On the Use of Regular Expressions for Searching Text

CHARLES L. A. CLARKE and GORDON V. CORMACK
University of Waterloo

The use of regular expressions for text search is widely known and well understood. It is then surprising that the standard techniques and tools prove to be of limited use for searching structured text formatted with SGML or similar markup languages. Our experience with structured text search has caused us to reexamine the current practice. The generally accepted rule of "leftmost longest match" is an unfortunate choice and is at the root of the difficulties. We instead propose a rule which is semantically cleaner. This rule is generally applicable to a variety of text search applications, including source code analysis, and has interesting properties in its own right. We have written a publicly available search tool implementing the theory in the article, which has proved valuable in a variety of circumstances.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*

General Terms: Algorithms

Additional Key Words and Phrases: Regular expressions, regular languages, SGML

1. INTRODUCTION

Regular expressions are widely regarded as a precise, succinct notation for specifying a text search, with a straightforward efficient implementation. Many people routinely use regular expressions to specify searches in text editors and with standalone search tools such as the Unix `grep` utility. Formally, a regular expression states a recognition problem: "Does a given string of text match a particular pattern?" However, searching is a different problem: "Locate the substrings of a text that match a particular pattern."

It is widely assumed that it is straightforward to reduce the search problem to the recognition problem while preserving the desirable properties of precision, succinctness and efficiency. This article illustrates difficulties with existing approaches that compromise these properties. We offer a new perspective on the use of regular expressions for searching text that preserves them. This perspective is particularly relevant to the search of highly structured text formatted with a markup language like SGML [Bryan 1988; ISO 1986].

This work was supported by the Information Technology Research Centre, Ontario and by the Natural Sciences and Engineering Research Council.
Authors' address: Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada; email: {claclark; gvcormac}@plg.uwaterloo.ca.

```
<act> <act-number> 1 </act-number>
<scene> <scene-number> 1 </scene-number>
  <direction> Thunder and lightning. Enter three Witches.  </direction>
  <speech> <speaker> First Witch </speaker>
     <line> When shall we three meet again? </line>
     <line> In thunder, lightning, or in rain? </line> </speech>
  <speech> <speaker> Second Witch </speaker>
      <line> When the hurly-burly's done, </line>
      <line> When the battle's lost and won. </line> </speech>
  <speech> <speaker> Third Witch </speaker>
      <line> That will be ere the set of sun. </line> </speech>
  <speech> <speaker> First Witch </speaker>
      <line> Where the place? </speech>
  <speech> <speaker> Second Witch </speaker>
      Upon the heath. </line> </speech>
```

Fig. 1. Excerpt from a structured text file.

We motivate our discussion with a simple example. Figure 1 presents a portion of a structured text file containing the Shakespearean play *Macbeth*, with the text formatted in the style of SGML. The start of each structural element is marked with a tag of the form <*name*>, and the end of a structural element is marked with a tag of the form </*name*>. The segment is taken from the play's opening scene, and it includes the act and scene numbers, stage directions, speakers, and their speeches. A representative search of such a text might be informally stated as "Locate speeches by witches that contain the words 'Dunsinane' or 'Birnam.'" Formulating this search using standard tools can prove quite challenging, particularly as the structural elements may be broken across multiple lines.

1.1 Regular Expressions

A regular expression r denotes a language $L(r)$, a set of strings composed of symbols from an alphabet Σ. Any regular language may be denoted by a regular expression built from five primitives defined as follows:

r	$L(r)$	
λ	$\{\text{“”}\}$	(empty string)
a	$\{\text{“a”}\}$	(alphabet symbol $a \in \Sigma$)
$r_1 \mid r_2$	$L(r_1) \cup L(r_2)$	(alternation)
$r_1 r_1$	$L(r_1) \circ L(r_2)$	(concatenation)
r^*	$L(r)^*$	(repetition)

where the concatenation of two languages $L_1 \circ L_2$ is defined as

$$L_1 \circ L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\},$$

and L^*, the closure of a language L, is defined as the smallest solution to

$$L^* = \{\text{“”}\} \cup (L \circ L^*).$$

While these five primitives are sufficient to describe any regular language, the five primitives alone do not yield a concise notation. In addition to these primitives, we use the operators &, −, and + defined as follows:

r	$L(r)$	
r_1 & r_2	$L(r_1) \cap L(r_2)$	(inclusion)
$r_1 - r_2$	$L(r_1) \setminus L(r_2)$	(exclusion)
r^+	$L(r) \circ L(r)^*$	(repetition at least once)

We use Σ as a regular expression denoting any symbol in the alphabet.

Regular expressions are easily recast in terms of the recognition problem: we say that a string x matches the regular expression r if $x \in L(r)$. For any regular expression, a finite automaton may be constructed that solves the recognition problem in $O(|x|)$ time where $|x|$ is the number of symbols in x. A single left-to-right scan is made over x. Storage requirements depend only on properties of r, not on the length of x. A regular expression matching a speech by a witch is

$$\texttt{<speech>}\ \Sigma^*\ \texttt{<speaker>}\ \Sigma^*\ \texttt{Witch}\ \Sigma^*\ \texttt{</speaker>}\ \Sigma^*\ \texttt{</speech>}$$

The fundamental results in the theory of regular languages and finite automata were developed in the 1950's and early 1960's. A standard account is given by Hopcroft and Ullman [1979]. This account includes an algorithm for the conversion of a regular expression to a nondeterministic finite automata (NFA) and a discussion of the closure properties of regular languages. The remainder of the article assumes basic familiarity with these results.

1.2 Substring Search

The search problem as stated at the beginning of the article is not precisely defined. A more precise definition might be "Given a universe U identify all elements of U that contain a substring x matching a particular pattern r." For searching text, a simple universe U is some a priori set of strings — file names in a directory, lines in a file, or documents in a collection. A simple search algorithm enumerates U in some order, reporting elements of U that contain matches to the pattern, until U is exhausted. This simple search strategy is used in Unix utilities like `sh` to search a universe of file names and `grep` to search a universe of lines from text files.

Requiring a priori that a search find a document, page, line, or word may yield a result too coarse or too fine to be useful. Further difficulties arise when there is no well-defined universe of possible solutions. This situation arises when the text is a continuous stream or is divided into units that are not suitable as search results. In our original example, it is clear that the desired universe is the set of all speeches; however, the text of Figure 1 is divided into lines only loosely related to the structure of the document, and most existing tools do not easily allow a search over arbitrarily defined units.

At its most general, the problem of searching a continuous stream of text with a regular expression may be characterized as follows: "Given a string x and a regular expression r, locate all substrings of x that match r." With respect to this characterization, the universe is the set of all substrings of x. Unfortunately, the cardinality of this set is quadratic in the length of x. Moreover, searching this universe may yield a plethora of overlapping and nested results. For this

reason, implementations attempting general search will restrict the search to find and report only a linear subset of the possible solutions. These arbitrary linearizing restrictions, which alter the semantics of the search, often appear simple, but are difficult to formalize, difficult to use precisely, and may be difficult to implement efficiently.

The most common restriction is the "leftmost longest match" rule. This is the rule mandated by the POSIX standard [IEEE 1992] and used in many software tools [Hargreaves and Berry 1992; Ousterhout 1994; Wall and Schwartz 1991]. The rule is carefully stated in the rationale to the POSIX standard:

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

Having said this, when performing a general search instead of looking for a single match, we are left with the question of what is the next match? There are two obvious choices: (1) begin the search again after the first character of the match or (2) begin the search again after the last character of the match. The first choice may result in a large number of nested solutions. The second of these choices is the one usually taken, but creates a bias for reporting the leftmost of overlapping matches. We refer to the technique of successively applying the leftmost longest-match rule, starting each time after the last character of the match, as "longest-match disjoint substring search." We say "disjoint" to indicate that the solutions may not overlap or nest. Using this rule, searching the *Macbeth* text file with the regular expression

$$\texttt{<speech>}\ \Sigma^*\ \texttt{<speaker>}\ \Sigma^*\ \texttt{Witch}\ \Sigma^*\ \texttt{<\speaker>}\ \Sigma^*\ \texttt{<\speech>}$$

results in a single match, starting at the fourth line of Figure 1 and continuing to the end of the last speech in the file. This clearly is not the intended result of this search. Furthermore, it is not obvious how to amend the regular expression to yield the intended result.

In the next two sections we propose an alternative linearizing restriction. By applying this restriction to a regular language we may search a text using a single left-to-right scan and constant storage. The restricted search is *most general* in that any solution to the general search will contain a solution to the restricted search. Our linearizing restriction may be characterized informally as

> Locate the set of shortest nonnested (but possibly overlapping) strings that each match the pattern.

Using this rule, which we term "shortest-match substring search," the result of searching *Macbeth* with the above regular expression will include all speeches by witches. It may also contain undesired matches. For example, an occurrence of the word "witch" in the body of a speech will result in a match that starts at the beginning of the current speech and continues to the end of the following speech. This problem is addressed in the fourth section by using regular expressions under shortest-match substring search to define universes for further search. The article concludes with a brief discussion of scanning and source code analysis and with a brief description of a new search tool based on shortest-match substring search.

The work described in this article was inspired by the authors' previous research in the area of structured text databases [Clarke et al. 1995a; Clarke et al. 1995b], where the linearizing restriction and its attendant properties proved key to the development of a closed search algebra with an efficient implementation. The success of the search algebra over indexed structured text suggested the application to regular-expression search over flat text.

1.3 Background and Related Work

Aho [1980; 1990] reviews algorithms for patterning matching in text, with particular emphasis on regular-expression search and practical experience gained with tools developed in conjunction with the Unix system. He discusses three possibilities for the output of a text search:

(1) If the target text is predivided into records, such as lines, the records containing the pattern may be reported.
(2) The longest-match disjoint substring rule may be used.
(3) A "yes" or "no" may be reported, indicating if the target text contains the pattern.

He assumes the last case for the purposes of his review.

Thompson [1968] gives a widely referenced construction for converting a regular expression to an NFA and provides an early description of the use of regular expressions for searching text. Thompson's algorithm reports each point in the target text where a match ends. A number of variants of Thompson's algorithm are described by Aho et al. [1974]. The algorithm described in Section 3.2 of this article is a extension of one of these variants.

An alternate to substring search is to report only one endpoint for each match to a regular expression r. Formally, Thompson's algorithm determines all m such that the text as a whole matches the regular expression $\Sigma^* r \Sigma^m$. For his Ph.D. thesis, Baeza-Yates [1989] formalized the problem as that of determining all m such that the text matches the regular expression $\Sigma^m r \Sigma^*$. Here, the values of m specify the locations in the text where a match begins.

At least two existing text-searching tools allow the user to explicitly define search universes, in one case by specifying an interrecord delimiter [Wu and Manber 1992a] and in the other by using longest-match disjoint substring search [Pike 1987a; Pike 1987b].

Considerable research has been devoted to addressing special cases of regular-expression search. Search algorithms have been developed for locating single keywords [Boyer and Moore 1977; Horspool 1980; Knuth et al. 1977], sets of keywords [Aho and Corasick 1975], keywords separated by sequences of "don't cares" [Abrahmson 1987; Fisher and Patterson 1974; Manber and Baeza-Yates 1991] and other simple patterns [Baeza-Yates and Gonnet 1992; Wu and Manber 1992b].

2. SHORTEST SUBSTRINGS

If L is a language over an alphabet Σ we define the language $G(L)$ as follows:

Definition 2.1. The string $x \in L$ is an element of $G(L)$ if and only if $\nexists y \in L$ such that y is a proper substring of x.

By *proper substring* we mean that y is a substring of x and that $y \neq x$. If L contains the empty string then $G(L) = \{\text{“”}\}$. For the remainder of the article we will assume that L does not contain the empty string.

We can now precisely state the search restriction advocated by this article: “Given a string x and a language L, locate all substrings of x that are members of $G(L)$.” If $x = a_1 a_2 ... a_n$, where the a_i are symbols from Σ, we can represent the result of such a search as a set of ordered pairs of integers, each giving a start and end position in x corresponding to an element in the result. We use the notation $x[u, v]$ with $u \leq v$ to indicate the substring of x starting at a_u and ending at a_v.

Definition 2.2. If $x \in \Sigma^*$ and L is a language over Σ then

$$\mathcal{G}(L, x) = \{(u, v) \mid x[u, v] \in G(L)\}.$$

For example,

$$\mathcal{G}(\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}, \text{“abracadabra”}) = \{(1, 2), (4, 5), (8, 9)\}.$$

Note that, although the string “abrac” is a member of the regular language denoted by $\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}$, it is not a member of $G(\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c})$, and the pair $(1, 5)$ is thus not included in the set. Several simple but significant properties of $\mathcal{G}(L, x)$ are immediately available.

THEOREM 2.3. *If $(u, v) \in \mathcal{G}(L, x)$ and $(u', v') \in \mathcal{G}(L, x)$ then either (1) $u < u'$ and $v < v'$, (2) $u > u'$ and $v > v'$, or (3) $u = u'$ and $v = v'$.*

PROOF. Shown by a straightforward case analysis. If $u \geq u'$ and $v < v'$, or if $u > u'$ and $v = v'$, then $x[u, v]$ is a proper substring of $x[u', v']$, and $x[u', v'] \notin G(L)$. If $u < u'$ and $v \geq v'$, or if $u = u'$ and $v > v'$, then $x[u', v']$ is a proper substring of $x[u, v]$, and $x[u, v] \notin G(L)$. □

THEOREM 2.4. *The elements of $\mathcal{G}(L, x)$ are totally ordered. The start and end points place identical total orders on the elements.*

PROOF. Immediate from Theorem 2.3. If $u < u'$ and $v < v'$ then $(u, v) < (u', v')$. If $u > u'$ and $v > v'$ then $(u, v) > (u', v')$. If $u = u'$ and $v = v'$ then $(u, v) = (u', v')$. □

As a further consequence of Theorem 2.3, the elements of $\mathcal{G}(L, x)$ do not nest but may overlap. That is, we may have $u < u' \leq v < v'$ or $u' < u \leq v' < v$, but not $u < u' \leq v' < v$ or $u' < u \leq v < v'$.

THEOREM 2.5. $|\mathcal{G}(L, x)| \leq n$.

PROOF. If $|\mathcal{G}(L, x)| > n$ then two distinct elements of $\mathcal{G}(L, x)$ must share a common start position. But by Theorem 2.3 their end positions would then be the same, in which case the elements could not in fact be distinct. □

The next theorem emphasizes the distinction between recognition and search. For classes of languages closed under concatenation (such as the regular languages), the problem of recognizing an element of L can be reduced to searching for elements of $G(L)$. For a class of such languages, if we have an algorithm that searches a text for elements of $G(L)$, then by the addition of start and end tokens, the same algorithm may be used to recognize members of L.

THEOREM 2.6. *If* ^ *and* $ *are not symbols in* Σ *then*

$$G(\{\text{“}\wedge\text{”}\} \circ L \circ \{\text{“}\ \$\ \text{”}\}) = \{\text{“}\wedge\text{”}\} \circ L \circ \{\text{“}\ \$\ \text{”}\}.$$

PROOF. Assume there exists $x = \hat{}w\$$ and $y = \hat{}z\$$ such that y is a proper substring of x, and x and y are elements of $\{\text{“}\wedge\text{”}\} \circ L \circ \{\text{“}\ \$\ \text{”}\}$. Since y is a proper substring of x, y must either be a substring of $\hat{}w$ or a substring of $w\$$. The first case implies that $ appears in w; the second implies that ^ appears in w. □

The members of L reported by a longest-match disjoint substring search are dependent on the text being searched. For example, a longest-match search for the regular expression $\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}$ in the string "ababab" results in three matches of the form ab. If we add a final "c" to the end of the string (making the string "abababc") the result changes to a single match of the form $\mathtt{a}\Sigma^*\mathtt{c}$. The string still contains three matches of the form ab, but these are no longer reported. In general, if $z \in L$ appears in the target text it is not possible to determine if a longest-match substring search will find a particular occurrence of z without reference to the entire text being searched. In contrast, over any text a shortest-match search reports all occurrences of the members of L that are in $G(L)$ and no others.

So far in this section of the article the exposition has not required that L be a regular language. The principle of shortest match may in fact be treated as a general principle for any text search application and has already proven itself for retrieval from indexed text [Clarke et al. 1995a; Clarke et al. 1995b]. However, we close the section by examining a specific property of regular languages.

THEOREM 2.7. *If* L *is a regular language then* $G(L)$ *is a regular language.*

PROOF. $G(L) = L \setminus ((L(\Sigma^+) \circ L \circ L(\Sigma^*)) \cup (L(\Sigma^*) \circ L \circ L(\Sigma^+)))$, which is regular by the various closure properties of regular languages. □

In a regular expression, we use the notation $[r]$, where r is a regular expression, to denote the language $G(L(r))$. For example, the expression $[\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}]$ denotes $G(L(\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}))$.

3. SUBSTRING SEARCH ALGORITHMS

In this section we compare algorithms for longest- and shortest-match substring search. A string may be recognized as member of a regular language by a single left-to-right scan with constant store. We argue informally that there is no longest-match search algorithm that shares this property. Any longest-match search algorithm using constant store can be forced to make multiple scans. In contrast, we present a simple algorithm for shortest-match substring search that makes a single left-to-right scan over the string and uses storage dependent only on the regular expression to be matched. In both cases we assume that matches are reported as they are discovered.

3.1 The Complexity of Longest-Match Disjoint Substring Search

Assume we have an algorithm that performs a longest-match search of a regular expression with a single scan of the target string using only constant store. Consider a longest-match search with the regular expression $\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}$ on a string of the form $(\mathtt{ab})^n\mathtt{d}$ for some fixed but arbitrary n. The string contains exactly n matches of

the form ab. Since a is the initial symbol in the string, the algorithm must make a full scan of the string, examining every character, to determine that this initial symbol is not part of a match of the form $\mathtt{a}\Sigma^*\mathtt{c}$. Since n may be arbitrarily large, no constant store may be used to maintain the potential matches that would be discovered while this determination is being made. It appears that our supposed algorithm cannot exist.

In this specific case, a longest-match search may be performed with two scans and constant store. In practice, longest-match search algorithms may make multiple scans of portions of their target strings, but this is not a serious problem as searches are generally restricted to a small segment of text.

3.2 Shortest-Match Substring Search

We detail an algorithm for shortest-match substring search for members of the regular language L over a text $x = a_1a_2...a_n$ of length n. We assume that an NFA M has been constructed to recognize L (perhaps from a regular expression). Let $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a set of states; Σ is an alphabet of symbols; δ is a state transition function mapping each element of $Q \times \Sigma$ onto a subset of Q; q_0 is the start state; and F is a set of final states. We are assuming that M has no λ-transitions for simplicity.

For the purposes of the algorithm, we assume that states are designated by numbers in the range 1 to $|Q|$ where the start state q_0 is assigned 1. The algorithm appears in Figure 2. The two integer arrays P and P' are indexed by state number with each element holding an index into x or the value 0. The symbols i and u are integer variables designating positions in the text, and the symbols j and q are integer variables designating states. The algorithm is a extension of a standard regular-expression search algorithm [Aho et al. 1974]. Correctness of the algorithm will be the subject of a later theorem.

Informally the algorithm operates as follows. The text is scanned from left to right (lines 3–20) with a new execution of the NFA beginning at the start state with each character scanned. For each state in the NFA, a position is recorded indicating the start of an interval of text that ends at the current character and that can leave the NFA in that state, if such an interval exists. The array P holds the current set of positions, with the array P' used for update purposes. If two intervals of text can leave the NFA in a particular state after the current character, then the shortest-substring rule requires that only the smallest of the two, with the largest start position, need be maintained by the algorithm. The use of the "max" function at lines 9 and 12 reflects this observation. Similarly, when a match is generated (line 14), only positions after the start of this match are kept (lines 15-16).

Storage requirements (except for the string itself) depend only on $|Q|$, the number of states of M. The outermost loop at lines 3–20 makes a single left-to-right scan over x. Now, let

$$T = \max_{1 \le j \le |Q|, a \in \Sigma} (|\delta(j, a)|).$$

T is the maximum cardinality of the transition function for any state and symbol. States have at most T transitions labeled with the same symbol. The loop at lines 8–9 makes at most T iterations for each iteration of the outer loop at lines 7–9. It

```
1        for j ← 1 to |Q| do
2            P_j ← 0;
3        for i ← 1 to n do begin
4            P_1 ← i;
5            for j ← 1 to |Q| do
6                P'_j ← 0;
7            for j ← 1 to |Q| do
8                for q ∈ δ(j, a_i) do
9                    P'_q ← max(P'_q, P_j);
10           u ← 0;
11           for j ← 1 to |Q| do
12               if j ∈ F then u ← max(u, P'_j);
13           if u > 0 then begin
14               Output (u, i);
15               for j ← 1 to |Q| do
16                   if P'_j ≤ u then P'_j ← 0;
17           end;
18           for j ← 1 to |Q| do
19               P_j ← P'_j;
20       end;
```

Fig. 2. Shortest-match substring search algorithm.

is apparent from the structure of the loops that the algorithm has worst-case time complexity of $O(|Q|Tn)$.

That the algorithm correctly performs a shortest-match substring search is the subject of the next theorem. As a final note, in an actual implementation of the algorithm, the arrays P and P' are more efficiently represented as lists of states and positions. States for which an array element would be 0 are omitted from this list.

THEOREM 3.2.1. *A pair (u, v) is output by the algorithm of Figure 2 if and only if $(u, v) \in \mathcal{G}(L, x)$.*

PROOF. We begin by establishing invariants for the array P over the loop at lines 3–20. At any point in the execution of the algorithm let t be the first element of the previous pair output or 0 if no pair has been output. When a pair is output on line 14, t is effectively updated. The invariants are (1) if $P_j \neq 0$, then j is not a final state, and the string $x[P_j, i]$ is the shortest suffix of $x[t+1, i]$ for which there is a path in M from the start state to state j and (2) if $P_j = 0$, there is no suffix of $x[t+1, i]$ for which there is a path in M from the start state to j.

Within the body of the loop at lines 3–20, the array P' is used to compute the updated value of P based on the previous value of P. Lines 18–19 effect the update.

The invariants for P' over the loop at lines 7–9 are as follows: (1) if $P'_k \neq 0$, then $x[P'_k, i]$ is the shortest suffix of $x[t+1, i]$ for which there is a path in M from the start state to k where the last transition is from a state numbered j or lower and (2) if $P'_k = 0$, then there is no suffix of $x[t+1, i]$ for which there is a path in M from the start state to k where the last transition is from a state numbered j or lower. Thus, after line 9, if $P'_j \neq 0$ then $x[P'_j, i]$ is the smallest suffix of $x[t+1, i]$

for which there is a path in M from the start state to j, and if $P'_j = 0$ then there is no suffix of $x[t+1, i]$ for which there is a path in M from the start state to j.

After line 9, there may be final state for which $P'_j \neq 0$. If this is the case, the loop on lines 10–12 discovers the largest u such that $x[u, i]$ is an element of L, thus $x[u, i]$ is an element of $G(L)$. The lines 13–17 output (u, i) (implicitly setting $t \leftarrow u$) and invalidate all partial or complete matches starting at or before u by setting the appropriate elements of P' to 0. This implies that after line 17, $P'_j = 0$ if j is a final state.

If (u, v) is a element of $G(L)$ it will be the shortest suffix of $x[t+1, i]$ for some t and will be output at line 14. □

4. EXPLICIT CONTAINMENT

A regular expression may be used to define an explicit universe for search. Using the search restriction advocated by this article the regular expression

$$[\ \texttt{<speech>}\ \Sigma^*\ \texttt{</speech>}\ ]$$

defines the universe of speeches.

We define a new operator "containing" ($\triangleright$) to express a search over an explicit universe. The regular expression $r \triangleright s$, where r and s are regular expressions, is defined as $[r]\ \&\ (\Sigma^* s \Sigma^*)$.

Consider our original search statement: "Find speeches by witches that contain the words 'Dunsinane' or 'Birnam.'" Using the containing operator and assuming our search restriction we may formulate this search as

$$(\texttt{<speech>}\ \Sigma^*\ \texttt{</speech>})\ \triangleright$$
$$(\texttt{<speaker>}\ \Sigma^*\ \texttt{Witch}\ \Sigma^*\ \texttt{</speaker>}\ \Sigma^*\ (\texttt{Birnam} \mid \texttt{Dunsinane}))$$

To this point we have not discussed the conversion of a regular expression to an NFA, but some explanation is required in connection with explicit containment. The conversion may be accomplished by any of the standard techniques [Brzozowski 1964; Berry and Sethi 1986; McNauthton and Yamada 1960; Thompson 1968], remembering that the algorithm of Figure 2 assumes transitions are labeled with a single symbol. Of some concern is the size of the NFA that may result from this conversion. The size of an NFA generally grows additively for alternation, concatenation, and the varieties of repetition; for inclusion it grows multiplicatively. For most applications this growth is not a problem. Unfortunately, in order to implement exclusion, the NFA must be converted to a DFA, with a possible exponential increase in size. It is then not reasonable to implement the "containing" operator by directly using the equation in the proof of Theorem 2.7.

Fortunately, explicit containment may be implemented without direct use of that equation. We first observe that if an interval of the text contains a solution to the regular expression s, then it contains a solution to $[s]$. It then follows that the regular expression $r \triangleright s$ is equivalent to $[r] \triangleright [s]$:

$$r \triangleright s \quad = \quad [r]\ \&\ (\Sigma^* s \Sigma^*) \quad = \quad [r]\ \&\ (\Sigma^* [s] \Sigma^*) \quad = \quad [r] \triangleright [s]$$

It is thus possible to implement explicit containment by running two concurrent copies of the algorithm of Figure 2, one to locate occurrences of $[r]$ and one to locate occurrences of $[s]$. When a match to $[r]$ is located, the location of the last match

to $[s]$ is checked. If that match to $[s]$ is contained within the bounds of the match to $[r]$, the match to $[r]$ is a solution to $r \triangleright s$. This technique is in fact a simple case of a more general algorithm [Clarke et al. 1995a].

5. SCANNING, SEARCHING, AND SOURCE CODE ANALYSIS

In the same year that Thompson published his regular-expression search algorithm, a group at MIT used regular languages for automatically constructing lexical analyzers [Johnson et al. 1968]. This system used the longest-match rule to resolve simple cases of matching ambiguity and reported errors in others. Since then, regular expressions, interpreted under the longest-match rule, have been widely used to specify scanners for programming languages and other translators [Aho et al. 1986; Lesk 1975]. Scanning is a somewhat different problem than searching, requiring a partitioning of the input into tokens, essentially performing a transduction of the text. The leftmost longest-match rule appears to be a more natural choice for scanning. Consider the typical definition of an identifier in a programming language: an alphabetic character followed by zero or more alphanumeric characters. This definition must be interpreted under a longest-match rule to be correct.

However, Theorem 2.6 provides a hint on how to use shortest-substring search to locate objects, such as identifiers, that have a longest-match component implicit in their definitions. Instead of directly using the definition given earlier, we might specify an identifier as a nonalphanumeric character, followed by an alphabetic character, followed by zero or more alphanumeric characters, followed by a nonalphanumeric character. In the search results, a surrounding pair of nonalphanumeric characters will be reported with each identifier.

Feature extraction from source code may be viewed as a specialized search problem to which regular-expression search may be applied. For example, Murphy and Notkin [1995] used a pattern-matching language based on regular expressions to assist in the construction of call graphs, cross reference lists, and similar code analysis output. Some features of program source code, comments in particular, may be more easily identified using a shortest-match rule. Under some mild restrictions, the regular expression

$$/*\Sigma^**/$$

will match all comments in a group of C source files.

Global search-and-replace operations in text editors provide a further example of a situation in which overlap among matches may not be considered appropriate. Like scanning, global search-and-replace is a transduction operation. Nonoverlapping substrings appear to be desirable for this application, since the mapping from input text to output text is clearly defined. Unfortunately, use of the leftmost longest-match rule makes the arbitrary restriction of applying the edit only to the leftmost of several overlapping intervals. The extension of the shortest-substring rule to transduction is an area for future research.

6. THE CGREP SEARCH TOOL

The theory described in this article has been used as the basis for a search tool that has proved to be of significant value in a number of applications. In searching and organizing folders of mail messages and news articles, the tool allows the selection

and extraction of articles based on combinations of the contents of the header lines and patterns occurring in the body. The tool has been used by one of the authors to assist in sorting, organizing, and indexing structured documents from the TREC collection [Harman 1993]. Other applications have included the extraction of World Wide Web hypertext links (URLs) from NetNews articles, searching files of machine code in binary format, searching comments in source code files, and counting actual occurrences of patterns in files (as opposed to lines containing the patterns). Formulating equivalent operations with existing text-processing tools has proven difficult and frustrating. The tool is made available through the MultiText project repository (`ftp://plg.uwaterloo.ca/pub/mt/cgrep`). A technical report distributed with the tool [Clarke and Cormack 1996] discusses the details of these examples, discusses performance, and provides a comparison with more specialized search tools.

7. CONCLUDING COMMENTS

This article formally defines and explores the properties of a "shortest-match" search rule for regular expressions and describes practical results concerning the application of regular languages to search. The shortest-substring rule provides a precise definition of which strings will be selected during a search without any dependence on the contents of the remainder of the text. Shortest substring search can be performed with a single left-to-right scan over the text with storage requirements dependent only on properties of the regular expression.

The motivating application for our approach is in searching structured text that is not divided into predefined retrieval units. The use of the shortest-substring rule allows regular expressions to be used to define explicit search universes, which then become the subject of further selection through regular-expression search. The necessity of an a priori division of the text into search records is thereby avoided.

The extension of the shortest-substring rule to transduction is an area for future research. The shortest-substring restriction may also be useful as a modification to existing search algorithms, such as those for approximate regular-expression search [Knight and Myers 1995; Myers and Miller 1989], or to extend other recognition algorithms to the search problem [Chang and Paige 1992]. An independently developed pattern-matching tool, `sgrep` [Jaakkola and Kilpeläinen 1995], is partially based on earlier work by the present authors [Clarke et al. 1995a] and incorporates some of the ideas discussed in this article.

ACKNOWLEDGEMENTS

Charlie Krasic, Dave Mason, Dave Shewchun, and Gord Vreugdenhil all made useful comments when the ideas of this article were in nascent form. We thank the anonymous referees for their comments and the early users of the `cgrep` search tool for their feedback.

REFERENCES

Abrahmson, K. 1987. Generalized string matching. *SIAM J. Comput. 16*, 1039–1051.

Aho, A. V. 1980. Pattern matching in strings. In *Formal Language Theory — Perspectives and Open Problems*, R. V. Book, Ed. Academic Press, New York, 325–344.

Aho, A. V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. MIT Press, Cambridge, Mass., 256–300.

Aho, A. V. and Corasick, M. J. 1975. Efficient string matching — An aid to bibliographic search. *Commun. ACM 18,* 6 (June), 333–340.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Baeza-Yates, R. A. 1989. Efficient text searching. Ph.D. thesis, Univ. of Waterloo, Waterloo, Canada.

Baeza-Yates, R. A. and Gonnet, G. H. 1992. A new approach to text searching. *Commun. ACM 35,* 10 (Oct.), 74–82.

Berry, G. and Sethi, R. 1986. From regular expressions to deterministic automata. *Theor. Comput. Sci. 48*, 117–126.

Boyer, R. S. and Moore, J. S. 1977. A fast string searching algorithm. *Commun. ACM 20,* 10 (Oct.), 762–772.

Bryan, M. 1988. *SGML — An Author's Guide to the Standard Generalized Markup Language.* Addison-Wesley, Reading, Mass.

Brzozowski, J. A. 1964. Derivatives of regular expressions. *J. ACM 11,* 4, 481–494.

Chang, C.-H. and Paige, R. 1992. From regular expressions to DFA's using compressed NFA's. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 644. Springer-Verlag, Berlin, 90–109.

Clarke, C. L. A. and Cormack, G. V. 1996. Context grep. Tech. Rep. CS-96-41, Computer Science Dept., Univ. of Waterloo, Waterloo, Canada.

Clarke, C. L. A., Cormack, G. V., and Burkowski, F. J. 1995a. An algebra for structured text search and a framework for its implementation. *Comput. J. 38,* 1, 43–56.

Clarke, C. L. A., Cormack, G. V., and Burkowski, F. J. 1995b. Schema-independent retrieval from hetrogeneous structured text. In *Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval.* UNLV, Las Vegas, Nev.

Fisher, M. and Patterson, M. 1974. String matching and other products. In *Proceedings of the SIAM-AMS Conference on Complexity of Computation*, R. M. Karp, Ed. American Mathematical Society, Providence, R.I., 113–125.

Hargreaves, K. A. and Berry, K. 1992. *Regex.* Free Software Foundation, Cambridge, Mass. Available from ftp://prep.ai.mit.edu/pub/gnu.

Harman, D. 1993. Overview of the first TREC conference. In *Proceedings of the 16th Annual International ACM SIGIR Conference.* ACM Press, New York, 36–47.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass.

Horspool, R. N. 1980. Practical fast searching in strings. *Softw. Pract. Exper. 10*, 501–506.

IEEE. 1992. *Standard for information technology — Portable Operating System Interface* (*POSIX*) *— Part 2* (*Shell and utilities*) *— Section 2.8* (*Regular expression notation*). IEEE Std 1003.2, Institute of Electrical and Electronics Engineers, New York.

ISO. 1986. *Information processing — Text and office systems — Standard Generalized Markup Language* (*SGML*). ISO 8879, International Standards Organization, Geneva, Switzerland.

Jaakkola, J. and Kilpeläinen, P. 1995. Sgrep home page. Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland. Available as http://www.cs.helsinki.fi/ jjaakkol/sgrep.html.

Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. 1968. Automatic generation of efficient lexical processors using finite state techniques. *Commun. ACM 11,* 12 (Dec.), 805–813.

Knight, J. R. and Myers, E. W. 1995. Super-pattern matching. *Algorithmica 13*, 211–243.

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

Lesk, M. E. 1975. Lex — A lexical analyzer generator. Computing Science Tech. Rep. 39, AT&T Bell Laboratories, Murray Hill, N.J.

Manber, U. and Baeza-Yates, R. 1991. An algorithm for string matching with a sequence of don't cares. *Inf. Process. Lett. 37*, 133–136.

McNauthton, R. and Yamada, H. 1960. Regular expressions and state graphs for automata. *IRE Trans. Electronic Comput. EC-9,* 1 (Mar.), 39–47.

Murphy, G. C. and Notkin, D. 1995. Lightweight source model extraction. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE '95*). ACM Press, New York.

Myers, E. W. and Miller, W. 1989. Approximate matching of regular expressions. *Bull. Math. Biol. 51,* 1, 5–37.

Ousterhout, J. K. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass.

Pike, R. 1987a. Structural regular expressions. In *Proceedings of the European UNIX User's Group Conference*. EUUG, Helsinki, Finland.

Pike, R. 1987b. The text editor sam. *Softw. Pract. Exper. 17,* 11 (Nov.), 813–845.

Thompson, K. 1968. Regular expression search algorithm. *Commun. ACM 11,* 6 (June), 419–422.

Wall, L. and Schwartz, R. L. 1991. *Programming Perl*. O'Reilly and Associates, Sebastopol, Calif.

Wu, S. and Manber, U. 1992a. agrep — A fast approximate pattern matching algorithm. In *Proceedings of the USENIX Winter 1992 Technical Conference*. USENIX Assoc., Berkeley, Calif, 153–162.

Wu, S. and Manber, U. 1992b. Fast text searching allowing errors. *Commun. ACM 35,* 10 (Oct.), 83–91.

Received July 1995; revised September 1996; accepted October 1996