A Collection of New Regular Grammar Pattern Matching Algorithms

Bruce W. Watson

Ribbit Software Systems Inc.
IST Technologies Research Group
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

Abstract. A number of new algorithms for regular grammar pattern matching is presented. The new algorithms handle patterns specified by regular grammars — a generalization of multiple keyword pattern matching and single keyword pattern matching, both considered extensively in and [14, Chapter 4] and in [18].

Among the algorithms is a Boyer-Moore type algorithm for regular grammar pattern matching, answering a variant of an open problem posed by A.V. Aho in 1980 [2, p. 342]. Like the Boyer-Moore and Commentz-Walter algorithms, the generalized algorithm makes use of shift functions which can be precomputed and tabulated.

It appears that many of the new algorithms can be efficiently implemented.

Key words: pattern matching, algorithms, regular grammars, regular pattern matching, algorithmics, string algorithms

1 Introduction

The pattern matching problem is: given a regular pattern grammar (for a formal definition see Section 2) and an input string S (over an alphabet V), find all substrings of S which correspond to the language denoted by some production in the grammar. Several restricted forms of this problem have been solved (all of which are discussed in detail in [14, Chapter 4], and in [3, 18]):

- The Knuth-Morris-Pratt [12] and Boyer-Moore [5] algorithms solve the problem when there is only a single production and its right-hand side has no nonterminals — it is in V^* (the single keyword pattern matching problem).
- The Aho-Corasick [1] and Commentz-Walter [6, 7] algorithms solve the problem when all productions in the grammar have right-hand sides without nonterminals — all of them are in V^* (this is the multiple keyword pattern matching problem). The Aho-Corasick and Commentz-Walter algorithms are generalizations of the Knuth-Morris-Pratt and Boyer-Moore algorithms respectively.

To date, very few regular grammar pattern matching algorithms have been developed. Only recently, the generalized Boyer-Moore algorithm was developed [14, 16, 17]*.

It should be noted that there do exist other regular pattern matching algorithms with good performance — for example, the one which was developed by R. Baeza-Yates [4, 10]. Those algorithms are not, however, considered here since they either use regular expressions† or they require some precomputation on the input string, and are therefore not suited to the type of application presented in this paper.

This paper is structured as follows:

- Section 2 gives the problem specification, and a naïve algorithm.
- Section 3 gives a family of algorithms which process the input string in a left-to-right manner.
- Section 4 gives a family of algorithms which process the input string in a right-to-left manner. These algorithms are not symmetrical with the ones in Section 3, due to our asymmetrical choice of right-linear grammars for our regular pattern grammars.
- Section 5 presents the conclusions of this paper.

Before continuing with the developement of the algorithms, we first give some of the definitions required for reading this paper.

1.1 Mathematical preliminaries

Most of the following definitions are standard ones relating to regular grammars and languages.

Definition 1.1 (Alphabet): An *alphabet* is a finite, non-empty set of symbols. □

Throughout this paper, we will assume some fixed alphabet V.

Definition 1.2 (Functions pref and suff): For a given string z, define $\mathbf{pref}(z)$ (respectively $\mathbf{suff}(z)$) to be the set of prefixes (respectively suffixes), including string z and the empty string ε, of z. □

Definition 1.3 (String manipulation operators): Since we will be manipulating the individual symbols of strings, and we do not wish to resort to such low-level details as indexing, we define the following four operators (all of which are infix operators, taking a string as the left operand, a natural number as the right operand, and yielding a string):

- $w \nwarrow k$ is the $k\,\mathbf{min}\,|w|$ left-most symbols of w.
- $w \nearrow k$ is the $k\,\mathbf{min}\,|w|$ right-most symbols of w.

*That research was performed jointly with Richard E. Watson of the Department of Mathematics, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada; he can now be reached at `rwatson@RibbitSoft.com`.

†Although regular grammars and regular expressions have the same descriptive power, they can yield algorithms which are substantially different.

- $w \swarrow k$ is the $|w| - k \,\mathbf{max}\, 0$ right-most symbols of w.
- $w \searrow k$ is the $|w| - k \,\mathbf{max}\, 0$ left-most symbols of w.

□

Definition 1.4 (Regular pattern grammar): A *(right) regular grammar* (also known as a *right linear grammar*) is defined to be a three tuple, (V, N, P), where:

- V is our alphabet, known as the *terminal* alphabet.
- N is an alphabet, known as the *nonterminal* alphabet.
- $P \subseteq N \times (V^* \cup V^*N)$ is a finite and nonempty set of (right linear) *productions*. We usually write a given production (A, w) as $A \longrightarrow w$. We also define left-hand side and right-hand side functions **lhs** and **rhs** (respectively) such that $\mathbf{lhs}(A \longrightarrow w) = A$ and $\mathbf{rhs}(A \longrightarrow w) = w$.

  We also define a function **vpart** on right-hand sides as follows: $\mathbf{vpart}(w)$ is the longest prefix of w containing only symbols in V; that is, we drop the nonterminal on the right, if there is one. More formally, for a right-hand side x (for $x \in V^*$) we have $\mathbf{vpart}(x) = x$; for a right-hand side xB (for $B \in N$, $x \in V^*$) we have $\mathbf{vpart}(xB) = x$.

□

We assume some fixed regular grammar (V, N, P) throughout this paper. Note that, unlike usual grammars (for parsing, etc.), we do not have a "start symbol". We have chosen to use right regular grammars, instead of left regular grammars (which have the same descriptive power), because they are symmetrical (under reversal); with of this choice, we must treat both left-to-right and right-to-left algorithms.

Definition 1.5 (Languages of strings and productions): We define function $\mathcal{L}$, mapping strings in $V^* \cup V^*N$ to regular languages over V, as follows (for $w \in V^*$):

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

and (for $B \in N$)

$$\mathcal{L}(wB) = \{w\}\mathcal{L}(B)$$

We extend function $\mathcal{L}$ to map productions and nonterminals to the regular languages they denote as follows (for $A \longrightarrow w \in P$):

$$\mathcal{L}(A \longrightarrow w) = \mathcal{L}(w)$$

and

$$\mathcal{L}(A) = (\cup\, w : p \in P \wedge A \in \mathbf{lhs}(p) : \mathcal{L}(p))$$

Since this definition may be recursive, we naturally take the usual fixed-point definition, which always yields regular languages. □

Property 1.6 (Language of a production): We have the following useful property of the language of a production $p \in P$:

$$\mathcal{L}(p) \subseteq \mathbf{vpart}(\mathbf{rhs}(p))V^*$$

□

Intuitively, the above property holds because all words in the language denoted by some production p have $\mathbf{vpart}(\mathbf{rhs}(p))$ as their prefix.

Throughout this paper, we adopt the convention of extending a given function which takes elements of some set D so that it takes elements of 2^D (sets of elements taken from D). (Typically, this is used to extend a function which takes one production, giving a function which takes a set of productions.)

Definition 1.7 (Chain rules): Productions of the form $A \longrightarrow B$ (for $A, B \in N$) are known as *chain rules*. (When B has been recognized as the left-hand side of a production matching a substring, production $A \longrightarrow B$ has been matched as well.) For this reason, we define function $\mathbf{crule} \in 2^P \longrightarrow 2^P$ (where 2^P denotes the set of all subsets of our production set P) as:

$$\mathbf{crule}(U) = \{\, A \longrightarrow B \mid A \longrightarrow B \in P \wedge B \in \mathbf{lhs}(U) \,\}$$

We define function $\mathbf{crule}^*$ to be the reflexive and transitive closure of function **crule**.

□

2 Problem specification and a naïve algorithm

We begin this section with a precise specification of the regular grammar pattern matching problem.

Definition 2.1 (Regular pattern matching problem): Given an input string $S \in V^*$, and our regular grammar, establish postcondition RPM :

$$O = \{\, (l, p) \mid lu = S \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$$

□

Intuitively, this means that we are registering all productions which match some substring of S, along with the left context (in S) of the match location. (That is, for simplicity, we are registering our matches by their begin-point.) We will use the notation O_x to refer to the set of productions in O which match with left context x (a prefix of S). More formally, $O_x = \{\, p \mid (x, p) \in O \,\}$.

We can now give our naïve (and nondeterministic) algorithm

Algorithm 2.2:

$O := \emptyset$;
for $l, u : lu = S \rightarrow$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
rof{ RPM }

□

Note that we are still making some assumptions about our ability to evaluate membership in $\mathcal{L}(p)$.

In the next two sections, we consider adding more determinism to our first algorithm. We will use the property that substrings of S can be characterized as "prefixes of suffixes" of S or as "suffixes of prefixes" of S.

3 Left-to-right algorithms

We begin by deciding to traverse input string S from left-to-right in the following algorithm.

Algorithm 3.1:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
od$\{\ RPM\ \}$

□

This algorithm is still far from practical to implement, and we now consider adding an inner repetition to implement the update of O. The inner repetition can consider prefixes of u in either increasing order or decreasing order. We begin by considering the former.

Algorithm 3.2:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{lv\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

The test $v \in \mathcal{L}(p)$ is particularly difficult to implement — indeed, almost as difficult as the problem we are trying to solve. There does not appear to be an easy manner in which to improve the algorithm, and we abandon its development here.

The following (alternative) algorithm considers prefixes of u in order of decreasing length.

Algorithm 3.3:

$l, u := \varepsilon, S;\ O := \{\varepsilon\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(S) \cap \mathcal{L}(p) \neq \emptyset \,\};$
do $u \neq \varepsilon \rightarrow$
  $l, u := l(u \nwarrow 1), u \swarrow 1;$
  $v, r := u, \varepsilon;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge u \in \mathcal{L}(p) \,\};$
  do $v \neq \varepsilon \rightarrow$
    $v, r := v \searrow 1, (v \nearrow 1)r;$
    $O := O \cup \{lv\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od{ *RPM* }

□

Improving this algorithm appears to be as difficult as the previous one. As a result, we do not pursue either of them any further.

3.1 A recursive algorithm

We can also develop a recursive algorithm which traverses input string S from left-to-right. Before presenting the recursive version, we first examine another imperative algorithm traversing S from left-to-right. It considers productions p and decompositions $u, r : ur = S$ such that $\mathbf{vpart}(p) \in \mathbf{suff}(u)$. For this algorithm only, we make two trivial restrictions to patterns and the input string: $S \neq \varepsilon$ and there is no production p such that $\mathbf{rhs}(p) = \varepsilon$. We also define the following predicate, which makes the subsequent algorithms more concise:

Definition 3.4 (Predicate R): Predicate R takes an argument in $N \times V^* \times (N \cup V) \times V^*$:

$$R(A, w, a, u) \equiv A \longrightarrow wa \in P \wedge w \in \mathbf{suff}(u)$$

□

Algorithm 3.5:

$u, r := \varepsilon, S;\ O := \emptyset;$
do $r \neq \varepsilon \rightarrow$
  { $ur = S$ }
  { Deal with productions without a nonterminal in the RHS. }
  $O := O \cup \{\, (u \searrow |w|, A \longrightarrow w(r \nwarrow 1)) \mid R(A, w, r \nwarrow 1, u) \,\};$
  { Deal with productions with a nonterminal in the RHS. }
  $O := O \cup \{\, (u \searrow |w|, A \longrightarrow wB) \mid R(A, w, B, u) \wedge B \in N \wedge$
    $\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset \,\};$
  $u, r := u(r \nwarrow 1), r \swarrow 1;$
od{ *RPM* }

□

We now present the recursive version of the same algorithm. Beginning with $O = \emptyset$, the procedure invocation $mat(\varepsilon, S)$ will yield postcondition RPM.

Algorithm 3.6:

proc $mat(u, r) \rightarrow$
  $\{\ ur = S \wedge r \neq \varepsilon\ \}$
  { Deal with productions without a nonterminal in the RHS. }
  $O := O \cup \{\ (u \searrow |w|, A \longrightarrow w(r \nwarrow 1)) \mid R(A, w, r \nwarrow 1, u)\ \}$;
  { Deal with productions with a nonterminal in the RHS. }
  $O := O \cup \{\ (u \searrow |w|, A \longrightarrow wB) \mid R(A, w, B, u) \wedge B \in N \wedge$
    $\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset\ \}$;
  if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1)$
  $[\!]$ $(r \swarrow 1) = \varepsilon \rightarrow$ **skip**
  fi
corp

□

This algorithm is guaranteed to terminate, since S is a finite string, and the division between u and r (in S) is monotonically moving from left-to-right with each recursive call.

The algorithm can be made more efficient (in the next few pages) by moving the second update of O below the recursive call.

The expression $\mathbf{suff}(u)$ occurs in several places within the update of O. We could introduce a new parameter such that $U : U = \mathbf{suff}(u)$. In the above algorithm, all tests for membership in $\mathbf{suff}(u)$ involve prefixes of $\mathbf{rhs}(P)$. For this reason, we can introduce the more general $U : \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))$. This is established with $U = \{\varepsilon\}$ in the initial invocation of mat. We now derive the new value for U in the recursive invocation of mat, based upon the old value:

$$
\begin{aligned}
& \mathbf{suff}(u(r \nwarrow 1)) \cap \mathbf{pref}(\mathbf{rhs}(P)) \\
= & \quad \{\text{ property of } \mathbf{suff}\ \} \\
& (\mathbf{suff}(u)(r \nwarrow 1) \cup \{\varepsilon\}) \cap \mathbf{pref}(\mathbf{rhs}(P)) \\
= & \quad \{\cup \text{ distributes over } \cap;\ \varepsilon \in \mathbf{pref}(\mathbf{rhs}(P))\ \} \\
& (\mathbf{suff}(u)(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\} \\
= & \quad \{\ \{wa\} \cap \mathbf{pref}(W) \neq \emptyset \equiv (\{w\} \cap \mathbf{pref}(W))\{a\} \cap \mathbf{pref}(W) \neq \emptyset\ \} \\
& ((\mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P)))(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\} \\
= & \quad \{\text{ definition of } U\ \} \\
& (U(r \nwarrow 1) \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\varepsilon\}
\end{aligned}
$$

The domain of U is finite, and it conceptually represents a *state*. There are some very efficient means of representing this particular domain, which is related to the Aho-Corasick state function. The new value of U can be more easily computed using the following function.

Definition 3.7 (Function τ): We define function $\tau \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \times (N \cup V) \longrightarrow 2^{\mathbf{pref}(\mathbf{rhs}(P))}$ as

$$\tau(U, a) = (Ua \cap \mathbf{pref}(\mathbf{rhs}(P))) \cup \{\emptyset\}$$

□

This function is easily precomputed, since it corresponds to the forward trie constructed from the right-hand sides of productions. The updates of O can be made much simpler with the introduction of the following function (most of which can also be precomputed):

Definition 3.8 (Function *Output*): Function $Output \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \times V^* \longrightarrow V^* \times 2^{\mathbf{pref}(\mathbf{rhs}(P))}$ is defined as

$$Output(U, u) = (\cup\, A, w : A \longrightarrow w \in P \wedge w \in U : \{u \searrow |w|\} \times \mathbf{crule}^*(A \longrightarrow w))$$

□

These two functions are used in the following version of our algorithm:

Algorithm 3.9:

---

proc $mat(u, r, U) \rightarrow$
  $\{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
  $O := O \cup Output(\tau(U, r \nwarrow 1), u);$
  if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
  $[\!]\ (r \swarrow 1) = \varepsilon \rightarrow$ **skip**
  fi;
  $O := O \cup (\cup\, B : B \in N \wedge \mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset : Output(\tau(U, B), u))$
corp

□

The second update of O still contains the predicate $\mathbf{pref}(r) \cap \mathcal{L}(B) \neq \emptyset$. With this update appearing after the recursive call to *mat*, all productions matching at u will already appear in O. The predicate can therefore be replaced with $B \in \mathbf{lhs}(O_u)$ in the following algorithm:

Algorithm 3.10:

---

proc $mat(u, r, U) \rightarrow$
  $\{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
  $O := O \cup Output(\tau(U, r \nwarrow 1), u);$
  if $(r \swarrow 1) \neq \varepsilon \rightarrow mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
  $[\!]\ (r \swarrow 1) = \varepsilon \rightarrow$ **skip**
  fi;
  $O := O \cup (\cup\, B : B \in N \wedge B \in \mathbf{lhs}(O_u) : Output(\tau(U, B), u))$
corp

□

For our final improvement, we change the algorithm such that at any given $u, r : ur = S$, we only register local matches (those p matching at u). Any other (nonlocal) matches are returned by the procedure to be locally registered at their appropriate local u level. To make the new version concise, we redefine our output function as follows:

Definition 3.11 (Function *Output*): Function $Output \in 2^{\mathbf{pref}(\mathbf{rhs}(P))} \longrightarrow Naturals \times 2^P$ now registers matching productions with the number of levels that they must be passed back for matching

$$Output(U) = (\cup\ A, w : A \longrightarrow w \in P \wedge w \in U : \{|w|\} \times \mathbf{crule}^*(A \longrightarrow w))$$

□

Since this function does not depend upon u (unlike our earlier definition), we can fully precompute it. The new matching scheme is presented in the final recursive algorithm (in which we introduce variable *matches* to hold the intermediate matches):

Algorithm 3.12:

---

proc $mat(u, r, U) \rightarrow$
$\{\ ur = S \wedge r \neq \varepsilon \wedge U = \mathbf{suff}(u) \cap \mathbf{pref}(\mathbf{rhs}(P))\ \}$
$matches := Output(\tau(U, r \nwarrow 1))$;
if $(r \swarrow 1) \neq \varepsilon \rightarrow matches := matches \cup mat(u(r \nwarrow 1), r \swarrow 1, \tau(U, r \nwarrow 1))$
$[\!]\ (r \swarrow 1) = \varepsilon \rightarrow$ **skip**
fi;
$matches := matches \cup (\cup\ B : B \in N \wedge B \in \mathbf{lhs}(matches_0) : Output(\tau(U, B)))$;
$O := O \cup \{u\} \times matches_0$;
$mat := (\cup\ i : 0 < i : \{i - 1\} \times matches_i)$
corp

□

For efficiency reasons, we also consider two methods by which we can represent the set *matches*. Both of the methods rely upon the fact that P is a finite set, and that the integers (in the first components of variable *matches*) are in $[0, (\mathbf{MAX}\ p : p \in P : |p|) - 1]$ (the upperbound is one less than the length of the longest production).

1. Use signature $matches \in P \longrightarrow 2^{[0,(\mathbf{MAX}\ p:p\in P:|p|)-1]}$. In other words, map each production to a set of levels that it must be passed up to match locally. The representation can use an array (indexed by an integer associated with each production in P) of bit vectors (each of length $(\mathbf{MAX}\ p : p \in P : |p|)$) indicating the number of levels back that the production must be passed for local registration. All of the updates of *matches* can be done using bit vector operations (bitwise-OR). Finding which productions have matched locally is done by looking up those productions whose corresponding bit vector has the 0 bit set. Computing the final return value of the procedure is done by bit shifting all of the entries in the representation of *matches*.

2. Use signature $matches \in [0, (\mathbf{MAX}\ p : p \in P : |p|) - 1] \longrightarrow 2^P$. In this representation, we map each level (to be passed on the return) to the set of productions which match at that level. The representation can use an array (indexed by level number) of bit vectors (each of length $|P|$). Again, all of the updates of *matches* can be done using bit vector operations (bitwise-OR). Finding which productions have matched locally is done by looking up those productions in entry 0 of *matches*. The return value can be computed trivially if the array is represented as a circular array (in which the current 0 position is determined by a separate pointer); in this case, the return value only involves updating the pointer.

These representations would also yield interesting representations for function *Output*.

4 Right-to-left algorithms

In this section, we consider algorithms which traverse input string from right-to-left in general. Our first algorithm consists of a single repetition:

Algorithm 4.1:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{pref}(u) \cap \mathcal{L}(p) \neq \emptyset \,\}$
od$\{\ RPM\ \}$

□

The update of O in the repetition must still be implemented. This will be addressed shortly in Section 4.1. Another possible implementation is to use a nested repetition to traverse the prefixes of u in order of increasing length, as in the following algorithm:

Algorithm 4.2:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

In Section 4.2, this algorithm will be used as the starting point for the derivation of a particularly efficient new algorithm.

The inner repetition of the above algorithm could also be structured to consider prefixes of u in order of decreasing length:

Algorithm 4.3:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := u, \varepsilon;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge u \in \mathcal{L}(p) \,\};$
  do $v \neq \varepsilon \rightarrow$
    $v, r := v \searrow 1, (v \nearrow 1)r;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

It does not appear that there are any straightforward methods for improving the efficiency of Algorithm 4.3.

4.1 Improving the single-repetition algorithm

In this section, we make some improvements to Algorithm 4.1. The update of O can be made much simpler by introducing a new variable W:

$$W = \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\}$$

The resulting algorithm is:

Algorithm 4.4:

$l, u := S, \varepsilon;$
$W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \varepsilon \in \mathcal{L}(x) \,\};$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\};$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

The initialization of W can be greatly simplified using the chain-rule relation **crule**:

Algorithm 4.5:

$l, u := S, \varepsilon;$
$W := \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid \mathbf{rhs}(p) = \varepsilon \,\}));$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\};$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

We now need an efficient implementation of the update of W. We can derive the update as follows, starting with the new value (after the updates of l and u):

$$
\begin{array}{ll}
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}((l \nearrow 1)u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{property of } \mathbf{pref} \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (\{\varepsilon\} \cup (l \nearrow 1)\mathbf{pref}(u)) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \cap \text{ distributes over } \cup \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge ((l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \vee \varepsilon \in \mathcal{L}(x)) \,\} \\
= & \quad \{\, \text{split quantification} \,\} \\
 & \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \varepsilon \in \mathcal{L}(x) \,\} \\
 & \cup \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{use } \mathbf{crule}^* \text{ for first quantification} \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, x \mid x \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}(x) \neq \emptyset \,\} \\
= & \quad \{\, \text{change of bound variable in second quantification: } x = (l \nearrow 1)x' \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge (l \nearrow 1)\mathbf{pref}(u) \cap \mathcal{L}((l \nearrow 1)x') \neq \emptyset \,\} \\
= & \quad \{\, a\mathbf{pref}(u) \cap \mathcal{L}(ax') \neq \emptyset \Rightarrow \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset; \text{ first conjunct} \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset \,\} \\
= & \quad \{\, az \in \mathbf{suff}(W) \Rightarrow z \in \mathbf{suff}(W) \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup \{\, (l \nearrow 1)x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge \\
 & \qquad\qquad \mathbf{pref}(u) \cap \mathcal{L}(x') \neq \emptyset \,\} \\
= & \quad \{\, \text{invariant on } W \,\} \\
 & \{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\})) \\
 & \cup (l \nearrow 1)\{\, x' \mid (l \nearrow 1)x' \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge x' \in W \,\}
\end{array}
$$

Since the domain of W is finite, we can define a state set, and an initial state.

Definition 4.6 (State set): State set is defined as $Q = \mathbf{suff}(\mathbf{rhs}(P))$. The initial state q_0 is defined to be

$$\{\varepsilon\} \cup \mathbf{lhs}(\mathbf{crule}^*(\{\, p \mid \mathbf{rhs}(p) = \varepsilon \,\}))$$

□

Using this state set, we can also define a *transition* function (using the derivation above).

Definition 4.7 (Function σ): Transition function $\sigma \in Q \times V \longrightarrow Q$ is defined as

$$\sigma(q, a) = q_0 \cup a\{\, y \mid ay \in \mathbf{suff}(\mathbf{rhs}(P)) \wedge y \in q \,\}$$

□

The state set and the transition function can be used in the final algorithm:

Algorithm 4.8:

$l, u := S, \varepsilon;$
$W := q_0;$
$O := \{S\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $W := \sigma(W, l \nearrow 1);$
  $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \mathbf{rhs}(p) \in W \,\}$
od$\{\ RPM\ \}$

□

This algorithm can be simplified somewhat by defining an output function for the update of O, and by applying Aho-Corasick-like simplification of the state set.

4.2 Improving an algorithm with two repetitions

We begin with Algorithm 4.2, duplicated here:

Algorithm 4.9:

$l, u := S, \varepsilon;\ O := \{S\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \{\, p \mid p \in P \wedge \varepsilon \in \mathcal{L}(p) \,\};$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$
  od
od$\{\ RPM\ \}$

□

In this algorithm, as we consider prefixes of u of increasing length, we can make use of some information already stored in the set O. We will use the variable v to keep track of partial matches corresponding to right-hand sides of productions. Once we have a completed right-hand side, the match can be registered, along with any other matches induced by chain rules. We consider the two possible forms of right-hand sides separately.

We begin by rewriting the set

$$\{\, p \mid p \in P \wedge v \in \mathcal{L}(p) \,\}$$

(used in the inner repetition's update of O in the algorithm above, and catering to the simpler form of right-hand side) as

$$\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\})$$

We now turn to the second form of right-hand side. In the following derivation, we rely upon the fact that the outer repetition considers string S from right-to-left. We

would like to register a match when there is some nonterminal $A \in \mathbf{lhs}(O_{lv})$ (that is, A is the left-hand side of some production matching in r, with left context lv) and vA is the right-hand side of some production. More formally, the set of such matches is

$$\mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\})$$

We use these two formulas in the following algorithm:

Algorithm 4.10:

$l, u := S, \varepsilon;\ O := \{S\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
  do $r \neq \varepsilon \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\});$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\});$
  od
od$\{\ RPM\ \}$

□

The twin updates of O in the inner repetition arise from the fact that we have two different types of right-hand sides to consider.

In the above algorithm, we note that, once $v \notin \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$, it is not possible to find a further match by extending v on the right. It is thus possible to terminate the inner repetition once further iterations are futile. This is done by extending the inner repetition guard to $r \neq \varepsilon$ **cand** $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$.

This change also happens to give us the inner repetition invariant $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$, which is initially true by the redundant initialization of v. This invariant encompasses the information which we will later use to improve the algorithm. For this reason, we would also like to have this as an invariant of the outer repetition. This can be done by adding the initialization $v, r := \varepsilon, \varepsilon$ at the beginning of the program. All of these improvements yield the following algorithm:

Algorithm 4.11:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \varepsilon \,\});$
  do $r \neq \varepsilon$ **cand** $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = v \,\});$
    $O := O \cup \{l\} \times \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\});$
  od
od$\{\ RPM\ \}$

□

The evaluation of the inner repetition guard can be done by using a trie.

Definition 4.12 (Function τ_{red}): The *trie* for (the finite set of keywords) $\mathbf{rhs}(P)$ (over combined alphabet $N \cup V$) is function $\tau_{red} \in \mathbf{pref}(\mathbf{rhs}(P)) \times V \longrightarrow \mathbf{pref}(\mathbf{rhs}(P)) \cup \{\bot\}$ defined by

$$\tau_{red}(w,a) = \begin{cases} aw & \text{if } wa \in \mathbf{pref}(\mathbf{rhs}(P)) \\ \bot & \text{if } wa \notin \mathbf{pref}(\mathbf{rhs}(P)) \end{cases}$$

This function is known as the *reduced trie* — a compressed version of the function τ given in Definition 3.7. □

Using the trie, we rewrite the conditional conjunct $v(r \nwarrow 1) \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$ as

$$\tau_{red}(v, r \nwarrow 1) \neq \bot$$

(This hinges upon the fact that $\mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \subseteq \mathbf{pref}(\mathbf{rhs}(P))$ and $S \in V^*$.) To make the algorithm more concise, we also define the following output function:

Definition 4.13 (Output function): Function $Out \in \mathbf{pref}(\mathbf{rhs}(P)) \longrightarrow 2^P$ is defined by

$$Out(w) = \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = w \,\})$$

□

Function Out is easily precomputed. It is obvious that we can use Out for the first update of O in the inner repetition. We can use this function, along with the reverse trie, to rewrite the second update of O in the inner repetition, as follows:

$$\begin{array}{ll} & \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = vA \wedge A \in \mathbf{lhs}(O_{lv}) \,\}) \\ = & \quad \{\, \text{definition of } \tau_{red} \,\} \\ & \mathbf{crule}^*(\{\, p \mid p \in P \wedge \mathbf{rhs}(p) = \tau_{red}(v,A) \wedge A \in \mathbf{lhs}(O_{lv}) \,\}) \\ = & \quad \{\, \text{definition of } Out \,\} \\ & Out(\{\, \tau_{red}(v,A) \mid A \in \mathbf{lhs}(O_{lv}) \,\}) \end{array}$$

Using these two functions yields the following algorithm:

Algorithm 4.14:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times Out(\varepsilon);$
do $l \neq \varepsilon \rightarrow$
  $l, u := l \searrow 1, (l \nearrow 1)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times Out(\varepsilon);$
  do $r \neq \varepsilon$ **cand** $\tau_{red}(v, r \nwarrow 1) \neq \bot \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times Out(v);$
    $O := O \cup \{l\} \times Out(\{\, \tau_{red}(v,A) \mid A \in \mathbf{lhs}(O_{lv}) \,\} \setminus \{\bot\});$
  od
od{ RPM }

□

4.3 Greater shift distances

In a manner analogous to the Commentz-Walter and Boyer-Moore algorithm derivations in [14, Chapter 4] or [18, 20], we can use the invariant $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$ on subsequent iterations of the outer repetition to make a shift of $k \geq 1$ symbols by replacing the assignment $l, u := l \searrow 1, (l \nearrow 1)u$ by $l, u := l \searrow k, (l \nearrow k)u$.

As with the Commentz-Walter and Boyer-Moore algorithms, we would like an ideal shift distance — the shift distance to the nearest match to the left (in input string S). Formally, this distance is given by: $(\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n)$. Any shift distance less than this is also acceptable, and we define a safe shift distance (similar to that given in [14, Chapter 4] and in [18, 20]).

Definition 4.15 (Safe shift distance): A shift distance k satisfying

$$1 \leq k \leq (\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n)$$

is a *safe shift distance*. We call the upperbound (the quantification) the *maximal safe shift distance* or the *ideal shift distance*. □

Using a safe shift distance, the update of l, u then becomes $l, u := l \searrow k, (l \nearrow k)u$. In order to compute a safe shift distance, we will weaken predicate $\mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset$ (which we call the *ideal shift predicate*) in the range of the maximal safe shift distance quantification. This technique of using predicate weakening to find a more easily computed shift distance was introduced in [18] and used in [14, 20]. The weakest predicate, *true*, yields a shift distance of 1 — which, in turn, yields our last algorithm. We now find a weakening of the ideal shift predicate which is stronger than *true*, but still precomputable.

In the following weakening, we will first remove the dependency of the ideal shift predicate on r and then l. The particular weakening that we derive will prove to yield precomputable shift tables. Assuming $1 \leq n \leq |l|$ and the (implied) invariant $u = vr$, we begin with the ideal shift predicate:

$$\begin{array}{ll}
& \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset \\
\equiv & \quad \{\, \text{invariant: } u = vr \,\} \\
& \mathbf{pref}((l \nearrow n)vr) \cap \mathcal{L}(P) \neq \emptyset \\
\Rightarrow & \quad \{\, \text{discard lookahead to } r\text{: } r \in V^*, \text{ monotonicity of } \mathbf{pref} \text{ and } \cap \,\} \\
& \mathbf{pref}((l \nearrow n)vV^*) \cap \mathcal{L}(P) \neq \emptyset \\
\Rightarrow & \quad \{\, \text{domain of } l \text{ and } n\text{: } n \leq |l|, \text{ so } (l \nearrow n) \in V^n \,\} \\
& \mathbf{pref}(V^n vV^*) \cap \mathcal{L}(P) \neq \emptyset \\
\equiv & \quad \{\, \text{property of } \mathbf{pref} \text{ (see [14, Chapter 2])} \,\} \\
& V^n vV^* \cap \mathcal{L}(P)V^* \neq \emptyset \\
\Rightarrow & \quad \{\, \text{property of } \mathcal{L}(P)\text{: } \mathcal{L}(P) \subseteq \mathbf{vpart}(\mathbf{rhs}(P))V^* \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P))V^*V^* \neq \emptyset \\
\equiv & \quad \{\, V^*V^* = V^* \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset \\
\equiv & \quad \{\, \text{property of languages (see [14, Chapter 2])} \,\} \\
& V^n vV^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset \vee V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset
\end{array}$$

Note that we have removed the dependence upon l, meaning that we can remove the upper bound on n in the **MIN** quantification. Given the last line above, we have the following approximation:

$$\begin{array}{ll} & (\mathbf{MIN}\ n : 1 \leq n \leq |l| \wedge \mathbf{pref}((l \nearrow n)u) \cap \mathcal{L}(P) \neq \emptyset : n) \\ \geq & \quad \{\text{derivation above, disjunction in the resulting range predicate}\} \\ & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n v V^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset : n) \\ & \mathbf{min}(\mathbf{MIN}\ n : 1 \leq n \wedge V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset : n) \end{array}$$

This last line above can be written more concisely with the introduction of a pair of auxiliary functions.

Definition 4.16 (Functions b_1, b_2): We define two functions b_1, b_2 with signatures $b_1, b_2 \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P))) \longrightarrow \mathit{Naturals}$ (the domain comes from the fact that $v \in \mathbf{pref}(\mathbf{vpart}(\mathbf{rhs}(P)))$) as:

$$\begin{array}{rcl} b_1(x) & = & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n x V^* \cap \mathbf{vpart}(\mathbf{rhs}(P)) \neq \emptyset : n) \\ b_2(x) & = & (\mathbf{MIN}\ n : 1 \leq n \wedge V^n v \cap \mathbf{vpart}(\mathbf{rhs}(P))V^* \neq \emptyset : n) \end{array}$$

These two functions are, in fact, reversed versions of the Commentz-Walter shift functions (known as d_1 and d_2) for (finite) keyword set $\mathbf{vpart}(\mathbf{rhs}(P))$. Their precomputation is extremely well understood, and is detailed in [18, 20]. The precomputation algorithm involves the trie (for $\mathbf{rhs}(P)$) τ_{red}, introduced earlier. □

Using the auxiliary functions, our approximation of the ideal shift distance is $b_1(v)\,\mathbf{min}\,b_2(v)$. Using the new shift distance yields our final algorithm (with new variable h to hold the shift distance):

Algorithm 4.17:

$l, u := S, \varepsilon;$
$v, r := \varepsilon, \varepsilon;\ O := \{S\} \times Out(\varepsilon);$
do $l \neq \varepsilon \rightarrow$
  $h := b_1(v)\,\mathbf{min}\,b_2(v);$
  $l, u := l \searrow h, (l \nearrow h)u;$
  $v, r := \varepsilon, u;\ O := O \cup \{l\} \times Out(\varepsilon);$
  do $r \neq \varepsilon$ **cand** $\tau_{red}(v, r \nwarrow 1) \neq \perp \rightarrow$
    $v, r := v(r \nwarrow 1), r \swarrow 1;$
    $O := O \cup \{l\} \times Out(v);$
    $O := O \cup \{l\} \times Out(\{\, \tau_{red}(v, A) \mid A \in \mathbf{lhs}(O_{lv}) \,\} \setminus \{\perp\});$
  od
od$\{\ RPM\ \}$

□

4.3.1 Specializing the pattern matching algorithm

By restricting the form of the regular grammars, we can specialize Algorithm 4.17 to obtain a reversed version of the Commentz-Walter and the Boyer-Moore algorithms.

The most straightforward specialization is to restrict the productions to be of the form $A \longrightarrow w$ for $w \in V^*$ and each nonterminal appears as at most one left-hand side. From this restriction, we have $\mathbf{vpart}(\mathbf{rhs}(P)) = \mathbf{rhs}(P)$. In this case, the set of productions essentially represents a finite set of keywords $\mathbf{rhs}(P)$ (the left-hand sides are redundant). We can then delete the second update of O in the inner repetition, since it is used exclusively for productions with a nonterminal as the right-most symbol of the right-hand side. The resulting algorithm is the reversal to the Commentz-Walter algorithm without lookahead. (For a presentation of the Commentz-Walter algorithm, see [14, Section 4.4] or [19].)

We can similarly restrict the set of productions to consist of a single production $A \longrightarrow w$ for $w \in V^*$. In this case, we obtain a variant of the Boyer-Moore algorithm. (For a number of variants of the Boyer-Moore algorithm, see [14, Section 4.5] and [11].)

4.3.2 Improving the algorithm

We now briefly mention two approaches to improving this algorithm (both of which are discussed in more detail in [14, Chapters 4 and 5]):

- In the derivation of a weakened range predicate, we eliminated any (right) lookahead into string r by replacing it with V^*. We could have retained a single symbol of lookahead, by replacing r with $(r \nwarrow 1)V^*$. We could then have further manipulated the predicate and defined a third shift function.

- Also in the derivation, we discarded any (left) lookahead into l by replacing $l \nearrow n$ with V^n. We could have kept a single symbol of lookahead by replacing $l \nearrow n$ with $V^{n-1}(l \nearrow 1)$. This would also have yielded a different shift function.

5 Conclusions

We have succeeded in deriving and presenting a number of new algorithms for the regular grammar pattern matching problem. The interesting, and possibly efficient, algorithms include a generalization of the Boyer-Moore algorithm, a recursive algorithm (which resembles a generalized Aho-Corasick algorithm), and an algorithm based on a type of finite automaton.

Interestingly, the Boyer-Moore type algorithm presented here was only derived after a regular tree pattern matching version of the algorithm was developed [15].

Future directions include implementing all of the algorithms and collecting benchmarking data.

Acknowledgements:

I would like to thank Richard Watson, Dr. Kees Hemerik, Dr. Gerard Zwaan, and Prof. Dr. Frans Kruseman Aretz for their technical assistance during the development of these algorithms. A great deal of feedback was also provided by the participants at the Prague Stringologic Club Workshop '96 in Prague, in particular the organizers Prof. Dr. Melichar, Dr. Martin Bloch, and Ing. Jan Holub (who also provided a great deal of help with the typesetting). I thank Drs. Nanette Saes for proofreading and offering suggestions for improvement of this paper.

References

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

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

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

[4] Baeza-Yates, R.: *Efficient Text Searching.* (Ph.D dissertation, University of Waterloo, Canada, May 1989).

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

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

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

[8] Crochemore, M., Rytter, W.: *Text Algorithms.* (Oxford University Press, Oxford, England, 1994).

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

[10] Gonnet, G.H., Baeza-Yates, R.: *Handbook of Algorithms and Data Structures (In Pascal and C).* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

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

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

[13] Watson, B.W.: The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/pattm/`.

[14] Watson, B.W.: *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995). Contact `watson@RibbitSoft.com`.

[15] Watson, B.W.: A Boyer-Moore (or Watson-Watson) type algorithm for regular tree pattern matching, in: Aarts, E.H.L., ten Eikelder, H.M.M., Hemerik, C., Rem, M., eds., *Simplex Sigillum Veri: Een Liber Amicorum voor prof.dr.*

F.E.J. Kruseman Aretz (Eindhoven University of Technology, ISBN 90-386-0197-2, 1995) 315–320.

[16] Watson, B.W.: A new algorithm for regular grammar pattern matching, *Proceedings of the Fourth European Symposium on Algorithms,* Barcelona, Spain, 1996.

[17] Watson, B.W., Watson, R.E.: A Boyer-Moore type algorithm for regular expression pattern matching, Computing Science Report 94/31, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `watson@RibbitSoft.com`.

[18] Watson, B.W., Zwaan, G.: A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992. Available by e-mail from `watson@RibbitSoft.com` or `wsinswan@win.tue.nl`.

[19] Watson, B.W., Zwaan, G.: A taxonomy of sublinear multiple keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1994. Available by e-mail from `wsinswan@win.tue.nl`.

[20] Watson, B.W., Zwaan, G.: A taxonomy of sublinear multiple keyword pattern matching algorithms, to appear in: *Science of Computer Programming,* (1996).

[21] Zwaan, G.: Sublinear pattern matching, in: Aarts, E.H.L., ten Eikelder, H.M.M., Hemerik, C., Rem, M., eds., *Simplex Sigillum Veri: Een Liber Amicorum voor prof.dr. F.E.J. Kruseman Aretz* (Eindhoven University of Technology, ISBN 90-386-0197-2, 1995) 335–350.