Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Theoretical Computer Science 299 (2003) 509–521

Theoretical Computer Science

www.elsevier.com/locate/tcs

A new regular grammar pattern matching algorithm

Bruce W. Watson[a,b,*]

[a] *Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa*
[b] *Ribbit Software Systems Inc. & FST Labs, Kelowna, B.C., Canada*

Received 25 October 2000; received in revised form 26 February 2002; accepted 21 May 2002
Communicated by M. Crochemore

Abstract

This paper presents a Boyer–Moore type algorithm for regular grammar pattern matching, answering a variant of an open problem posed by Aho (Pattern Matching in Strings, Academic Press, New York, 1980, p. 342). The new algorithm handles patterns specified by regular (left linear) grammars—a generalization of the Boyer–Moore (single keyword) and Commentz-Walter (multiple keyword) algorithms.

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

In most cases, the Boyer–Moore (respectively, Commentz-Walter) algorithm has greatly outperformed the Knuth–Morris–Pratt (respectively, Aho–Corasick) algorithm. In practice, an earlier version of the algorithm presented in this paper also frequently outperforms the regular grammar generalization of the Aho–Corasick algorithm.

Keywords: Pattern matching algorithms; Regular grammars; Boyer–Moore algorithm

1. Introduction

The pattern matching problem is: given a regular pattern grammar (formally defined later, and itself consisting of a finite number of *productions*) and an input string S (over an alphabet V), find all substrings of S which correspond to the language denoted by

* Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa.
E-mail addresses: watson@cs.up.ac.za, watson@fst-labs.com (B.W. Watson).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00532-7

some production in the grammar. Several restricted forms of this problem have been solved (all of which are discussed in detail in [13, Chapter 4; 2,8,18,20):

- The Knuth–Morris–Pratt [12] and Boyer–Moore [5] algorithms solve the problem when productions are simplified to *keywords* (over alphabet V) and the problem is restricted to a single such keyword.
- The Aho–Corasick [3] and Commentz-Walter [6,7] algorithms also solve the problem when productions are simplified to keywords—though without the restriction to a single keyword. 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. The Aho–Corasick algorithm proper (not considering the *precomputation phase*) has performance independent of the number of keywords or the size of the alphabet, while the Commentz-Walter algorithm performs very well when there are relatively few long keywords.

Although both the Knuth–Morris–Pratt and Aho–Corasick algorithms have better worst-case running time than the Boyer–Moore and Commentz-Walter algorithms (respectively), the latter two algorithms are known to be extremely efficient in practice (see [13, Chapter 13; 11,15). Interestingly, to date no generalization (to the case where L is an arbitrary regular grammar) of the Boyer–Moore and Commentz-Walter algorithms has been discovered. In 1980, Aho stated the following open problem:

> It would also be interesting to know whether there exists a Boyer–Moore type algorithm for regular expression pattern matching. [1, p. 342].

In this paper, we present a regular *grammar* pattern matching algorithm. A related algorithm (which used regular expressions instead of regular grammars) was presented in [17] and in [13, Chapter 5]. That research was performed jointly with Richard E. Watson. An early version of the present paper was presented at the 1996 Symposium on Algorithms in Barcelona [16].

As with the Boyer–Moore and Commentz-Walter algorithms, the new algorithm requires shift tables. The precomputation of these shift tables is discussed, and shown to be related to the shift tables used by the Commentz-Walter algorithm, for which precomputation algorithms are well studied in [20, Section 4]. Finally, the new algorithm is specialized to obtain a variant of the Commentz-Walter (multiple keyword) algorithm—showing that it is indeed a generalization of the Boyer–Moore algorithm. The original version of this algorithm has been implemented, and in practice it frequently displays better performance than a regular generalization of the Aho–Corasick algorithm.

The derivation of the new algorithm closely parallels the development of the Commentz-Walter algorithm, especially in the use of predicate weakening to find a practically computed shift distance—see [13, Chapter 4] or [20]. In the Commentz-Walter algorithm, information from previous match attempts is used to make a shift of at least one symbol. The shift functions are finite, and can therefore be precomputed and tabulated. In the new algorithm, we also use information from previous match attempts. Directly using all of the information may yield shift functions which are infinite, and therefore impossible to precompute. The main result in the development of

the algorithm is a predicate weakening step which allows us to use finite shift functions in place of the (possibly) infinite ones—thereby yielding a practical algorithm. In [20], correctness proofs (which apply to the algorithms in this paper) are provided, and for space reasons these are *not* repeated here.

It should be noted that there does exist another regular pattern matching algorithm with good performance, due to Baeza-Yates [4,10]. That algorithm requires some precomputation on the input string, and is therefore suited to a different kind of problem than the one presented in this paper.

This paper is structured as follows:

- Section 2 gives the problem specification, and a simple first algorithm.
- Section 3 presents the essential idea of greater shift distances while processing the input text, as in the Boyer–Moore algorithm.
- Section 4 specializes the new pattern matching algorithm, obtaining the Commentz-Walter algorithm and the Boyer–Moore algorithm.
- Section 5 considers the running-time analysis of the algorithm and the precomputation phase.
- Section 6 discusses some techniques for further improving the performance of the algorithm.
- Section 7 presents the conclusions of this paper.

Before continuing with the algorithm's development, 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. An alphabet is a finite, nonempty set of symbols (sometimes called characters).

Throughout this paper, we will assume some fixed alphabet V. We use the standard notation V^* to denote the set of all strings over V.

Definition 2. For a given string z, define $\mathsf{suff}(z)$ to be the set of suffixes (including string z and the empty string ε) of z.

Definition 3. 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 on the left, a natural number on the right, and yielding a string):

- $w \uparrow k$ is the k **min** $|w|$ left-most symbols of w.
- $w \uparrow k$ is the k **min** $|w|$ right-most symbols of w.
- $w \downarrow k$ is the $|w| - k$ **max** 0 right-most symbols of w.
- $w \downarrow k$ is the $|w| - k$ **max** 0 left-most symbols of w.

Here, we have chosen an infix operator form for **min** and **max** to emphasize their associative and commutative nature, which will later be used when we quantify over these operators.

Definition 4. A regular pattern grammar (also known as a left linear grammar) is defined to be a three tuple, (V,N,P), where:

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

Note that, unlike usual grammars (for parsing, etc.), we do not have a "start symbol". We use the fixed regular grammar (V,N,P) throughout this paper.

Definition 5. We define a function vpart on right-hand sides as follows: $\mathsf{vpart}(w)$ is the longest suffix of w containing only symbols in V; that is, we drop the nonterminal on the left, if there is one. More formally, for right-hand side x: $x \in V^*$ we have

$$\mathsf{vpart}(x) = x$$

and for right-hand side Bx: $B \in N$, $x \in V^*$ we have

$$\mathsf{vpart}(Bx) = x.$$

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 a production to one which takes a set of productions.)

In terms of the set of describable languages, regular grammars are as powerful as regular expressions. Using regular grammars yields a slightly simpler algorithm—cf. [13, Chapter 5].

Definition 6. We define the function $\mathscr{L}$ by mapping productions to the (regular) languages they denote. The function is defined in the usual way; that is, $\mathscr{L}(p)$ $(p \in P)$ is the least language such that (for $A \in N$, $w \in V^*$):

$$\mathscr{L}(A \to w) = \{w\}$$

and (for $B \in N$)

$$\mathscr{L}(A \to Bw) = \mathscr{L}(B)\{w\}.$$

Claim 7. *We have the following useful property of the language of a production* $p \in P$:

$$\mathscr{L}(p) \subseteq V^*\mathsf{vpart}(\mathsf{rhs}(p)).$$

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

Definition 8. Productions of the form $A \to B$ (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 $A \to B$ has been matched as well.) For this reason, we define function $\mathsf{chain\_rule} \in 2^P \to 2^P$ (where 2^P denotes the set of all subsets of our production set P) as

$$\mathsf{chain\_rule}(U) = \{A \to B \mid A \to B \in P \wedge B \in \mathsf{lhs}(U)\}.$$

We define the function $\mathsf{chain\_rule}^*$ to be the reflexive and transitive closure of function $\mathsf{chain\_rule}$.

2. Problem specification and a simple first algorithm

We begin this section with a precise specification of the regular grammar pattern matching problem. In our presentation, we will consider substrings of S as suffixes of prefixes.

Definition 9. Given an input string $S \in V^*$, and our regular grammar, establish post-condition (named for *regular grammar pattern matching*) *RPM*:

$$O = \{(p, r) \mid ur = S \wedge \mathsf{suff}(u) \cap \mathscr{L}(p) \neq \emptyset\}.$$

Intuitively, this means that we are registering all productions which match at some substring of S, along with the right context (in S) of the match location. (Note that, for simplicity, we are registering our matches by their end-point.)

We describe our naïve first algorithm after [13, Algorithm 4.10]. In this algorithm, the prefixes (u) of S (the outer repetition) and the suffixes (v) of u (the inner repetition) are considered in order of increasing length. [1]

Algorithm 2.1.

$u, r := \varepsilon, S;\ \ O := \{p \mid p \in P \wedge \varepsilon \in \mathscr{L}(p)\} \times \{S\};$
do $r \neq \varepsilon \to$
  $u, r := u(r \uparrow 1), r \downarrow 1;$
  $l, v := u, \varepsilon;\ \ O := O \cup \{p \mid p \in P \wedge \varepsilon \in \mathscr{L}(p)\} \times \{r\};$
  do $l \neq \varepsilon \to$
    $l, v := l \downarrow 1, (l \uparrow 1)v;$
    $O := O \cup \{p \mid p \in P \wedge v \in \mathscr{L}(p)\} \times \{r\}$
  od
od
$\{\ RPM\ \}$

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

Note that we are still making some assumptions about our ability to evaluate membership in $\mathscr{L}(p)$. In the next section, we consider a more practical algorithm.

Later, we will use the notation O_x to refer to the set of productions in O which match ending at x (a suffix of S). More formally, $O_x = \{p \mid (p,x) \in O\}$.

2.1. A more practical algorithm

In the previous algorithm, as we consider suffixes 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 side separately.

We begin by rewriting the set

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

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

$$\mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = v\}).$$

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

$$\mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = Av \wedge A \in \mathsf{lhs}(O_{vr})\}).$$

We use these two formulas in the following algorithm.

Algorithm 2.2.

$u, r := \varepsilon, S;\ \ O := \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = \varepsilon\}) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r \uparrow 1), r \downarrow 1;$
  $l, v := u, \varepsilon;\ \ O := O \cup \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = \varepsilon\}) \times \{r\};$
  do $l \neq \varepsilon \rightarrow$
    $l, v := l \downarrow 1, (l \uparrow 1)v;$
    $O := O \cup \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = v\}) \times \{r\};$
    $O := O \cup \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = Av \wedge A \in \mathsf{lhs}(O_{vr})\}) \times \{r\}$
  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 $v \notin \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P)))$, it is not possible to find a further match by extending v on the left. It is thus possible to terminate the inner repetition once further iterations are futile. This is done by extending the inner repetition guard to

$$l \neq \varepsilon \ \textbf{cand} \ (l \upharpoonright 1)v \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P))).$$

This change also happens to give us the inner repetition invariant

$$v \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P))),$$

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. That can be done by adding the initialization $l, v := u, \varepsilon$ at the beginning of the program.

The evaluation of the inner repetition's new guard can be done by using a *reverse trie* [9], as is done in the Commentz-Walter algorithm.

Definition 10. The *reverse trie* for (the finite set of keywords) $\mathsf{rhs}(P)$ (over combined alphabet $N \cup V$) is function

$$\tau_r \in \mathsf{suff}(\mathsf{rhs}(P)) \times (N \cup V) \to \mathsf{suff}(\mathsf{rhs}(P)) \cup \{\bot\}$$

defined by

$$\tau_r(w, a) = \begin{cases} aw & \text{if } aw \in \mathsf{suff}(\mathsf{rhs}(P)), \\ \bot & \text{if } aw \notin \mathsf{suff}(\mathsf{rhs}(P)). \end{cases}$$

Using the trie, we rewrite the conditional conjunct $(l \upharpoonright 1)v \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P)))$ as $\tau_r(v, l \upharpoonright 1) \neq \bot$. (This hinges upon the fact that $\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P))) \subseteq \mathsf{suff}(\mathsf{rhs}(P))$ and $S \in V^*$.)

To make the algorithm more concise, we also define the following output function:

Definition 11. Output function $\mathsf{Output} \in \mathsf{suff}(\mathsf{rhs}(P)) \to 2^P$ is defined by

$$\mathsf{Output}(w) = \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = w\}).$$

It is obvious that we can use Output 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:

$$\begin{aligned} & \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = Av \wedge A \in \mathsf{lhs}(O_{vr})\}) \\ = \quad & \langle \text{definition of } \tau_r \rangle \\ & \mathsf{chain\_rule}^*(\{p \mid p \in P \wedge \mathsf{rhs}(p) = \tau_r(v, A) \wedge A \in \mathsf{lhs}(O_{vr})\}) \\ = \quad & \langle \text{definition of } \mathsf{Output} \rangle \\ & \mathsf{Output}(\{\tau_r(v, A) \mid A \in \mathsf{lhs}(O_{vr})\}). \end{aligned}$$

Using these two functions yields the following algorithm:

Algorithm 2.3.

$u, r := \varepsilon, S;$
$l, v := u, \varepsilon;\ \ O := \mathsf{Output}(\varepsilon) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $u, r := u(r \uparrow 1), r \downarrow 1;$
  $l, v := u, \varepsilon;\ \ O := O \cup \mathsf{Output}(\varepsilon) \times \{r\};$
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l \upharpoonright 1) \neq \perp \rightarrow$
    $l, v := l \downarrow 1, (l \upharpoonright 1)v;$
    $O := O \cup \mathsf{Output}(v) \times \{r\};$
    $O := O \cup \mathsf{Output}(\{\tau_r(v, A) \mid A \in \mathsf{lhs}(O_{vr})\}) \times \{r\}$
  od
od
$\{\ RPM\ \}$

3. Greater shift distances

In a manner analogous to the Commentz-Walter and Boyer–Moore algorithm derivations in [13, Chapter 4] or [18,20], we can use the invariant

$$v \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P)))$$

on subsequent iterations of the outer repetition to make a shift k of more than one symbol by replacing the assignment $u, r := u(r \uparrow 1), r \downarrow 1$ by $u, r := u(r \uparrow k), r \downarrow k$.

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 right (in input string S). Formally, this distance is given by

$$(\mathbf{MIN}\, n:\ 1 \leqslant n \leqslant |r| \wedge \mathsf{suff}(u(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset : n).$$

Any shift distance less than this is also acceptable, and we define a safe shift distance (similar to that given in [13, Chapter 4; 11,20).

Definition 12. A shift distance k satisfying

$$1 \leqslant k \leqslant (\mathbf{MIN}\, n:\ 1 \leqslant n \leqslant |r| \wedge \mathsf{suff}(u(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset : n)$$

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 k, the update of u, r then becomes $u, r := u(r \uparrow k), r \downarrow k$. In order to compute a safe shift distance, we will weaken predicate $\mathsf{suff}(u(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset$ (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 [13,20]. 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 l and then r. The particular weakening that we derive will prove to yield precomputable shift tables. Assuming $1 \leqslant n \leqslant |r|$ and the (implied) invariant $u = lv$, we begin with the ideal shift predicate:

$$
\begin{aligned}
& \mathsf{suff}(u(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset \\
\equiv \quad & \langle \text{invariant: } u = lv \rangle \\
& \mathsf{suff}(lv(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset \\
\Rightarrow \quad & \langle \text{discard lookahead to } l\text{: } l \in V^*, \text{ monotonicity of suff and } \cap \rangle \\
& \mathsf{suff}(V^* v(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset \\
\Rightarrow \quad & \langle \text{domain of } r \text{ and } n\text{: } n \leqslant |r|, \text{ so } (r \uparrow n) \in V^n \rangle \\
& \mathsf{suff}(V^* v V^n) \cap \mathscr{L}(P) \neq \emptyset \\
\equiv \quad & \langle \text{property of suff (see [13, Chapter 2])} \rangle \\
& V^* v V^n \cap V^* \mathscr{L}(P) \neq \emptyset \\
\Rightarrow \quad & \langle \text{property of } \mathscr{L}(P)\text{: } \mathscr{L}(P) \subseteq V^* \mathsf{vpart}(\mathsf{rhs}(P)) \rangle \\
& V^* v V^n \cap V^* V^* \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset \\
\equiv \quad & \langle V^* V^* = V^* \rangle \\
& V^* v V^n \cap V^* \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset \\
\equiv \quad & \langle \text{property of languages (see [13, Chapter 2])} \rangle \\
& V^* v V^n \cap \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset \vee v V^n \cap V^* \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset.
\end{aligned}
$$

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

$$
\begin{aligned}
& (\mathbf{MIN}\, n\text{: } 1 \leqslant n \leqslant |r| \wedge \mathsf{suff}(u(r \uparrow n)) \cap \mathscr{L}(P) \neq \emptyset : n) \\
\geqslant \quad & \langle \text{derivation above, disjunction in the resulting range predicate} \rangle \\
& (\mathbf{MIN}\, n\text{: } 1 \leqslant n \wedge V^* v V^n \cap \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset : n) \\
& \mathbf{min}(\mathbf{MIN}\, n\text{: } 1 \leqslant n \wedge v V^n \cap V^* \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset : n).
\end{aligned}
$$

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

Definition 13. We define two functions d_1, d_2 with signatures[2]

$$d_1, d_2 \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P))) \to \mathbb{N}$$

[2] The domain comes from the fact that $v \in \mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P)))$.

as

$$d_1(x) = (\mathbf{MIN}\, n\colon\ 1 \leqslant n \wedge V^* x V^n \cap \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset : n),$$

$$d_2(x) = (\mathbf{MIN}\, n\colon\ 1 \leqslant n \wedge x V^n \cap V^* \mathsf{vpart}(\mathsf{rhs}(P)) \neq \emptyset : n).$$

These two functions are, in fact, the Commentz-Walter shift functions for (finite) keyword set $\mathsf{vpart}(\mathsf{rhs}(P))$). Their precomputation is well understood, and is presented in detail in [20, Section 4]. The precomputation algorithm involves the reverse trie (for $\mathsf{rhs}(P)$) τ_r, introduced earlier.

Using the auxiliary functions, our approximation of the ideal shift distance is $d_1(v)$ **min** $d_2(v)$. Using the new shift distance yields our final algorithm (with new variable h to hold the shift distance):

Algorithm 3.1 (An efficient algorithm).

$u, r := \varepsilon, S;$
$l, v := u, \varepsilon;\ \ O := \mathsf{Output}(\varepsilon) \times \{S\};$
do $r \neq \varepsilon \rightarrow$
  $h := d_1(v)$**min** $d_2(v);$
  $u, r := u(r \uparrow h), r \downarrow h;$
  $l, v := u, \varepsilon;\ \ O := O \cup \mathsf{Output}(\varepsilon) \times \{r\};$
  do $l \neq \varepsilon$ **cand** $\tau_r(v, l \uparrow 1) \neq \perp \rightarrow$
    $l, v := l \downarrow 1, (l \uparrow 1)v;$
    $O := O \cup \mathsf{Output}(v) \times \{r\};$
    $O := O \cup \mathsf{Output}(\{\tau_r(v, A) \mid A \in O_{vr}\}) \times \{r\}$
  od
od
{ *RPM* }

4. Specializing the pattern matching algorithm

By restricting the form of the regular grammars, we can specialize the pattern matching algorithm to obtain the Commentz-Walter and the Boyer–Moore algorithms.

The most straightforward specialization is to restrict the productions to be of the form $A \rightarrow w$ for $w \in V^*$ and each nonterminal appears as at most one left-hand side. From this restriction, we have $\mathsf{vpart}(\mathsf{rhs}(P)) = \mathsf{rhs}(P)$. In this case, the set of productions essentially represents a finite set of keywords $\mathsf{rhs}(P)$ (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 left-most symbol of the right-hand side. The resulting algorithm is identical to the Commentz-Walter algorithm without lookahead. For a presentation of the Commentz-Walter algorithm, see [13, Section 4.4] or [20].

We can similarly restrict the set of productions to consist of a single production $A \to w$ 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 algorithms, see [13, Section 4.5; 11].)

5. Running time analysis

In this section, we consider the running-time analysis of the algorithm and the precomputation phase.

As is shown in [20, Section 4], precomputation and storage of $d_1, d_2, \mathsf{Output}$ and τ_r can all be done in time and space

$$\mathcal{O}(|\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(p)))|).$$

Note that $\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(p)))$ is the domain of $d_1, d_2, \mathsf{Output}$ and the co-domain of τ_r and that $|\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(p)))|$ is linear in the sum of the lengths of the right-hand sides, that is

$$|\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(p)))| \in \mathcal{O}\left(\sum_{p:p\in P} |\mathsf{vpart}(\mathsf{rhs}(p))|\right)$$

We assume that lookups in precomputed functions $d_1, d_2, \mathsf{Output}$ and τ_r all take constant time. This is typically achieved using an array implementation and encoding elements of $\mathsf{suff}(\mathsf{vpart}(\mathsf{rhs}(P)))$ as integers. Furthermore, we assume that the take and drop operators on strings can be implemented in constant time, as they are implemented as string indexing operations. Finally, with clever encoding and preallocated space, the update of set O can also be implemented in constant time.

The outer repetition of Algorithm 3.1 is executed $|S|$ times. The inner repetition is executed $\mathcal{O}((\mathbf{MAX}\ p\colon\ p \in P\colon\ |\mathsf{vpart}(\mathsf{rhs}(p))|))$ since the depth of the reverse trie is equal to the length of the vpart of longest right-hand side. It follows that the worst-case running time of Algorithm 3.1 is

$$\mathcal{O}(|S| \cdot (\mathbf{MAX}\ p\colon\ p \in P\colon\ |\mathsf{vpart}(\mathsf{rhs}(p))|))$$

The space requirement for the algorithm (not counting storing the precomputed functions) is $\mathcal{O}(|S|)$.

6. Improving the algorithm

We now briefly mention two approaches to improving this algorithm (both of which are discussed in more detail in [13, Chapters 4 and 5]). We only mention them briefly here:

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

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

7. Conclusions

We have achieved our aim of deriving an efficient generalized Boyer–Moore type pattern matching algorithm for regular grammar pattern matching. The stepwise derivation began with a simple, intuitive first algorithm. A more practical algorithm was obtained through a special characterization of 'matching productions', using the transitive closure of a relation to deal with chain rules in the pattern grammar. The algorithm was further improved through the introduction of a reverse trie. The idea of shift distances greater than one symbol (as in the Boyer–Moore and Commentz-Walter algorithms) was introduced. The use of predicate weakening was instrumental in deriving a practical approximation to the ideal shift distance. The resulting approximation makes use of two functions, which turn out to be the Commentz-Walter shift functions; an algorithm computing these functions has previously been derived with correctness arguments in [18,19,20].

It was shown that the Commentz-Walter and Boyer–Moore algorithms can be derived as special cases of our algorithm, showing the new algorithm to be a truly generalized pattern matching algorithm.

The algorithm presented here is deceptively simple; it is easily derived and understood, yet it took over 14 years from the statement of Aho's open problem to the derivation of a first algorithm—see [17]. Interestingly, the first algorithm (in [17]) was more complicated than the one given here. The algorithm presented here was only derived after a regular tree pattern matching version of the algorithm was developed [14].

Acknowledgements

I would like to thank Richard E. Watson, Kees Hemerik, Gerard Zwaan, and Frans Kruseman Aretz for their technical assistance during the development of this algorithm, and the ones leading up to it. I would particularly like to thank the referees for their valuable comments and Nanette Y. Saes for her proofreading and suggestions for improvement of this paper.

References

[1] A.V. Aho, Pattern Matching in Strings, Academic Press, New York, 1980, pp. 325–347.
[2] A.V. Aho, Algorithms for Finding Patterns in Strings, Vol. A. North-Holland, Amsterdam, 1990, pp. 257–300.
[3] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Comm. ACM 18 (6) (1975) 333–340.

[4] R. Baeza-Yates, Efficient text searching, Ph.D. Thesis, Computer Science, University of Waterloo, May 1989.
[5] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 62–72.
[6] B. Commentz-Walter, A string matching algorithm fast on the average, in: H. Maurer (Ed.), Proc. 6th Internat. Colloq. on Automata, Languages and Programming, Springer, Berlin, 1979, pp. 118–131.
[7] B. Commentz-Walter, A string matching algorithm fast on the average, Tech. Report 79.09.007, IBM Heidelberg Scientific Center, 1979.
[8] M.A. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.
[9] E. Fredkin, Trie memory, Comm. ACM 3 (9) (1960) 490–499.
[10] G.H. Gonnet, R. Baeza-Yates, Handbook of Algorithms and Data Structures (In Pascal and C), 2nd Edition, Addison-Wesley, Reading, MA, 1991.
[11] S. Hume, D. Sunday, Fast string searching, Software—Practice Experience 21 (11) (1991) 1221–1248.
[12] D.E. Knuth, J. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[13] B.W. Watson, Taxonomies and toolkits of regular language algorithms, Ph.D. Thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, September 1995.
[14] B.W. Watson, A Boyer-Moore (or Watson-Watson) type algorithm for regular tree pattern matching, in: E. Aarts, H. ten Eikelder, C. Hemerik, M. Rem (Eds.), Simplex Sigillum Veri: Een Liber Amicorum voor prof.dr. F.E.J. Kruseman Aretz, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1995, pp. 315–320.
[15] B.W. Watson, The performance of single and multiple keyword pattern matching algorithms, in: N. Ziviani, R. Baeza-Yates, K. Guimaraes (Eds.), Proc. 3rd South Amer. Workshop on String Processing, Internat. Informatics Series, Vol. 4, Carleton University Press, Recife, Brazil, 1996, pp. 280–294.
[16] B.W. Watson, A new regular grammar pattern matching algorithm, in: J. Diaz, M. Serna (Eds.), Proc. European Symp. on Algorithms, Lecture Notes in Computer Science, Vol. 1136, Springer, Berlin, Barcelona, Spain, 1996, pp. 364–377.
[17] B.W. Watson, R.E. Watson, A Boyer–Moore type algorithm for regular expression pattern matching, Tech. Report 31, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1994.
[18] B.W. Watson, G. Zwaan, A taxonomy of keyword pattern matching algorithms, Tech. Report No. 27, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1992.
[19] B.W. Watson, G. Zwaan, A taxonomy of sublinear keyword pattern matching algorithms, Tech. Report 13, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1995.
[20] B.W. Watson, G. Zwaan, A taxonomy of sublinear multiple keyword pattern matching algorithms, Sci. Comput. Programming 27 (2) (1996) 85–118.