On the use of Regular Expressions for Searching Text

Charles L. A. Clarke Gordon V. Cormack
Department of Computer Science
University of Waterloo, Waterloo, Canada

Technical Report CS-95-07
February, 15, 1995

Abstract

The use of regular expressions to search text is well known and understood as a useful technique. It is then surprising that the standard techniques and tools prove to be of limited use for searching text formatted with SGML or other similar markup languages. Experience with structured text search has caused us to carefully re-examine the current practice. The generally accepted rule of "left-most longest match" is an unfortunate choice and is at the root of the difficulties. We instead propose a rule which is semantically cleaner and is incidentally more simple and efficient to implement. This rule is generally applicable to any text search application.

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 stand-alone search tools such as the Unix **grep** utility. A regular expression states a recognition problem: "Does a given string of text match a particular pattern?" However, searching is a different problem: "Find the substrings of a text that match a particular pattern."

It is widely assumed that it is trivial to reduce the search problem to the recognition problem while preserving the desirable properties of precision, succinctness and efficiency. This paper 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 [9, 17].

We motivate our discussion with a simple example. Figure 1 presents a portion of a structured text file containing the Shakespearean play *Macbeth*. The text is formatted in the style of SGML. The start of a 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 includes the act and scene numbers, stage directions, speakers and their speeches. A typical search of such text might be informally stated as: "Find 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.

```
<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>
```

Figure 1: Excerpt from a structured text file

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	$\{``"\}$	(empty string)
a	$\{``a"\}$	(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

$$L^* = \{``"\} \cup (L \circ L^*)$$

While these five primitives are sufficient to describe any regular language, the five primitives alone do not yield a concise notation. For example, consider 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)

Regular languages are known to be closed under intersection and set difference, yet the expressions r_1 & r_2 and $r_1 - r_2$ are not easily represented using the five primitives, and the representation may be exponentially larger than r_1 and r_2. Even simpler constructions cause exponential growth in regular expressions: r^+ is trivially defined in terms of r^* yet an equivalent expression using only the five primitives may be exponentially larger than one using $+$. Throughout this exposition we use these and other operators, defined as necessary, to present succinctly our approach. As an immediate example, a regular expression denoting all of the symbols in the alphabet may be easily constructed using the alternation operation and the symbols in the alphabet, but it is far simpler to use Σ itself to denote this regular expression.

Regular expressions are easily re-cast 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 that contains the words "Dunsinane" or "Birnam" is:

`<speech>` Σ^* `<speaker>` Σ^* `Witch` `<\speaker>` Σ^* (`Birnam` | `Dunsinane`) Σ^* `<\speech>`

1.2 Text Search

The search problem as stated at the beginning of the paper is not precisely defined. A more precise definition might be: "Given a universe U find all elements of U that contain a substring x matching a particular pattern r." Even this statement is not as precise as we would like, because it may not be possible or desirable to find all solutions. Reducing search to recognition involves two arbitrary choices: the definition of a universe or search space, and the selection of a particular solution or set of solutions. In existing applications these arbitrary choices are seldom stated at all, let alone stated formally.

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. The time required for this search technique is $O(\Sigma_{x \in U} |x|)$. That is, the time to search a text database is at worst proportional to the number of symbols in the database. 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 database is a continuous stream or is divided into units that are not suitable as search results. In the earlier example, it is clear that the most desirable universe is the set of all speeches, but the text is divided into lines only loosely related to the structure of the document, and existing tools do not easily allow a search over arbitrarily defined units.

At its most general, the problem of scanning and searching a continuous stream of text with a regular expression may be characterized as follows: "Given a string x and a regular expression r, find 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. Searching this universe requires quadratic time and may yield a plethora of overlapping and nested results. For this reason, implementations attempting general search restrict the search to find and report only some subset of the solutions. These arbitrary restrictions, which alter the semantics of the search, often appear simple, but are difficult to formalize, difficult to use precisely, and difficult to implement efficiently.

The most common restriction is the "left-most longest match" rule. This is the rule mandated by the POSIX standard [16] and used in many software tools [13]. 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, rather than 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 second of these choices is the one usually taken, as the first choice may result in a large number of nested solutions. We refer to the technique of successively applying the left-most 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

`<speech>` Σ^* `<speaker>` Σ^* `Witch` Σ^* `<\speaker>` Σ^* `<\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 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 section we propose an alternative linearizing restriction. By applying this restriction to a regular language we may search a text using a single right-to-left 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: "Find the set of shortest non-nested (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; this problem is addressed in the penultimate section.

1.3 Historical Notes

The basic results in the theory of regular languages and finite automata were developed in the 1950's and early 1960's. A standard account of these results and their development is given by Hopcroft and Ullman [14]. This account includes an algorithm for the conversion of a regular expression to a nondeterministic finite automata and a discussion of the closure properties of regular languages.

In 1968, Thompson described the use of regular expressions for searching text [22]. Thompson's algorithm reports each point in the target text where a match ends. In the same year, a group at MIT used regular languages for automatically constructing lexical analyzers [18]. This system used the longest-match rule to resolve simple cases of matching ambiguity and reported errors in others. A number of variants of Thompson's algorithm are described by Aho, Hopcroft and Ullman [5]. The algorithm described in section 3.2 of this paper is a extension of one of the variants proposed in their discussion. A general review of algorithms for searching text, with particular emphasis on the practical experience gained with tools developed in conjunction with the Unix system, is given by Aho [2]. More recently, a general algorithm for regular expression search in preprocessed text with average-case time complexity of $O(\sqrt{n})$ has been developed by Baeza-Yates and Gonnet [6].

Much effort has been devoted to addressing special cases of regular expression search. Search algorithms have been developed for finding single keywords [8, 15, 19], sets of keywords [4], keywords separated by sequences of "don't cares" [1, 12, 20], and other simple patterns [7, 23]. A recent general survey of the area is given by Aho [3].

2 Shortest Substrings

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

Definition 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 $y \neq x$. If L contains the empty string then $G(L) = \{``"\}$. For the remainder of the paper we will assume that L does not contain the empty string.

We can now precisely state the search restriction avocated by this paper: "Given a string x and a language L, find 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 *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,

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

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

Theorem 1 *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: 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 $x[u, v] \not\in \mathcal{G}(L, x)$. 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 $x[u', v'] \not\in \mathcal{G}(L, x)$. □

The elements of $\mathcal{G}(L, x)$ are thus totally ordered. The start and end points place identical total orders on the elements (if $u < u'$ and $v < v'$ then $(u, v) < (u', v')$). 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 $u' < u \leq v < v'$.

As a further consequence of Theorem 1 we have $|\mathcal{G}(L, x)| \leq n$. For if $|\mathcal{G}(L, x)| > n$ then two distinct elements of $\mathcal{G}(L, x)$ must share a common start position. But by Theorem 1 their end positions would then be the same, in which case the elements could not in fact be distinct.

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 *If ^ and \$ are not symbols in Σ then* $G(\{``\wedge"\} \circ L \circ \{``\,\$\,"\}) = \{``\wedge"\} \circ L \circ \{``\,\$\,"\}$.

Proof: Assume there exists $x =\hat{}w\$$ and $y =\hat{}z\$$, elements of $\{``\wedge"\} \circ L \circ \{``\,\$\,"\}$ such that y is a proper substring of x. 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 $\texttt{ab} \mid \texttt{a}\Sigma^*\texttt{c}$ 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 "ababc") the result changes to a single match of the form $\texttt{a}\Sigma^*\texttt{c}$. 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 paper 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 [11, 10]. However, we close the section by examining a specific property of regular languages.

Theorem 3 *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 $[\texttt{ab} \mid \texttt{a}\Sigma^*\texttt{c}]$ denotes $G(L(\texttt{ab} \mid \texttt{a}\Sigma^*\texttt{c}))$.

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 demonstrate 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 and consequently no storage is required for the matches.

3.1 The Complexity of Longest-Match Disjoint Substring Search

Suppose 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 ab | $a\Sigma^*c$ on a string of the form $(ab)^n d$ 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 $a\Sigma^*c$. 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 do make multiple scans of portions of their target strings, but this is not a serious problem as searches are generally restricted to a single line 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 $x = a_1 a_2 ... a_n$ 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, j, q and u designate integer variables.

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. Noting that the loop at lines 8–9 makes at most $|Q|$ iterations for each iteration of the outer loop at lines 7–9, it is apparent from the structure of the loops that the algorithm has worst-case time complexity of $O(|Q|^2 n)$. 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 4 *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 smallest suffix of $x[t+1, i]$ which specifies a path in M from the start state to state j. 2) If $P_j = 0$, there is no suffix of x[t+1,i] which specifies a path in M from the start state to j, implying that no substring of x[t+1,i] is an element of $G(L)$.

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 smallest suffix of $x[t+1, i]$ which specifies a path in M from the start state to k where the last

```
for i ← 1 to |Q| do
    P_i ← 0;
for i ← 1 to n do begin
    P_1 ← i;
    for j ← 1 to |Q| do
        P'_j ← 0;
    for j ← 1 to |Q| do
        for q ∈ δ(j, a_i) do
            P'_q ← max(P'_q, P_j);
    u ← 0;
    for j ← 1 to |Q| do
        if j ∈ F then u ← max(u, P'_j);
    if u > 0  then begin
        Output (u, i);
        for j ← 1 to |Q| do
            if P'_j ≤ u then P'_j ← 0;
    end;
    for j ← 1 to |Q| do
        P_j ← P'_j;
end;
```

Figure 2: Shortest-match substring search algorithm

transition is from a state numbered j or lower. 2) If $P'_k = 0$, then there is no 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 $x[t+1, i]$ that specifies a path in M from the start state to j, and if $P'_j = 0$ then there is no suffix of $x[t+1, u]$ that specifies 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 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 paper the regular expression

[`<speech>` Σ^* `<\speech>`]

defines the universe of speeches.

We define a new operator "containing" (`>>`) to express a search over an explicit universe. The regular expression r `>>` 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:

(`<speech>` Σ^* `<\speech>`) `>>` (`<speaker>` Σ^* `Witch` `<\speaker>` Σ^* (`Birnam` | `Dunsinane`))

To this point we have not discussed the issue of converting a regular expresion to an NFA, but some explanation is required in connection with explicit containment. In practice, we accomplish the conversion using a variant of a well-known technique [5, 22]. Generally, the differences being that we construct an NFA with no λ-transitions and mantain the NFA as a list of these transitions. Of some concern is the size of the NFA that will result from this conversion. The size of an NFA 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 3.

Fortunately, explicit containment may be implemented without direct use of that equation. We observe that the regular expression r `>>` $[s]$ is equilvalent to r `>>` s. It is thus possible to implement explicit containment by running two concurrent copies of the algorithm of figure 2, reporting a match to r only when it contains a match to s. This technique is in fact a simple case of the more general algorithm described in [11].

5 Conclusion

Despite surface appearances, this paper does not describe theoretical results concerning regular languages. Instead, this paper describes practical results concerning the application of regular languages to text searching. We demonstrate that the theory of regular languages has perhaps been misapplied to text searching. We draw general lessons concerning the extension of language recognition to search.

Acknowledgements

Charlie Krasic, Dave Mason and Gord Vreugdenhil made useful comments when the ideas of this paper were in nascent form. The Government of the Province of Ontario provided primary funding for this work through its Information Technology Research Centre. Additional funding was provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] K. Abrahmson. Generalized string matching. *SIAM Journal on Computing*, 16:1039–1051, 1987.

[2] Alfred V. Aho. Pattern matching in strings. In Ronald V. Book, editor, *Formal Language Theory — Perspectives and Open Problems*, pages 325–344. Academic Press, New York, 1980.

[3] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 256–300. The MIT Press/Elsevier, Cambridge/Amsterdam, 1990.

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

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[6] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. In *Proceedings 16th International Coloquium on Automata, Languages and Programming*, pages 46–62, Stresa, Italy, 1989.

[7] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[8] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

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

[10] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Schema-independent retrieval from hetrogeneous structured text. In *Fourth Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, Nevada, April 1995. An early version of this paper was distributed as University of Waterloo Computer Science Department Technical Report number CS-94-39 [21].

[11] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 1995. To appear. An early version of this paper was distributed as University of Waterloo Computer Science Department Technical Report number CS-94-30 [21].

[12] M.J. Fisher and M. Patterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings)*, volume 7, pages 113–125. American Mathematical Society, Providence, Rhode Island, 1974.

[13] Kathryn A. Hargreaves and Karl Berry. *Regex.* Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, 1992. `ftp://prep.ai.mit.edu/pub/gnu`.

[14] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[15] R. Nigel Horspool. Practical fast searching in strings. *Software — Practice and Experience*, 10:501–506, 1980.

[16] Institute of Electrical and Electronics Engineers. *Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and Utilities) — Section 2.8 (Regular Expression Notation)*, September 1992. IEEE Std 1003.2.

[17] International Standards Organization. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, October 1986. ISO 8879.

[18] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, December 1968.

[19] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[20] Udi Manber and Ricardo Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, February 1991.

[21] The MultiText Project. Project repository: `ftp://plg.uwaterloo.ca/pub/mt`.

[22] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[23] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.