next occurrence of a match, but then calculating the value of the shift function is equivalent to the pattern matching problem itself. Therefore, we strive for shift functions that are easier to calculate and that do not exceed the ideal shift (called *safe shift functions*), i.e. we aim at approximations of the ideal shift from below. The ideal shift function is the minimum over a range characterized by some predicate. We derive various approximations from below by systematically weakening this predicate and derived predicates. Considerations that play a role in these derivations are, for instance, whether or not to look ahead at symbols of the unscanned part of the input string, what information to use about the last scanned (non-matching) symbol, and the extent to which this information is coupled with that on the recognized suffix, factor, or factor oracle term. Thus, we obtain several safe shift functions, leading to an equal number of algorithms. The simplest weakening of the predicate its that to true, yielding the shift function that is always equal to 1. Smaller shift functions are derived since they typically take less precomputation time and less storage space (for instance, a one-dimensional table instead of a two-dimensional table).

In Watson and Zwaan's original paper the algorithms included are those based on extending matching to the left as long as what is read is a *suffix* of some keyword [38]. This included algorithms such as Boyer–Moore, Commentz-Walter, as well as their common ancestors (including Fan and Su's algorithm [23,24]) and a common descendant. In the current paper, algorithms reading backward as long as what is read is a *factor* of such a keyword are newly included. Furthermore, the Boyer–Moore like Boyer–Moore–Horspool algorithm [25], based on suffixes, is newly included.

1.3. 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 [29, p. 1208].

Although this definition is somewhat biology oriented, we can create a classification according to essential details of algorithms or data structures in a certain field as well. In our case, such a classification takes the form of a *(directed acyclic) taxonomy graph*. (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.)

The construction of a taxonomy has a number of goals:

- Providing algorithm correctness arguments, often absent in literature.
- Clarifying the algorithms' working and interrelationships.
- Helping in correctly and easily implementing the algorithms [37,35].
- Leading to new algorithms, by finding and filling gaps in the taxonomy.

The process of taxonomy construction is preceded by surveying the existing literature of algorithms in the problem field 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 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 distinguishing the various algorithms each belong to one of the following categories:

- *Problem details* involve minor changes to pre- and postconditions, restricting in- or output. For example, restricting the input of the keyword pattern matching problem from a set of keywords to a single one. (Other problem details that could be considered would be to allow not just keywords, but also patterns including wild cards or character classes.)
- *Algorithm details* are used to specify variance in algorithmic structure. In the case of keyword pattern matching, these deal with e.g. the particular shift value function used to update the current reading position in the input text.
- *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. For example, they might include the representation of a finite automaton for pattern matching, including the use of bit-parallelism.
- *Performance details* are about variance in running time and memory consumption.

As the representation and performance details mainly influence implementation but do not influence the other goals stated above, problem and algorithm details are most important. These details form the taxonomy graph edges.

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). As one sees commonalities among them, one may find generalizations which allow combining multiple taxonomies into one larger one with the new generalization as the root. Once a complete taxonomy has been constructed, it is presented top-down.

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 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 been used for garbage collection algorithms [26], finite automata construction and minimization algorithms [35], graph representations [6], finite tree automata construction and tree acceptance/tree pattern matching algorithms [13], and other problem fields.

1.4. Notation

We will often use notations corresponding to their use in existing literature on specific algorithms, as a large part of this paper consists of derivations of existing algorithms. Nevertheless, we tried to adopt standard conventions for naming variables, functions and sets whenever possible. We use A and B for arbitrary sets, $\mathcal{P}(A)$ for the powerset of a set A, V for the (non-empty and finite) alphabet and V^* for words over the alphabet, $P = \{p_0, p_1, \ldots, p_{|P|-1}\} \subseteq V^*$ for a finite, non-empty pattern set with $lmin_P = \big(\mathbf{MIN}\, p : p \in P : |p|\big)$, as well as R for predicates, M for finite automata and Q for state sets. States are represented by q and q_0. Symbols $a, b, \ldots, e$ represent alphabet symbols from V, while $p, s, \ldots, z$ represent words over alphabet V. Symbols $i, j, \ldots, n$ represent integer values. We use $\perp$ ('bottom') to denote an undefined value. Sometimes functions, relations or predicates are used that have names longer than just a single character.

A (deterministic) finite automaton ((D)FA) is a 5-tuple $M = \langle Q, V, \delta, q_0, F\rangle$ where Q is a finite set of states, $\delta \in Q \times V \to Q$ is the transition relation, $q_0 \in Q$ is a start state and $F \subseteq Q$ is a set of final states. We extend δ to $\delta^* \in Q \times V^* \to Q$ defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.

We use p^R for the reversal of a string p, and use string reversal on a set of strings as well. A string u is a *factor* (resp. *prefix*, *suffix*) of a string v if $v = sut$ (resp. $v = ut$, $v = su$). We use functions **fact**, **pref** and **suff** for the set of factors, prefixes and suffixes of a (set of) string(s) respectively. We write $u \leq_p v$ to denote that u is a prefix of v. The infix operators $\uparrow$, $\downarrow$, $\restriction$, $\llcorner$ (pronounced 'left take', 'left drop', 'right take' and 'right drop' respectively) for $0 \leq k$ are defined as: $w{\uparrow}k$ is the k **min** $|w|$ leftmost symbols of w, $w{\downarrow}k$ is the $(|w| - k)$ **max** 0 rightmost symbols of w, $w{\restriction}k$ is the k **min** $|w|$ rightmost symbols of w and $w{\llcorner}k$ is the $(|w| - k)$ **max** 0 leftmost symbols of w. For example, $(hers){\uparrow}3 = her$, $(hers){\downarrow}1 = ers$, $(hers){\restriction}5 = hers$ and $(hers){\llcorner}10 = \varepsilon$.

A basic understanding of *quantifications* is assumed. We use the notation $(Q_\oplus\, a : R(a) : E(a))$ where $Q_\oplus$ is the *quantifier symbol* associated with an associative and commutative binary operator $\oplus$ (with unit $e_\oplus$), a is the *quantified variable* introduced, R is the *range predicate* on a, and E is the *quantified expression*. By definition, we have $(Q_\oplus\, a : false : E(a)) = e_\oplus$.

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

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

We use predicate calculus in derivations [22] and present algorithms in an extended version of (part of) the guarded command language [21]. In that language, $x, y := X, Y$ is used for multiple-variable assignment, while **if** $b \to S$ [] $\neg b \to T$ **fi** represents executing S if b evaluates to true, and T if $\neg b$ evaluates to true. The extensions of the basic language are **as** $b \to S$ **sa** as a shortcut for **if** $b \to S$ [] $\neg b \to$ skip **fi**, and **for** $x : R \to S$ **rof** for executing statement list S once for each value of x initially satisfying R (assuming there is a finite number of such values for x), in arbitrarily chosen order [34]. We also use **cand** (**cor**) for *conditional con-* resp. *disjunction*, i.e. the second operand is evaluated if and only if necessary to determine the value of the con- or disjunction.

Algorithm and problem details used will be introduced in the course of the text, but a list and description of the details is also available in Appendix.

1.5. 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

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 input string $S \in V^*$ and pattern set P, is to establish

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

that is to let O be the set of triples forming a splitting of S in three such that the middle part – v – is a keyword in P.

A trivial (but unrealistic) solution to the problem is

Algorithm 2.1 ()

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

The sequence of details describing this algorithm is the empty sequence (sequences of details are introduced in Section 1.3 and Fig. 1).

We may proceed by considering a substring of S as "suffix of a prefix of S" or as "prefix of a suffix of S". We choose the first possibility as this is the way that the algorithms we consider treat substrings of input string S. The second leads to algorithms processing S from right to left instead. Applying "examine prefixes of a given string in any order" (algorithm detail (P)) to S, we obtain:

Algorithm 2.2 (P)

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

This algorithm is used as a starting point in [12,35] to derive prefix-based algorithms such as Aho–Corasick, Knuth–Morris–Pratt and Shift-And/-Or. These algorithms – although very efficient for a number of situations – are not discussed in this paper, as their behaviour is not sublinear.

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:

Algorithm 2.3 (P, S)

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

Algorithm (P, S) consists of two nested non-deterministic repetitions. Each can be determinized by considering prefixes (or suffixes as the case is) in increasing (called detail (+)) or decreasing (detail (−)) order of length. Since the algorithms we consider achieve sublinear behaviour by examining string S from left to right, and patterns in P from right to left, we focus our attention on:

Algorithm 2.4 (P_+, S_+)

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

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

To obtain a more efficient algorithm, we strengthen inner loop guard $l \neq \varepsilon$. In [38,35], it was strengthened by adding **cand** $(l\uparrow 1)v \in$ **suff**(P). A more general strengthening is possible. Suppose we have a function $\mathbf{f} \in \mathcal{P}(V^*) \to \mathcal{P}(V^*)$ satisfying $P \subseteq \mathbf{f}(P) \wedge$ **suff**$(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$ (i.e. $\mathbf{f}$ is such that P is included in $\mathbf{f}(P)$ and $\mathbf{f}(P)$ is suffix-closed) then we have (for all $w, x \in V^*$) $w \notin \mathbf{f}(P) \Rightarrow w \notin P$ and $w \notin \mathbf{f}(P) \Rightarrow xw \notin P$ (application of right conjunct followed by left one). We may therefore strengthen the guard to $l \neq \varepsilon$ **cand** $(l\uparrow 1)v \in \mathbf{f}(P)$ (algorithm detail (GS), for guard strengthening). This leads to:

Algorithm 2.5 (P_+, S_+, GS)

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

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

Several choices for $\mathbf{f}(P)$ are possible, of which we mention:

- **suff**(P), leading to the original sublinear keyword pattern matching taxonomy. This choice will be discussed in Section 3.
- **fact**(P), discussed in Section 4.
- **factoracle**$(P^R)^R$, a superset of **fact**$(P^R)^R$ (= **fact**(P)); an implementation of this choice using a kind of automata called *factor oracle* is discussed in Section 5.
- A function returning a superset of **suff**(P). This could be implemented using **sufforacle**, i.e. the function defining the language recognized by a *suffix oracle* [3,4] on a set of keywords. We will not explore this option here.

Direct evaluation of $(l\uparrow 1)v \in \mathbf{f}(P)$ is expensive. To efficiently compute it, 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 2.6 (*Transition Function of Automaton Recognizing* $\mathbf{f}(P)^R$). *For transition function* $\delta_{R,\mathbf{f},P}$ *of a (weakly deterministic) finite automaton* $M = \langle Q, V, \delta_{R,\mathbf{f},P}, q_0, F\rangle$ *recognizing* $\mathbf{f}(P)^R$, *the property that* $\delta^*_{R,\mathbf{f},P}(q_0, w^R) \neq \bot \equiv w^R \in \mathbf{f}(P)^R$ *holds.*

Note that Property 2.6 requires $\mathbf{f}(P)^R$ to be prefix-closed (i.e. $\mathbf{pref}(\mathbf{f}(P)^R) \subseteq \mathbf{f}(P)^R$) hence $\mathbf{f}(P)$ to be suffix-closed (i.e. $\mathbf{suff}(\mathbf{f}(P)) \subseteq \mathbf{f}(P)$).

Since we will always refer to the same set P, we will use $\delta_{R,\mathbf{f}}$ instead of $\delta_{R,\mathbf{f},P}$. This transition function $\delta_{R,\mathbf{f}}$ can be computed beforehand, as done for $\mathbf{f} = \mathbf{suff}$ in [38, Section 4.1]. The transition function is called τ_P there, while it is called $\tau_{P,r}$ in [35] to distinguish it from the *forward trie* function. We generalize the function by means of a parameter $\mathbf{f}$ and make it into a transition function on automata.

By making $q = \delta^*_{R,\mathbf{f}}(q_0, ((l\uparrow 1)v)^R)$ an invariant of the algorithm's inner repetition, guard conjunct $(l\uparrow 1)v \in \mathbf{f}(P)$ can be changed to $q \neq \bot$. We call this algorithm detail (EGC), for efficient guard computation. This algorithm detail leads to the following algorithm skeleton:

Algorithm 2.7 (P_+, S_+, GS, EGC)

$u, r := \varepsilon, S;$
if $\varepsilon \in P \to O := \{(\varepsilon, \varepsilon, S)\}$ [] $\varepsilon \notin P \to O := \varnothing$ **fi**;
$\{$ invariant: $ur = S$
$\quad \wedge\ O = \left(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\right)$ $\}$
do $r \neq \varepsilon \to$
$\quad u, r := u(r\uparrow 1), r\downarrow 1;\ \ l, v := u, \varepsilon;\ \ q := \delta_{R,\mathbf{f}}(q_0, l\uparrow 1);$
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad \{$ invariant: $u = lv \wedge v \in \mathbf{f}(P) \wedge q = \delta^*_{R,\mathbf{f}}(q_0, ((l\uparrow 1)v)^R)$ $\}$
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \bot \to$
$\quad\quad l, v := l\downarrow 1, (l\uparrow 1)v;$
$\quad\quad q := \delta_{R,\mathbf{f}}(q, l\uparrow 1);$
$\quad\quad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
$\quad \{\ l = \varepsilon$ **cor** $(l\uparrow 1)v \notin \mathbf{f}(P)\ \}$
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. in $\Theta(1)$, by providing a map from states of the automaton to booleans. The construction of such a map may require quite some precomputation time. We do not consider relative precomputation times of the various algorithms, since they are both relatively hard to compare in terms of Θ notation and are assumed to be relatively small compared to the time taken to perform the actual pattern matching (i.e. we assume the text on which the matching is performed to be relatively long).

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 $lmin_P = \left(\mathbf{MIN}\, p : p \in P : |p|\right)$) 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 (LMIN)). Note that this algorithm detail could be applied to *any* of the pattern matching algorithms in the taxonomy shown in Fig. 1.

As a result of using algorithm detail (LMIN) with Algorithm 2.7 (P_+, S_+, GS, EGC), after assignment $l, v := l\downarrow 1, (l\uparrow 1)v$ in the inner loop, $v \in \mathbf{f}(P')$ holds (instead of $v \in \mathbf{f}(P)$ as before). Due to Property 2.6, in case $|v| = lmin_P$ (i.e. $v \in P'$) we need to verify any matches $v(r\uparrow i) \in P$ for $i \leq lmax_P - lmin_P$ (where $lmax_P = \left(\mathbf{MAX}\, p : p \in P : |p|\right)$). Since there is a longest keyword, we do not need to increase i past the mentioned maximum value. This leads to the following algorithm skeleton (where details (GS) and (EGC) still need to be instantiated):

Algorithm 2.8 (P_+, S_+, GS, EGC, LMIN)

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

do $l \neq \varepsilon$ **cand** $q \neq \bot \rightarrow$
  $l, v := l\downarrow 1, (l\uparrow 1)v;$
  $q := \delta_{R,\mathbf{f}}(q, l\uparrow 1);$
od;
$\{\ l = \varepsilon$ **cor** $(l\uparrow 1)v \notin \mathbf{f}(P')\ \}$
as $|v| = lmin_P \rightarrow$
  $w, s := v, r;$
  as $w \in P \rightarrow O := O \cup \{(l, w, s)\}$ **sa**;
  do $|w| \neq lmax_P \wedge s \neq \varepsilon \rightarrow$
    $w, s := w(s\uparrow 1), s\downarrow 1;$
    as $w \in P \rightarrow O := O \cup \{(l, w, s)\}$ **sa**
  od
sa
od$\{\ R\ \}$

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

It is possible to improve the efficiency of the last part of this algorithm by introducing a forward trie function (as in [35, 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** $|v| = lmin_P \rightarrow \ldots$ **sa** statement, or by the space and time spent when introducing the forward trie.
- Reduced maximal shift distances with detail (SSD), to be introduced in the next subsection: 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.
- 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 2.7 (P+, S+, GS, EGC), 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 on k is the distance to the next match, the maximal safe shift distance. Any smaller k is safe as well, and we thus define a safe shift distance as:

Definition 2.9 (*Safe Shift Distance*)**.** A shift distance k satisfying

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

is called a *safe shift distance*.

The use of assignment $u, r := u(r\uparrow k), r\downarrow k$ for a safe shift distance k forms algorithm detail (SSD). 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 $M(u, r)$ 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 2.7 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 2.10 (P+, S+, GS, EGC, 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: $ur = S$
$\quad \wedge O = \left(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\right)$
$\quad \wedge u = lv \wedge v \in \mathbf{f}(P) \wedge \left(l = \varepsilon \textbf{ cor } (l\uparrow 1)v \notin \mathbf{f}(P)\right)$ $\}$
do $r \neq \varepsilon \to$
$\quad u, r := u(r\uparrow k(l,v,r)), r\downarrow k(l,v,r);\ \ l, v := u, \varepsilon;\ \ q := \delta_{R,\mathbf{f}}(q_0, l\uparrow 1);$
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad \{$ invariant: $q = \delta^*_{R,\mathbf{f}}(q_0, ((l\uparrow 1)v)^R)$ $\}$
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \perp \to$
$\quad\quad l, v := l\downarrow 1, (l\uparrow 1)v;$
$\quad\quad q := \delta_{R,\mathbf{f}}(q, l\uparrow 1);$
$\quad\quad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
od$\{ R \}$

Using this algorithm skeleton, various sublinear algorithms may be obtained by choosing appropriate $\mathbf{f}(P)$ and shift function k. 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 [36], 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 consider using the set of suffixes of P, $\mathbf{suff}(P)$, for $\mathbf{f}(P)$ in Algorithm 2.7, i.e. instantiating **f** with **suff**.

As indicated in Section 2, direct evaluation of $(l\uparrow 1)v \in \mathbf{suff}(P)$ 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 2.6, update a state variable q to uphold invariant $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\uparrow 1)v)^R)$. The guard conjunct $(l\uparrow 1)v \in \mathbf{suff}(P)$ then becomes $q \neq \perp$.[1]

We can also introduce the safe shift distance, leading to the following instantiation of Algorithm 2.10:

Algorithm 3.1 (P+, S+, GS=S, EGC=RSA, 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: $ur = S$
$\quad \wedge O = \left(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\right)$
$\quad \wedge u = lv \wedge v \in \mathbf{suff}(P) \wedge \left(l = \varepsilon \textbf{ cor } (l\uparrow 1)v \notin \mathbf{suff}(P)\right)$ $\}$
do $r \neq \varepsilon \to$
$\quad u, r := u(r\uparrow k(l,v,r)), r\downarrow k(l,v,r);\ \ l, v := u, \varepsilon;\ \ q := \delta_{R,\mathbf{suff}}(q_0, l\uparrow 1);$
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad \{$ invariant: $q = \delta^*_{R,\mathbf{suff}}(q_0, ((l\uparrow 1)v)^R)$ $\}$
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \perp \to$
$\quad\quad l, v := l\downarrow 1, (l\uparrow 1)v;$
$\quad\quad q := \delta_{R,\mathbf{suff}}(q, l\uparrow 1);$
$\quad\quad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
od$\{ R \}$

Based on this algorithm skeleton, various shift functions can be obtained. This leads to the Commentz-Walter, Fu-San and multiple keyword Boyer–Moore and Boyer–Moore–Horspool algorithms. With the exception of multiple keyword Boyer–Moore–Horspool, which is described in [12], these were previously described in [38] already. We therefore discuss the latter algorithms only briefly here.

[1] These two algorithm details are equivalent to the introduction of algorithm detail (RT) in [38,35], 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.

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 give an approximation that does not depend on r. 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 Section 3.9. Taking the safe shift distance upperbound of Definition 2.9, [38] shows that

$$\begin{aligned} M(u,r) &= \big(\mathbf{MIN}\, n : 1 \le n \wedge \mathbf{suff}\big(u(r{\uparrow}n)\big) \cap P \neq \varnothing : n\big) \\ &\ge \big(\mathbf{MIN}\, n : 1 \le n \wedge \mathbf{suff}(uV^n) \cap P \neq \varnothing : n\big)\,. \end{aligned}$$

Further on in this paper, the latter predicate will be referred to as $N(u)$. It is our starting point for further derivations, as we 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 3.1). We will write $k(l,v)$ instead of $k(l,v,r)$.

3.2. Restriction to one symbol lookahead

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 [23,24]. It is obtained by weakening the predicate in the domain of the approximation of the upperbound in Section 3.1, $\mathbf{suff}(uV^n) \cap P \neq \varnothing$, to $V^*(l{\uparrow}1)vV^n \cap P \neq \varnothing \vee vV^n \cap V^*P \neq \varnothing$ [38, Section 3.2].

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 $l{\uparrow}1$ being either empty or consisting of one symbol. In the latter case we say to have one symbol lookahead. Observe that the symbol is 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

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

Function k_{opt} can be expressed as follows

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

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

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

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

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

(function d_2 in [14,15]), and $d_i \in \mathbf{suff}(P) \to \mathbb{N}$ is defined for $y \in \mathbf{suff}(P)$ by

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

(function d_1 in [14,15]). 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). This algorithm had previously been described by Fan and Su in [23,24]. 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. Shift functions presented in the following subsections give shifts smaller than k_{opt} that are expressed solely in functions needing one dimensional tables.

3.3. Lookahead symbol is mismatching

An approximation from below of d_{opt} yields an algorithm that is the common ancestor of multiple keyword Boyer–Moore [7] and the Commentz-Walter algorithm [14,15]. Essentially, the resulting shift function is not based on the identity of the lookahead symbol $l{\uparrow}1$ 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.

Assuming $l \neq \varepsilon$ and $(l{\uparrow}1)v \notin \mathbf{suff}(P)$, the range predicate $V^*(l{\uparrow}1)vV^n \cap P \neq \varnothing$ from d_{opt} can be weakened to

$$V^*(l{\uparrow}1)V^{|v|+n} \cap P \neq \varnothing \quad \wedge \quad V^*\big(V \setminus MS(v)\big)vV^n \cap P \neq \varnothing,$$

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

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

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 preceding range allows the derivation [38, Section 3.3] of a safe shift distance

$$char_{cw}(l{\uparrow}1, |v|) \mathbf{max}\, d_{vi}(v)$$

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

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

and $char_{cw} \in V \times \mathbb{N} \to \mathbb{N}$ is defined for all $a \in V, z \in N$ by

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

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

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

Approximation from below of k_{opt} by k_{bmcw} is referred to as algorithm detail (BMCW) and gives algorithm (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW).

3.4. The multiple keyword Boyer–Moore algorithm

The multiple keyword generalization of the Boyer–Moore algorithm [7] only differs from the algorithm of the preceding subsection in the way the lookahead symbol is taken into account. Assuming $l \neq \varepsilon$, we can show that

$$\begin{array}{ll} & char_{cw}(l{\uparrow}1, |v|) \\ \geq & \quad \{\ \text{derivation in [38, Section 3.4]}\ \} \\ & char_{bm}(l{\uparrow}1) - |v| \end{array}$$

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

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

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

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

Approximating k_{bmcw} from below by k_{bm} is referred to as algorithm detail (BM). The resulting algorithm is characterized by (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)).

The formula for the Boyer–Moore shift function given here differs from but is equivalent to the ones given in [7,1], as is shown in [38].

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, which is allowed since $d_{vi}(v) \geq d_i(v)$ for all v [38, Section 3.5].

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

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

Approximating k_{bmcw} from below by k_{cw} is referred to as algorithm detail (CW). It results in the Commentz-Walter algorithm [14,15], characterized by (P_+, S_+, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, CW).

3.6. Complete decoupling of recognized suffix and lookahead symbol

The derivations in the previous two 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 for $y \in \mathbf{suff}(P)$ by

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

The algorithm can be characterized by 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}. Approximating k_{opt} from below by k_{nla} is referred to as algorithm detail (NLA) (NO LOOKAHEAD at mismatching symbol) and gives 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 range predicate $\mathbf{suff}(u(r{\uparrow}n)) \cap P \neq \varnothing$ of Definition 2.9:

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

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

$$V^*(l{\uparrow}1) v V^n \cap V^* P \neq \varnothing$$

(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:

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

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:

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

To use $char_{bm}(l\uparrow 1)$ as a safe shift distance, we need $l \neq \varepsilon$ to hold in case $v = \varepsilon$. Note that in Algorithm 3.1 (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

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

The use of shift function k_{bmh} yields algorithm (P$_+$, S$_+$, GS=S, EGC=RSA, SSD, NLAU, OPT, BMCW, BMH), the Set Horspool algorithm [31, Section 3.3.2]. (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 $v \neq \varepsilon$.) Adding problem detail (OKW) leads to the single-keyword Horspool algorithm [31, Section 2.3.2], [25].

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 2.9 (which is $M(u, r)$) as a starting point, we can show that

$$\begin{array}{ll} & M(u, r) \\ \geq & \quad \{\text{ definition } char_{la} \text{ (below), derivation in [38, Section 3.8] }\} \\ & char_{la}(r\uparrow 1) + 1 \end{array}$$

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

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

Let $M(u, r)$ denote the expression for the upperbound on k, and let $N(u)$ denote the last expression in the derivation in Section 3.1. We then have

$$M(u, r) = M(u, r)\ \mathbf{max}\ M(u, r) \geq N(u)\ \mathbf{max}\big(char_{la}(r\uparrow 1) + 1\big).$$

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 $\mathbf{max}\big(char_{la}(r\uparrow 1)+1\big)$ 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)).

4. Factor-based sublinear pattern matching

We now derive a family of algorithms by using detail choice (GS=F) (Guard Strengthening = Factor), i.e. instantiating **f** with **fact** in Algorithm 2.7.

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 $\delta_{R,\mathbf{fact}}$ of 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 $l \neq \varepsilon$ **cand** $q \neq \bot$.

Note that various automata exist whose transition functions can be used for $\delta_{R,\mathbf{fact}}$, including the trie built on $\mathbf{fact}(P)^R$ and the *suffix automaton* or the *dawg* (for directed acyclic word graph) on $\mathbf{fact}(P)^R$ [18].

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 Section 4.1.

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

Algorithm 4.1 (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: $ur = S$
$\quad \wedge O = \big(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\big)$
$\quad \wedge u = lv \wedge v \in \mathbf{fact}(P) \wedge \big(l = \varepsilon \ \mathbf{cor}\ (l\uparrow 1)v \notin \mathbf{fact}(P)\big)$ }
do $r \neq \varepsilon \to$
$\quad u, r := u(r\uparrow k(l, v, r)), r\downarrow k(l, v, r)$; $l, v := u, \varepsilon$; $q := \delta_{R,\mathbf{fact}}(q_0, l\uparrow 1)$;
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad$ { invariant: $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\uparrow 1)v)^R)$ }
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \perp \to$
$\quad\quad l, v := l\downarrow 1, (l\uparrow 1)v$;
$\quad\quad q := \delta_{R,\mathbf{fact}}(q, l\uparrow 1)$;
$\quad\quad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
od{ R }

Since $(l\uparrow 1)v \notin \mathbf{fact}(P) \Rightarrow (l\uparrow 1)v \notin \mathbf{suff}(P)$, we may use *any* safe shift function derived for suffix-based sublinear algorithms, discussed in Section 3. This results in a large number of new algorithms, since most such shift functions have not been used with a factor-based algorithm before. In using such a shift function, we replace the $\mathbf{suff}(P)$ part of their domain by $\mathbf{fact}(P)$, meaning that precomputation changes.

4.1. The no-factor shift

We can do better than simply using the shift functions from Section 3, since $(l\uparrow 1)v \notin \mathbf{fact}(P)$ is stronger than $(l\uparrow 1)v \notin \mathbf{suff}(P)$. In [12] we show that predicate $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$ from the domain of the approximation of the upperbound on k given in Section 3.1 may be rewritten to $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P$.

Observe that the left conjunct – $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$ – is the original predicate, used to derive safe shift distances for suffix-based sublinear algorithms in Section 3.

We derive

$$\begin{array}{ll} & \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing \wedge |v| + n \geq lmin_P : n\big) \\ \geq & \quad \{\ \mathbf{MIN} \text{ with conjunctive range}\ \} \\ & \big(\mathbf{MIN}\, n : 1 \leq n \wedge \mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing : n\big) \\ & \quad \mathbf{max}\ \big(\mathbf{MIN}\, n : 1 \leq n \wedge |v| + n \geq lmin_P : n\big) \\ \geq & \quad \{\ \} \\ & M(u, r)\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|)) \end{array}$$

We may thus use any shift function that is a weakening of

$$M(u, r)\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|)).$$

The left operand of the outer **max** corresponds to any suffix-based safe shift function, while the right operand corresponds to the shift in case $(l\uparrow 1)v$ 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 suffix-based safe shift function k_{ssd} (algorithm detail (NFS)). We call it the no-factor shift, since it uses $(l\uparrow 1)v \notin \mathbf{fact}(P)$. In particular, we may use safe shift distance 1 with the no-factor shift to get shift distance $1\ \mathbf{max}\ (1\ \mathbf{max}\ (lmin_P - |v|)) = 1\ \mathbf{max}\ (lmin_P - |v|)$.

This equals the shift distance used in the basic ideas for backward DAWG matching [16], [31, page 27] and – combined with algorithm detail (LMIN) mentioned in Section 2 – set backward DAWG matching [31, page 68]. The actual algorithms described in the literature use an improvement based on a property of DAWGs. We discuss this in Section 4.2.

The shift function $1\ \mathbf{max}\ (lmin_P - |v|)$ just requires precomputation of $lmin_P$, yet gives quite large shift distances. This is the reason why factor-based sublinear keyword pattern matching algorithms have gotten a lot of attention since their first descriptions in literature. The algorithms in literature do not combine it with any of the more involved precomputed shift functions as those described in [9,35,38]. If precomputation time is not an important issue however, combining such a shift function and the no-factor shift may be advantageous, potentially yielding larger shifts. As far as we know, such a combination has not been described or used before. Since about ten shift functions are given in [9], the combination of a single one with the no-factor shift already gives us about ten new factor-based sublinear keyword pattern matching algorithms. It remains to be investigated whether these algorithms indeed improve over the running time of the algorithms from literature.

4.2. Cheap computation of a particular shift function

We now consider a different weakening of $\mathbf{suff}(u(r\uparrow n)) \cap P \neq \varnothing$, namely the one to $\mathbf{suff}((v\uparrow last_{v,P})V^n) \cap P \neq \varnothing$, with

$$last_{v,P} = \big(\mathbf{MAX}\, m : 0 \leq m \leq |v| \wedge v\uparrow m \in \mathbf{pref}(P) : m\big)\,.$$

This weakening is derived in [12], which also shows the resulting shift distance to be at least as big as $1\ \mathbf{max}\ (lmin_P - last_{v,P})$. The latter may thus be used as a safe shift distance as well. The value of $last_{v,P}$ 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 $last_{v,P}$ '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, $last_{v,P}$ 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

$$k_{lskp} = 1\ \mathbf{max}\ (lmin_P - last_{v,P}).$$

This shift function does not depend on l and can therefore be seen as a variant of algorithm detail (NLA) discussed in Section 3.7. (The shift function does not directly depend on v either, but it indirectly depends on v due to its dependence on $last_{v,P}$.) Calculating the shift distance using k_{lskp} is algorithm detail (LSKP) (longest suffix which is keyword prefix). Using variable $last_{v,P}$ and shift function k_{lskp}, the algorithm becomes:

Algorithm 4.2 (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**;
$l, v := \varepsilon, \varepsilon$;
$last_{v,P} := 0$;
{ invariant: $ur = S$
$\quad \wedge\ O = \big(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\big)$
$\quad \wedge\ u = lv \wedge v \in \mathbf{fact}(P) \wedge \big(l = \varepsilon\ \mathbf{cor}\ (l\uparrow 1)v \notin \mathbf{fact}(P)\big)$ }
do $r \neq \varepsilon \to$
$\quad k := 1\ \mathbf{max}(lmin_P - last_{v,P})$;
$\quad u, r := u(r\uparrow k), r\downarrow k;\ \ l, v := u, \varepsilon$;
$\quad q, last_{v,P} := \delta_{R,\mathbf{fact}}(q_0, l\uparrow 1), 0$;
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad$ { invariant: $q = \delta^*_{R,\mathbf{fact}}(q_0, ((l\uparrow 1)v)^R)$
$\qquad\qquad \wedge\ last_{v,P} = \big(\mathbf{MAX}\, m : m \leq |v| \wedge v\uparrow m \in \mathbf{pref}(P) : m\big)$ }
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \bot \to$
$\qquad l, v := l\downarrow 1, (l\uparrow 1)v$;
$\qquad q := \delta_{R,\mathbf{fact}}(q, l\uparrow 1)$
$\qquad$ **as** $q \in F \to last_{v,P} := |v|$ **sa**;
$\qquad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
od{ R }

This algorithm is a variant of the Set Backward DAWG Matching [16], [31, page 68] algorithm, which adds algorithm detail (LMIN). Adding algorithm detail (OKW) to the resulting algorithm results in (single-keyword) Backward DAWG Matching.

Algorithm detail (NFS) is included in neither detail sequence, since the no-factor shift can never be larger than the k_{lskp} shift: $lmin_P - |v| \leq lmin_P - last_{v,P} = k_{lskp}$.

5. Factor-oracle-based sublinear pattern matching

We now derive a family of algorithms by using $\mathbf{factoracle}(P^R)^R$ for $\mathbf{f}(P)$. We may do so since $\mathbf{factoracle}(P^R)^R \supseteq \mathbf{fact}(P^R)^R$ and **factoracle** is suffix-closed [3,4,11]. We strengthen the inner repetition guard, which now becomes $l \neq \varepsilon$ **cand** $(l\uparrow 1)v \in \mathbf{factoracle}(P^R)^R$.

In [28], a characterization of **factoracle**, the language of a *factor oracle*, is given. Direct evaluation of $(l\uparrow 1)v \in \mathbf{factoracle}(P^R)^R$ is inefficient however, so the transition function of the factor oracle [3,4,11] recognizing the set $\mathbf{factoracle}(P^R)$ is used. Using function $\delta_{\mathbf{factoracle}(P^R)}$[2] and making $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l\uparrow 1)v)^R)$ an invariant of the inner repetition (algorithm detail (EGC=RFO), the guard becomes $l \neq \varepsilon$ **cand** $q \neq \bot$.

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

[2] Since $\mathbf{factoracle}(P)^R \neq \mathbf{factoracle}(P^R)$ could hold, we cannot use $\delta_{R,\mathbf{factoracle}}$ to describe the transition function. We introduce $\delta_{\mathbf{factoracle}(P^R)}$, the transition function of the automaton recognizing $\mathbf{factoracle}(P^R)$.

- Easier construction of (possibly more compact) automata: The factor oracle recognizing the words in $\mathbf{factoracle}(P^R)$ is easier to construct and may be smaller than an automaton recognizing $\mathbf{fact}(P^R)$ (see [3,4,10,11]).
- More comparisons: When $(l{\uparrow}1)v \notin \mathbf{fact}(P)$ yet $(l{\uparrow}1)v \in \mathbf{factoracle}(P^R)^R$, 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.

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) to (GS=S, EGC=RSA).

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

Algorithm 5.1 (P_+, S_+, GS=FO, EGC=RFO, 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: $ur = S$
$\quad \wedge O = \big(\bigcup x, y, z : xyz = S \wedge xy \leq_p u \wedge y \in P : \{(x, y, z)\}\big)$
$\quad \wedge u = lv \wedge v \in \mathbf{factoracle}(P^R)^R$
$\quad\quad \wedge \big(l = \varepsilon$ **cor** $(l{\uparrow}1)v \notin \mathbf{factoracle}(P^R)^R\big)$ }
do $r \neq \varepsilon \to$
$\quad u, r := u(r{\uparrow}k(l, v, r)), r{\downarrow}k(l, v, r);\ \ l, v := u, \varepsilon;\ \ q := \delta_{\mathbf{factoracle}(P^R)}(q_0, l{\uparrow}1);$
$\quad$ **as** $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**;
$\quad$ { invariant: $q = \delta^*_{\mathbf{factoracle}(P^R)}(q_0, ((l{\uparrow}1)v)^R)$ }
$\quad$ **do** $l \neq \varepsilon$ **cand** $q \neq \perp \to$
$\quad\quad l, v := l{\downarrow}1, (l{\uparrow}1)v;$
$\quad\quad q := \delta_{\mathbf{factoracle}(P^R)}(q, l{\uparrow}1);$
$\quad\quad$ **as** $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
$\quad$ **od**
od{ R }

Since $\mathbf{factoracle}(P^R)^R \supseteq \mathbf{fact}(P^R)^R = \mathbf{fact}(P)$, we have

$$(l{\uparrow}1)v \notin \mathbf{factoracle}(P^R)^R \Rightarrow (l{\uparrow}1)v \notin \mathbf{fact}(P).$$

Any shift function may therefore be used satisfying

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

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 [4], [31, pages 69–72] equals our algorithm (P_+, S_+, GS=FO, EGC=RFO, LMIN, SSD, NFS, ONE), while the single keyword Backward Oracle Matching algorithm [3], [31, pages 34–36], [10,11] corresponds to (P_+, S_+, GS=FO, EGC=RFO, SSD, NFS, ONE, OKW).

6. Final remarks

We presented a revised and expanded version [12] of the original taxonomy of sublinear keyword pattern matching algorithms [38,35], 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 could also be described using a generalization of the alternative Commentz-Walter algorithm skeleton presented in [36], 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 accommodate 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 [38], 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 includes the addition of such algorithms, including the (Multi-)BNDM algorithm [30], the algorithm by Crochemore et al. presented in [17], the Wu–Manber algorithm [39], and Sunday's variant of Boyer–Moore–Horspool [33]. In addition, the possibility of a bit-parallel suffix-based pattern matching algorithm, i.e. a bit-parallel Commentz-Walter, could be considered.

Appendix. Algorithm and problem details

In this appendix we list algorithm and problem details with their 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).

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in: Handbook of Theoretical Computer Science, vol. A, Elsevier, Amsterdam, 1990, pp. 255–300.
[2] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (1975) 333–340.
[3] C. Allauzen, M. Crochemore, M. Raffinot, Efficient experimental string matching by weak factor recognition, in: Proceedings of the 12th Conference on Combinatorial Pattern Matching, in: LNCS, vol. 2089, 2001.
[4] C. Allauzen, M. Raffinot, Oracle des facteurs d'un ensemble de mots, Tech. Rep. 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
[5] A. Apostolico, Z. Galil, Pattern Matching Algorithms, Oxford University Press, 1997.
[6] G. Barla-Szabo, A taxonomy of graph representations, Master's Thesis, Department of Computer Science, University of Pretoria, November 2002.
[7] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (10) (1977) 62–72.
[8] L. Cleophas, B.W. Watson, G. Zwaan, Automaton-based sublinear keyword pattern matching, in: Proceedings of the 11th International Conference on String Processing and Information Retrieval, SPIRE 2004, in: LNCS, vol. 3246, Springer, 2004.
[9] L. Cleophas, B.W. Watson, G. Zwaan, A new taxonomy of sublinear keyword pattern matching algorithms, Tech. Rep. 04/07, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, March 2004.
[10] L. Cleophas, G. Zwaan, B.W. Watson, Constructing factor oracles, Tech. Rep. 04/01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, January 2004.
[11] L. Cleophas, G. Zwaan, B.W. Watson, Constructing factor oracles, Journal of Automata, Languages and Combinatorics 10 (5/6) (2005) 627–640.
[12] L.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.
[13] L.G.W.A. Cleophas, Tree algorithms: two taxonomies and a toolkit, Ph.D. Thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, April 2008.
[14] B. Commentz-Walter, A string matching algorithm fast on the average, in: H.A. Maurer (Ed.), Proceedings of the 6th International Colloquium on Automata, Languages and Programming, Springer, Berlin, 1979.
[15] B. Commentz-Walter, A string matching algorithm fast on the average, Tech. Rep. TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.
[16] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string matching algorithms, Algorithmica 12 (4/5) (1994) 247–267.
[17] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, W. Rytter, Fast practical multi-pattern matching, Information Processing Letters 71 (3–4) (1999) 107–113.
[18] M. Crochemore, C. Hancart, Automata for Matching Patterns, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 2, Springer, 1997.
[19] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[20] M. Crochemore, W. Rytter, Jewels of Stringology — Text Algorithms, World Scientific Publishing, 2003.
[21] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, NJ, 1976.
[22] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer, New York, NY, 1990.
[23] J.-J. Fan, K.-Y. Su, An efficient algorithm for matching multiple patterns, IEEE Transactions Knowledge and Data Engineering 5 (1993) 339–351.
[24] J.-J. Fan, K.-Y. Su, An efficient algorithm for matching multiple patterns, in: J. Aoe (Ed.), Computer Algorithms: String Pattern Matching Strategies, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 91–104.
[25] R.N. Horspool, Practical fast searching in strings, Software—Practice & Experience 10 (6) (1980) 501–506.
[26] H.B.M. Jonkers, Abstraction, specification and implementation techniques, with an application to garbage collection, Tech. Rep. 166, Mathematisch Centrum, Amsterdam, 1983.
[27] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM Journal of Computing 6 (2) (1977) 323–350.
[28] A. Mancheron, C. Moan, Combinatorial characterization of the language recognized by factor and suffix oracles, in: Proceedings of the Prague Stringology Conference 2004, Department of Computer Science and Engineering, Czech Technical University, Prague, 2004.
[29] F.C. Mish (Ed.), Merriam Webster's Collegiate Dictionary, 10th ed., Merriam Webster, Springfield, MA, 1993.
[30] G. Navarro, M. 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.
[31] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings: Practical on-line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2002.
[32] W. Smyth, Computing Patterns in Strings, Addison-Wesley, 2003.
[33] D.M. Sunday, A very fast substring search algorithm, Communications of the ACM 33 (8) (1990) 132–142.
[34] J.P.H.W. van den Eijnde, Program derivation in acyclic graphs and related problems, Tech. Rep. 92/04, Faculty of Computing Science, Technische Universiteit Eindhoven, 1992.
[35] B.W. Watson, Taxonomies and toolkits of regular language algorithms, Ph.D. Thesis, Faculty of Computing Science, Technische Universiteit Eindhoven, September 1995.
[36] B.W. Watson, A new family of Commentz-Walter-style multiple-keyword pattern matching algorithms, in: Proceedings of the Prague Stringology Club Workshop 2000, Department of Computer Science and Engineering, Czech Technical University, Prague, 2000.
[37] B.W. Watson, L. Cleophas, SPARE parts: a C++ toolkit for string pattern recognition, Software—Practice & Experience 34 (7) (2004) 697–710.
[38] B.W. Watson, G. Zwaan, A taxonomy of sublinear multiple keyword pattern matching algorithms, Science of Computer Programming 27 (2) (1996) 85–118.
[39] S. Wu, U. Manber, A fast algorithm for multi-pattern searching, Tech. Rep. TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.