A new family of Commentz-Walter-style multiple-keyword pattern matching algorithms

Bruce W. Watson

watson@OpenFIRE.org
www.OpenFIRE.org
University of Pretoria
(Department of Computer Science)
Pretoria 0002, South Africa

1 Introduction

In this paper, I present a new family of Commentz-Walter-style multiple-keyword string pattern matching algorithm. The problem is: given a finite set of keywords P and an input string S, find all occurrences (including overlapping ones) of a keyword (in P) as a sub string of S; register such an occurrence by the end-point within S.

The algorithm family presented in this paper is, in fact, very closely related to the original Commentz-Walter algorithm. In [3, 4], all of the (then-known) Commentz-Walter-style algorithms were presented with a common algorithmic *skeleton* — they only differed in the *shift* function used to move through the input string. 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 computed. 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, and programming-language-specific coding tricks.

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 [3] or the bibliography therein. The style of presentation is in Dijkstra's guarded commands [2]. In such a style, a number of seemingly redundant variables may be presented — though this is usually only to make repetition (repetition) invariants easier to express. Furthermore, some variables are assumed to have inefficient types, such as 'sub string of the input S', whereas an implementation in C, C++ or Pascal would likely use indexing into S.

The mathematical prerequisites are quite simple. Function suff maps a string or a set of strings to the corresponding set of suffixes (including the empty string ε). In this paper, we use the *reverse trie* for P; such a trie consists of state set $\mathsf{suff}(P)$ (though, in practice, they would be represented using integers instead of strings), and transition function τ_r, which maps a state and an input character to the corresponding extended element (that is, state) of $\mathsf{suff}(P)$ or to $\perp$ if there is no such element. The notation $z \restriction 1$ refers to the rightmost symbol of z.

2 The Commentz-Walter algorithm

The Commentz-Walter algorithm was derived in the mid- to late-1970s, as a multiple-keyword generalization of the highly efficient Boyer-Moore style of pattern matching algorithm. It operates by moving from left to right through input string S. 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 of more than one character — without missing any matches. This is done based on information gathered during the match attempt.

The following rendition of the algorithm is taken directly from [3], where τ_r is the transition function of the *reverse trie* (there are also some other some minor adjustments of notation to avoid introducing many new definitions):

Algorithm 2.1 (Commentz-Walter skeleton):

$u, r := \varepsilon, S;$
$l, v := \varepsilon, \varepsilon;$
$O := (\{\varepsilon\} \cap P) \times \{S\};$
{ invariant: $u = lv \wedge v \in \mathsf{suff}(P)$ }
do $r \neq \varepsilon \rightarrow$
    $u, r :=$ shift u, r right by $k(l, v, r)$;
    $l, v := u, \varepsilon;$
    $O := O \cup (\{\varepsilon\} \cap P) \times \{r\};$
    { invariant: $u = lv \wedge v \in \mathsf{suff}(P)$ }
    do $l \neq \varepsilon \wedge \tau_r(v, l \restriction 1) \neq \perp \rightarrow$
        $l, v :=$ shift l, v right by one character;
        $O := O \cup (\{v\} \cap P) \times \{r\}$
    od

od
$\{\ O = \{\ (p, r) \mid ur = S \wedge p \in \mathsf{suff}(u)\ \}\ \}$

□

Note that the states of the reverse trie are elements of the set $\mathsf{suff}(P)$, with ε being the start state.

The expression $k(l, v, r)$ is a place-holder for any of the shift functions given in either the original Commentz-Walter article [1] or in [3, Chapter 4]. A great many of these shift functions were developed subsequent to the original article — and some of them have proven to have extremely high performance, while being reasonably easy to precompute.

The ability of these algorithms to skip large portions of input S (without examining every single character) yields high performance — usually exceeding that of the Aho-Corasick algorithm, as is shown in [3, Chapter 13]. (This begs the question: why is the Aho-Corasick algorithm more popular in text-books and implementations? The Commentz-Walter family of algorithms have worst-case performance which is substantially below that of the (stable) Aho-Corasick variants; the Aho-Corasick algorithms have also been particularly easy to implement compared to the Commentz-Walter algorithms.)

Despite this superb performance, profiling the Commentz-Walter skeleton shows some interesting facts:

1. A great deal of the time is spend making transitions (using the reverse trie) in the inner repetition.
2. A similar amount of time is spent checking (also in the inner repetition) whether a match has been encountered.

The first item appears (at this time) unavoidable, whereas the second one is amenable to optimization (as shown in the rest of this paper). The registration of matches can be expressed as a *repetition-invariant computation*, meaning that it can be removed from the inner repetition, dramatically improving performance.

3 Some observations

The inner repetition iterates (using variable v) over the set $\mathsf{suff}(u) \cap \mathsf{suff}(P)$. We can characterize this set in a way which will prove easier to implement. In the following derivation, let w be the longest element of $\mathsf{suff}(u) \cap \mathsf{suff}(P)$ (there is a unique longest element, since $\mathsf{suff}(u)$ is linearly ordered according to the suffix relation):

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

The inner repetition registers, as matches, only those elements $v : v \in P$. It follows that the set registered during a single iteration of the outer repetition is $v \in \mathsf{suff}(u) \cap \mathsf{suff}(P) \cap P$, which can be rewritten:

$$
\begin{array}{ll}
 & \{ v \mid v \in \mathsf{suff}(u) \cap \mathsf{suff}(P) \wedge v \in P \} \\
= & \quad \{ \text{ set comprehension } \} \\
 & \{ v \mid v \in \mathsf{suff}(u) \cap \mathsf{suff}(P) \cap P \} \\
= & \quad \{ \text{ set comprehension } \} \\
 & \mathsf{suff}(u) \cap \mathsf{suff}(P) \cap P \\
= & \quad \{ \text{ previous derivation } \mathsf{suff}(u) \cap \mathsf{suff}(P) = \mathsf{suff}(w) \cap \mathsf{suff}(P) \} \\
 & \mathsf{suff}(w) \cap \mathsf{suff}(P) \cap P \\
= & \quad \{ \text{ set containment: } P \subseteq \mathsf{suff}(P) \} \\
 & \mathsf{suff}(w) \cap P
\end{array}
$$

This yields a set by which O could be updated after, instead of during, the inner repetition.

The remaining problem is to calculate w. Fortunately, the post-condition of the inner repetition is that v is the longest element of $\mathsf{suff}(u)$ which is still an element of $\mathsf{suff}(P)$ — $v = w$.

For conciseness, we can define function $\mathsf{WCWOutput} \in \mathsf{suff}(P) \longrightarrow \mathcal{P}(P)$ as

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

4 The new algorithm

Given the repetition-invariance of the updates (to variable O, in which matches are registered), we can present the new algorithm skeleton:

Algorithm 4.1 (Watson-Commentz-Walter skeleton):

$u, r := \varepsilon, S;$
$l, v := \varepsilon, \varepsilon;$
$O := (\{\varepsilon\} \cap P) \times \{S\};$
{ invariant: $u = lv \wedge v \in \mathsf{suff}(P)$ }
do $r \neq \varepsilon \rightarrow$
    $u, r :=$ shift u, r right by $k(l, v, r)$;
    $l, v := u, \varepsilon;$
    $O := O \cup (\{\varepsilon\} \cap P) \times \{r\};$
    { invariant: $u = lv \wedge v \in \mathsf{suff}(P)$ }
    do $l \neq \varepsilon \wedge \tau_r(v, l \restriction 1) \neq \perp \rightarrow$
        $l, v :=$ shift l, v right by one character;
    od;
    $O := O \cup \mathsf{WCWOutput}(v) \times \{r\}$
od
{ $O = \{ (p, r) \mid ur = S \wedge p \in \mathsf{suff}(u) \}$ }

□

Note that we retain the invariants found in the original skeleton, meaning that exactly the same shift functions may be used. The precomputation of *WCWOutput* remains to be discussed. As of this writing, an efficient algorithm has not been derived, though it is clearly possible to precompute it via a brute-force method. In all likelihood, an efficient precomputation algorithm will involve a breadth-first traversal of the reverse trie (the states of which, $\mathsf{suff}(P)$, form the domain of *WCWOutput*). This will be discussed further in the final version of this paper.

5 Closing comments

In this section, we briefly discuss the expected[1] performance and the history of this algorithm variant.

Examining the generated machine code (from a compiler such as Gnu C++) reveals that the new skeleton's inner repetition consists of roughly half the number of instructions as in the original algorithm's skeleton. Since much (upwards of 80%) of the execution time is spend in the inner repetition, we can expect WCW to run in roughly 60

The derivation presented here is markedly different from the one first obtained in 1999. In the first derivation, I started with a naïve algorithm

[1] Only the expected performance is presented here, since benchmarking is ongoing and will be presented in the final version of the paper.

which was used as a precursor to the Aho-Corasick algorithm in [3]. From that algorithm, a simple adjustment (in the step used to obtain the Aho-Corasick algorithm) yields the invariant that we need to register matches $\mathsf{suff}(u) \cap \mathsf{suff}(P) \cap P$. As in this paper, this set can be characterized by its longest element — eventually leading to the WCW algorithm. The alternative derivation is, in some sense, preferrable since it also allows for sets other than $\mathsf{suff}(P)$ in the registration process. In particular, $\mathsf{pref}(P)$ can be used (leading to the Aho-Corasick algorithm); more interestingly, the set of *factors* of P can be used — perhaps creating a new algorithm.

References

1. Commentz-Walter, B. A string matching algorithm fast on the average. (Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979).
2. Dijkstra, E.W. *A Discipline of Programming.* (Prentice Hall, Englewood Cliffs, N.J., 1976).
3. Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995). See `www.OpenFIRE.org`
4. Watson, B.W. and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. (Science of Computer Programming 27(2), September 1996, pp. 85–118).