A new family and structure for Commentz-Walter-style multiple-keyword pattern matching algorithms

A New Family and Structure for Commentz-Walter-style Multiple-Keyword Pattern Matching Algorithms

BW Watson

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa,
watson@cs.up.ac.za

Abstract

In this paper, I present a new family of Commentz-Walter-style multiple-keyword string pattern matching algorithms. The algorithms share a common algorithmic skeleton, which is significantly optimized when compared to the original Commentz-Walter skeleton and subsequently derived improvements. The new skeleton is derived via correctness-preserving stepwise algorithmic improvements, in the Eindhoven style of programming.
Keywords: *multiple-keyword string pattern matching, Commentz-Walter algorithm, algorithm derivation, algorithm optimization, stringology*
Computing Review Categories: *D.1.4, E.1, F.2.2, G.2.2*

1 Introduction

In this paper, I present a new family of Commentz-Walter-style multiple-keyword string pattern matching algorithms. The original Commentz-Walter algorithm appears in [4, 5]. The problem is: given a finite set of keywords P and an input string S (both over some alphabet V), find all occurrences (including overlapping ones) of a keyword (an element of P) as a substring of S, registering each such occurrence by its end-point within S. (End-point registration is relatively arbitrary — we could also have chosen to register an occurrence by its begin-point, or both begin- and end-points.)

The algorithm family presented in this paper is, in fact, very closely related to the original Commentz-Walter algorithm which was derived in the mid-to-late 1970s as a multiple-keyword generalization of the highly efficient Boyer-Moore [3] pattern matching algorithm. It operates by moving from left-to-right through input string S with an outer repetition. At each stop in S, a match attempt (conducted by an inner repetition) proceeds from right-to-left, registering any matches along the way. Following the inner repetition, the algorithm may make a shift to the right (using a so-called *shift function*) of more than one character — without missing any matches. This is done based on information gathered during the match attempt. In [10, 12], all of the (then-known) Commentz-Walter-style algorithms were presented with a common algorithmic *skeleton* — they only differed in the particular shift function used. In this paper, we present a significantly improved skeleton, which is able to use the same shift functions (and, therefore, all of their precomputable forms). In addition to the changed skeleton, the output function (used to register the matches) is changed — though it remains easily precomputed.

There have previously been numerous efforts to optimize the Commentz-Walter algorithms, though most of them have focused on new shift functions, more easily precomputed shift functions, or programming-language-specific coding tricks. A related algorithm derivation appeared as [11] — though the approach taken there differs significantly from the one presented here.

This paper is structured as follows:

- Section 2 gives a first naïve algorithm.
- Section 3 specializes the first algorithm to a more efficient (but still not directly implementable) one.
- Section 4 derives an efficient update of the variable O (used to store matches).
- Section 5 relates the given algorithm skeleton to the Commentz-Walter algorithm.
- Section 6 discusses the precomputation of the output function required in the new algorithm skeleton.
- Section 7 presents the closing comments of this paper.

1.1 Preliminaries

I assume that the reader is familiar with the field of multiple-keyword pattern matching, and with the Commentz-Walter algorithm in particular; for more information, see [12, 1, 7, 9, 10] or the bibliographies therein. The style of presentation is in Dijkstra's guarded commands [8]. In such a style, a number of seemingly redundant variables may be presented — though this is usually only to make repetition (loop) invariants easier to express. Furthermore, some variables are assumed to have inefficient types, such as 'substring of the input S', whereas an implementation in C, C++ or Pascal would likely use indexing into S.

Throughout this paper we assume a fixed alphabet V.

Definition 1 *Function* suff *maps a string or a set of strings to the corresponding set of suffixes (not necessarily proper, and including the empty string* ε*).*

In this paper, we use the *reverse trie* for P [12], which is defined as follows:

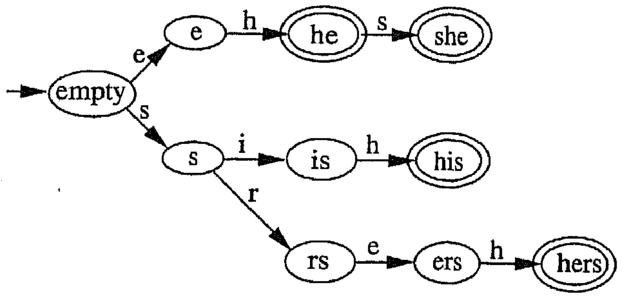
Definition 2 *The reverse trie for P is function*

$$\tau_r \in \mathsf{suff}(P) \times V \longrightarrow \mathsf{suff}(P) \cup \{\perp\}$$

defined by

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

Example 1 *(Reverse trie.) The reverse trie for keywords* $\{he, she, his, hers\}$ *is:* hershey*:*



In order to manipulate strings easily we will use the following operators:

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):*

- $w \upharpoonright k$ *is the* $k\,\mathbf{min}\,|w|$ *left-most symbols of* w.
- $w \upharpoonleft k$ *is the* $k\,\mathbf{min}\,|w|$ *right-most symbols of* w.
- $w \downharpoonleft k$ *is the* $|w| - k\,\mathbf{max}\,0$ *right-most symbols of* w.
- $w \downharpoonright k$ *is the* $|w| - k\,\mathbf{max}\,0$ *left-most symbols of* w.

Definition 4 *Define binary operator* $\mathbf{max}_s$ *to return the maximum of two strings, under the suffix ordering — assuming that either the left operand is a suffix of the right one, or vice-versa.*

2 A first algorithm

In this section, we present a first (brute-force) algorithm which we will later refine. Assuming we register matches in some variable O, the pattern matching postcondition is:

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

That is, in variable O, we accumulate pairs (v,r) where $v \in P$ is an occurrence of a pattern and r is its right context within S. We can rewrite the right-hand side:

$$\begin{array}{ll} & (\cup\, l,v,r : lvr = S \wedge v \in P : \{(v,r)\}) \\ = & \langle \text{two nested quantifications with } u \text{ as new dummy variable} \rangle \\ & (\cup\, u,r : ur = S : (\cup\, l,v : lv = u \wedge v \in P : \{(v,r)\})) \\ = & \langle \text{eliminate dummy } l \text{ and note that } v \in \mathsf{suff}(u) \rangle \\ & (\cup\, u,r : ur = S : (\cup\, v : v \in \mathsf{suff}(u) \wedge v \in P : \{(v,r)\})) \\ = & \langle \text{reverse distribute} \times \text{over} \cup \rangle \\ & (\cup\, u,r : ur = S : (\cup\, v : v \in \mathsf{suff}(u) \wedge v \in P : \{v\}) \\ & \times \{r\}) \\ = & \langle \text{set comprehension} \rangle \\ & (\cup\, u,r : ur = S : (\mathsf{suff}(u) \cap P) \times \{r\}) \end{array}$$

We can now re-state our postcondition as R:

$$O = (\cup\, u,r : ur = S : (\mathsf{suff}(u) \cap P) \times \{r\})$$

In the first algorithm, we will use variables u, r (to iterate over input string S) and O (to accumulate matches). Variable u will proceed from from left to right through S (from ε to S), and so we choose as invariant P_0:

$$\begin{array}{lll} ur & = & S \wedge O \\ & = & (\cup\, u',r' : u'r' = S \wedge r \in \mathsf{suff}(r') : (\mathsf{suff}(u') \cap P) \times \\ & & \{r'\}) \end{array}$$

The first algorithm is then:

Algorithm 2.1:

```
$u,r := \varepsilon, S$;
$O := (\mathsf{suff}(u) \cap P) \times \{r\}$;
{ invariant: $P_0$
  variant: $|r|$ }
do $r \neq \varepsilon \rightarrow$
    $u,r := u(r \upharpoonright 1), r \downharpoonleft 1$;
    $O := O \cup (\mathsf{suff}(u) \cap P) \times \{r\}$
od
{ $R$ }
```

□

Clearly, the update of O in the repetition needs elaboration. This is pursued in the following section.

3 Implementing the update of O

The derivation in this section is closely related to the one presented in [10, Chapter 4].

To update O, we could introduce some variable[1] W with invariant $W = \mathsf{suff}(u) \cap P$. This is, of course, as difficult as implementing our first algorithm and a compromise is to choose some set $X : P \subseteq X$, in which case we could maintain invariant $W = \mathsf{suff}(u) \cap X$. The update of O, which was by

$$(\mathsf{suff}(u) \cap P) \times \{r\}$$

is now by

$$(W \cap P) \times \{r\}$$

This is still somewhat unsatisfactory — though fortunately we can characterize the set $\mathsf{suff}(u) \cap X$ (W, which we no longer need explicitly) concisely. Set $\mathsf{suff}(u)$ has the nice property of being *linearly ordered* (both according to length and also according to the suffix (partial) relation on strings) — meaning that set $\mathsf{suff}(u) \cap X$ has a unique longest element. We introduce variable v such that it is that longest element — that is

$$v = (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap X : x)$$

This leads to the following derivation:

$$\begin{array}{ll} & \mathsf{suff}(u) \cap X \\ = & \langle \text{ split } \mathsf{suff}(u) \text{ into } \{z \mid z \in \mathsf{suff}(u) \wedge |z| > |v|\} \\ & \quad \text{and } \mathsf{suff}(v) \rangle \\ & (\{z \mid z \in \mathsf{suff}(u) \wedge |z| > |v|\} \cup \mathsf{suff}(v)) \cap X \\ = & \langle \text{ distribute } \cap \text{ over } \cup \rangle \\ & (\{z \mid z \in \mathsf{suff}(u) \wedge |z| > |v|\} \cap X) \cup (\mathsf{suff}(v) \cap X) \\ = & \langle \text{ by } v \text{ being the longest,} \\ & \quad \{z \mid z \in \mathsf{suff}(u) \wedge |z| > |v|\} \cap X = \emptyset \rangle \\ & \mathsf{suff}(v) \cap X \end{array}$$

The update of O is now by

$$((\mathsf{suff}(v) \cap X) \cap P) \times \{r\}$$

Thanks to the fact that $P \subseteq X$, we can further rewrite this as

$$(\mathsf{suff}(v) \cap P) \times \{r\}$$

With this characterization, we have the following algorithm.

Algorithm 3.1:

```
$u, r := \varepsilon, S$;
$v := (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap X : x)$;
$O := (\mathsf{suff}(v) \cap P) \times \{r\}$;
{ invariant: $P_0$
  variant: $|r|$ }
do $r \neq \varepsilon \to$
    $u, r := u(r \upharpoonleft 1), r \downharpoonleft 1$;
    $v := (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap X : x)$;
    $O := O \cup (\mathsf{suff}(v) \cap P) \times \{r\}$
od
{ $R$ }
```

□

In the next section, we elaborate on the assignment to v.

There are three straightforward choices for X (satisfying the requirement $P \subseteq X$):

1. Use the set of prefixes of P (known as $\mathsf{pref}(P)$). In [10, Chapter 4], this leads directly to the Aho-Corasick family of algorithms [2].

2. Use the set of factors of P (the set of all — not necessarily proper — substrings of elements of P). In [6], this approach is used to arrive at some efficient new algorithms.

3. Use the set $\mathsf{suff}(P)$.

We only pursue the third approach in this paper, and we now specialize parts of the algorithm according to our choice.

4 Choosing $X = \mathsf{suff}(P)$ and computing v

The obvious way to compute a maximal element of a linearly ordered set is using a linear search[2]. We introduce a new variable l and an inner repetition with invariant P_1:

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

This results in the following algorithm (for reasons explained in §5, we want P_1 as an invariant of the outer repetition as well, and we initialize l, v at the algorithm beginning as well):

Algorithm 4.1:

```
$u, r := \varepsilon, S$;
$l, v := u, \varepsilon$;
$O := (\mathsf{suff}(v) \cap P) \times \{r\}$;
{ invariant: $P_0 \wedge P_1$
  variant: $|r|$ }
do $r \neq \varepsilon \to$
    $u, r := u(r \upharpoonleft 1), r \downharpoonleft 1$;
    $l, v := u, \varepsilon$;
    { invariant: $P_1$
      variant: $|(\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)| - |v|$ }
    do $l \neq \varepsilon \wedge (l \upharpoonright 1)v \in \mathsf{suff}(P) \to$
        $l, v := l \downharpoonright 1, (l \upharpoonright 1)v$
    od;
    { $v = (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)$ }
    $O := O \cup (\mathsf{suff}(v) \cap P) \times \{r\}$
od
{ $R$ }
```

□

[1]Our algorithms will not contain variable W — we only use it here for illustrative purposes.

[2]This raises the question of whether other approaches are possible; this is not considered here, though we note later that other approaches can lead to the Aho-Corasick algorithm [2].

The guard of the inner repetition can also be specialized. In particular, the test $(l \upharpoonright 1)v \in \mathsf{suff}(P)$ can be implemented using the reverse trie transition function τ_r since $v \in \mathsf{suff}(P)$ we get

$$(l \upharpoonright 1)v \in \mathsf{suff}(P) \equiv \tau_r(v, l \upharpoonright 1) \neq \bot$$

We can also introduce an *output function* used in the update of O.

Definition 5 *Function* $\mathsf{WCWOutput} \in \mathsf{suff}(P) \longrightarrow \mathcal{P}(P)$ *is defined as*

$$\mathsf{WCWOutput}(x) = \mathsf{suff}(x) \cap P$$

This function can be easily precomputed by a traversal of the reverse trie, as discussed in §6.

This gives algorithm:

Algorithm 4.2:

```
$u, r := \varepsilon, S$;
$l, v := u, \varepsilon$;
$O := \mathsf{WCWOutput}(v) \times \{r\}$;
{ invariant: $P_0 \wedge P_1$
  variant: $|r|$ }
do $r \neq \varepsilon \rightarrow$
    $u, r := u(r \rceil 1), r \rfloor 1$;
    $l, v := u, \varepsilon$;
    { invariant: $P_1$
      variant: $|(\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)| - |v|$ }
    do $l \neq \varepsilon \wedge \tau_r(v, l \upharpoonright 1) \neq \bot \rightarrow$
        $l, v := l \lfloor 1, (l \upharpoonright 1)v$
    od;
    { $v = (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)$ }
    $O := O \cup \mathsf{WCWOutput}(v) \times \{r\}$
od
{ $R$ }
```

□

5 Greater shifts

By design, our algorithm has precisely the same invariants as the Commentz-Walter algorithm, cf. [10, Chapter 4] and [12]. This allows us to change the update of u, r, in the outer repetition, to use a shift $k(l, v, r)$ (k is the shift function which can depend on l, v and r). The details of such shift functions are beyond the scope of this paper, but can be found in [12, 10]. The final algorithm skeleton (in which we underline the main statement differing from the Commentz-Walter skeleton) is:

Algorithm 5.1 (Watson skeleton):

```
$u, r := \varepsilon, S$;
$l, v := u, \varepsilon$;
$O := \mathsf{WCWOutput}(v) \times \{r\}$;
{ invariant: $P_0 \wedge P_1$
  variant: $|r|$ }
do $r \neq \varepsilon \rightarrow$
    $shift := k(l, v, r)$;
    $u, r := u(r \rceil shift), r \rfloor shift$;
    $l, v := u, \varepsilon$;
    { invariant: $P_1$
      variant: $|(\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)| - |v|$ }
    do $l \neq \varepsilon \wedge \tau_r(v, l \upharpoonright 1) \neq \bot \rightarrow$
        $l, v := l \lfloor 1, (l \upharpoonright 1)v$
    od;
    { $v = (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)$ }
    $\underline{O := O \cup \mathsf{WCWOutput}(v) \times \{r\}}$
od
{ $R$ }
```

□

Note that we retain the invariants found in the original skeleton, meaning that exactly the same shift functions may be used.

For purposes of comparison, the original Commentz-Walter algorithm is [5, 4]:

Algorithm 5.2 (Commentz-Walter skeleton):

```
$u, r := \varepsilon, S$;
$l, v := u, \varepsilon$;
$O := (\{v\} \cap P) \times \{r\}$;
{ invariant: $P_0 \wedge P_1$
  variant: $|r|$ }
do $r \neq \varepsilon \rightarrow$
    $shift := k(l, v, r)$;
    $u, r := u(r \rceil shift), r \rfloor shift$;
    $l, v := u, \varepsilon$;
    $O := O \cup (\{v\} \cap P) \times \{r\}$;
    { invariant: $P_1$
      variant: $|(\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)| - |v|$ }
    do $l \neq \varepsilon \wedge \tau_r(v, l \upharpoonright 1) \neq \bot \rightarrow$
        $l, v := l \lfloor 1, (l \upharpoonright 1)v$;
        $\underline{O := O \cup (\{v\} \cap P) \times \{r\}}$
    od
    { $v = (\mathbf{MAX}_s x : x \in \mathsf{suff}(u) \cap \mathsf{suff}(P) : x)$ }
od
{ $R$ }
```

□

The difference between the two algorithms above is the lowering of the O update out of the repetition.

6 Precomputing WCWOutput

In this section, we briefly discuss precomputation of our output function $\mathsf{WCWOutput} \in \mathsf{suff}(P) \longrightarrow \mathcal{P}(P)$. The following derivation (in which $x \in \mathsf{suff}(P)$) gives us a recursive form for this function, enabling us to devise a depth-first traversal of the set $\mathsf{suff}(P)$ (which, conveniently, is the state set of the reverse trie):

$\mathsf{WCWOutput}(x)$

$$
\begin{aligned}
= \quad & \langle \text{definition of WCWOutput} \rangle \\
& \mathsf{suff}(x) \cap P \\
= \quad & \langle \text{definition of suff} \rangle \\
& (\{x\} \cup \mathsf{suff}(x \rfloor 1)) \cap P \\
= \quad & \langle \text{distribute } \cap \text{ over } \cup \rangle \\
& (\{x\} \cap P) \cup (\mathsf{suff}(x \rfloor 1) \cap P) \\
= \quad & \langle \text{definition of WCWOutput} \rangle \\
& (\{x\} \cap P) \cup \mathsf{WCWOutput}(x \rfloor 1)
\end{aligned}
$$

Armed with this, we can provide the following recursive algorithm, using the auxiliary procedure (in which *Out* is a global variable initially mapping each element of $\mathsf{suff}(P)$ to $\emptyset$):

Algorithm 6.1 (WCWOutput **precomputation):**

```
$visit\_node(\varepsilon)$
{ $Out = \mathsf{WCWOutput}$ }

proc $visit\_node(x : \mathsf{suff}(P)) \rightarrow$
if $x \in P \rightarrow Out(x) := \{x\}$
[] $x \notin P \rightarrow Out(x) := \emptyset$
fi;
{ $Out(x \rfloor 1) = \mathsf{WCWOutput}(x \rfloor 1)$ }
$Out(x) := Out(x) \cup Out(x \rfloor 1)$;
{ $Out(x) = \mathsf{WCWOutput}(x)$ }
for $a : a \in V \wedge \tau_r(x,a) \neq \perp \rightarrow$
    $visit\_node(ax)$
rof
corp
```

□

7 Closing remarks

Examining the generated machine code (from a compiler such as Gnu C++) reveals that the new skeleton's inner repetition consists of instructions typically consuming 5 clock cycles on a Pentium III, as compared to 6 clock cycles for the original algorithm. The overhead is in checking whether a keyword has matched and registering such a match. Since much (upwards of 80%) of the execution time is spend in the inner repetition, we can expect the new skeleton to run substantially faster than the original skeleton.

References

[1] Alfred V. Aho. *Algorithms for finding patterns in strings*, volume A, pages 257–300. North-Holland, 1990.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

[4] Barbara Commentz-Walter. A string matching algorithm fast on the average. Technical Report 79.09.007, IBM Heidelberg Scientific Center, 1979.

[5] Barbara Commentz-Walter. A string matching algorithm fast on the average. In H.A. Maurer, editor, *Proceedings of the Sixth International Colloquium on Automata, Languages and Programming*, pages 118–131. Springer-Verlag, 1979.

[6] Maxime A. Crochemore, A. Czumaj, L. Gąsieniec, Thierry Lecroq, T. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71:3–4, 1999.

[7] Maxime A. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[8] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

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

[10] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, September 1995.

[11] Bruce W. Watson. A new family of commentz-walter-style multiple-keyword pattern matching algorithms. In Borivoj Melichar, editor, *Proceedings of the Fifth Prague Stringologic Workshop*, Bratislava, Slovakia, September 2000. Czech Technical University.

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