

Fast Text Searching for Regular Expressions or Automaton Searching on Tries

RICARDO A. BAEZA-YATES

University of Chile, Santiago, Chile

AND

GASTON H. GONNET

Informatik, Zurich, Switzerland

Abstract. We present algorithms for efficient searching of regular expressions on preprocessed text, using a Patricia tree as a logical model for the index. We obtain searching algorithms that run in logarithmic expected time in the size of the text for a wide subclass of regular expressions, and in sublinear expected time for any regular expression. This is the first such algorithm to be found with this complexity.

Categories and Subject Descriptors: E.1 [Data Structures]; F.2 [Analysis of Algorithms and Problem Complexity]; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval; I.5.4 [Pattern Recognition]: Applications—*text processing*

General Terms: Algorithms

Additional Key Words and Phrases: Digital trees, finite automata, regular expressions, text searching

1. Introduction

Pattern matching and text searching are very important components of many problems, including text editing, data retrieval and symbol manipulation. The problem consists in finding occurrences of a given pattern in a long string (the text).

Important variations of pattern matching are given by preprocessing or not preprocessing the text (building an index) and by the language used to specify the query. In this paper we are interested in the preprocessing case when the query is specified by a regular expression. Formally, given the text string t and a regular

The work of R. A. Baeza-Yates was partially funded by the University of Waterloo, the Institute for Computer Research, the province of Ontario, and Chilean Fondecyt Grant No. 95-0622.

Authors' present addresses: R. A. Baeza-Yates, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile, e-mail: rbaeza@dcc.uchile.cl; G. H. Gonnet, Informatik, ETH, Zurich, Switzerland.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1996 ACM 0004-5411/96/1100-0915 \$03.50

expression q (the query), the problem consists of finding whether $t \in \Sigma^*q\Sigma^*$ (q for short) and obtaining some or all of the following information:

- (1) The location where an occurrence (or specifically the first, the longest, etc.) of q exists. Formally, if $t \in \Sigma^*q\Sigma^*$, find a position $m \geq 0$ such that $t \in \Sigma^mq\Sigma^*$. For example, the first occurrence is defined as the least m that fulfills this condition.
- (2) The number of occurrences of the pattern in the text. Formally, the number of all possible values of m in the previous category.
- (3) All the locations where the pattern occurs (the set of all possible values of m).

In general, these problems do not have the same complexity. As we are interested in starting positions for a match, it is enough to find the shortest string that matches q in a given position. We assume that ϵ is not a member of $L(q)$. If it is, the answer is trivial; all values for m are possible.

The traditional approach for searching text for matches of a regular expression is to use the finite automaton that recognizes the language defined by the regular expression [Thompson 1968] with the text as input. The main problems with this approach are:

- in general, all the text must be processed, giving an algorithm with running time linear in the size of the text [Rivest 1977]. For many applications, this is unacceptable.
- if the automaton is deterministic, both the construction time and number of states can be exponential in the size of the regular expression [Hopcroft and Ullman 1979]. This is not a crucial problem in practice, because the size of the query is typically small and therefore can be treated as if bounded by a constant.

Although there have been many results for searching strings, sets of strings, patterns with don't cares or classes of symbols (see Gonnet and Baeza-Yates [1991, chapter 7]), no new results for the time complexity of searching for regular expressions have been published.

In this paper, we present the first algorithms found which achieve sublinear expected time to search any regular expression in a text and constant or logarithmic expected time for some restricted regular expressions. The main idea behind these algorithms is the simulation of the finite automaton of the query over a digital tree (or Patricia tree) of the text (our index). We run the automaton on all paths of the digital tree from the root to the leaves, stopping when possible. The time savings come from the fact that each edge of the tree is traversed at most once, and that every edge represents pairs of symbols in many places of the text. We present the exact average case analysis for this simulation, which is also interesting in its own right.

The paper is organized as follows: Section 2 gives the background necessary for the rest of the paper. Sections 3 and 4 present algorithms for restricted classes of regular expressions. Section 5 gives the main result of the paper: an algorithm to search any regular expression, with its expected time analysis. Section 6 gives a heuristic to optimize generic pattern matching queries, in particular when given as regular expressions. A preliminary version of this paper

was presented in Baeza-Yates and Gonnet [1989a] and these results are part of Baeza-Yates [1989].

2. Preliminaries

2.1. NOTATION. We use Σ to denote the *alphabet* (a set of symbols). A *string* over an alphabet Σ is a finite length sequence of symbols from Σ . The *empty string* (ϵ) is the string with no symbols. If x and y are strings, xy denotes the *concatenation* of x and y . If $w = xyz$ is a string, then x is a *prefix*, and z a *suffix* of w . The *length* of a string x ($|x|$) is the number of symbols in x . Any contiguous sequence of letters y from a string w is called a *substring*.

We use the usual definition of regular expressions (RE for short) defined by the operations of concatenation, union (+) and star or Kleene closure (*) [Hopcroft and Ullman 1979]. We also use:

— Σ to denote any symbol from Σ (when the ambiguity is clearly resolvable by context).

— $r?$ to denote zero or one occurrence of r (that is, $r? = \epsilon + r$).

— $r^{\leq k}$ to denote $\bigcup_{i=0}^k r^i$ (finite closure).

We use regular languages as our query domain, and regular languages can be represented by regular expressions. Our main motivation has been the work done as part of the New Oxford English Dictionary (OED) project at the University of Waterloo, where the main problem was the size of the dictionary. In this context, many interesting queries may be expressed as regular expressions. Some example queries from the OED are given in Appendix A.

Regular languages are accepted by deterministic (DFA) or nondeterministic (NFA) finite automata. We use the standard definition of finite automata [Hopcroft and Ullman 1979]. There is a simple algorithm [Aho et al. 1974] that, given a regular expression r , constructs a NFA that accepts $L(r)$ in $O(|r|)$ time and space. There are also algorithms to convert a NFA to a DFA without ϵ transitions ($O(|r|^2)$ states) and to a DFA ($O(2^{|r|})$ states in the worst case) [Hopcroft and Ullman 1979].

2.2. TEXT DATABASES. Text databases can be classified according to their update frequency as *static* or *dynamic*. Static text does not suffer changes, or its update frequency is very low. Static databases may have new data added and can still be considered static. Examples of static text are historical data (no changes) and dictionaries, which is the case we are interested in.

For this type of database, it is worthwhile to *preprocess* the text and to *build indices* or other data structures to speed up the query time. The preprocessing time will be amortized during the time the text is searched before the data changes again. One possible characterization of static databases is that updating the index can be done at discrete times which are much larger than query response time (for example, once a day or once a year). This is the case when searching the Oxford English Dictionary.

On the other hand, dynamic text is text that changes too frequently to justify preprocessing, for example, in text-editing applications. In this case we must use search algorithms that scan the text sequentially, or that have efficient algorithms to update the index.

Text can be viewed as a very long string of data. Often text has little or no structure, and in many applications we wish to process the text without concern for the structure. Gonnet [1983] used the term, *unstructured database*, to refer to this type of data. Examples of such collections are: dictionaries, legal cases, articles on wire services, scientific papers, etc.

Text can instead be structured as a sequence of words. Each *word* is a string that does not include any symbol from a special separator set. For example, a “space” is usually an element of such a set.

2.3. A LOGICAL INDEX FOR TEXT. Let’s assume that the text to be searched is a single string and padded at its right end with an infinite number of null (or any special) characters. A *suffix* or *semi-infinite string (sistring)* [Knuth 1973; Gonnet 1988] is the sequence of characters starting at any position of the text and continuing to the right. For example, if the text is

The traditional approach for searching a regular expres-
sion...

the following are some of the possible sistrings:

The traditional approach for searching...
he traditional approach for searching a...
e traditional approach for searching a...
onal approach for searching a regular...

Two sistrings starting at different positions are always different. To guarantee that no one semi-infinite string be a prefix of another, it is enough to end the text with a unique end-of-text symbol that appears nowhere else [Knuth 1973]. Thus, sistrings can be unambiguously identified by their starting position [Aho et al. 1974, “position trees”]. The result of a lexicographic comparison between two sistrings is based on the text of the sistrings, not their positions. Now we introduce our text index, which is a binary digital tree or binary trie [Gonnet and Baeza-Yates 1991] constructed from the set of sistrings of the text. We use a particular class of tries. However, we should remark that this is a theoretical model, because in practice they use too much space. Later we discuss how this index can be implemented efficiently in practice.

Tries are recursive tree structures that use the digital decomposition of strings to represent a set of strings and to direct the searching. Tries were invented by de la Briandais [1959] and the name was suggested by Fredkin [1960], from information *retrieval*. If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, that string’s position is stored in an external node.

Figure 1 shows a binary trie (binary alphabet) for the string “01100100010111...” after inserting the sistrings that start from positions 1 through 8. (In this case, the sistring is represented by its starting position in the text.)

The *height* of a trie is the number of nodes in the longest path from the root to an external node. The length of any path from the root to an external node is

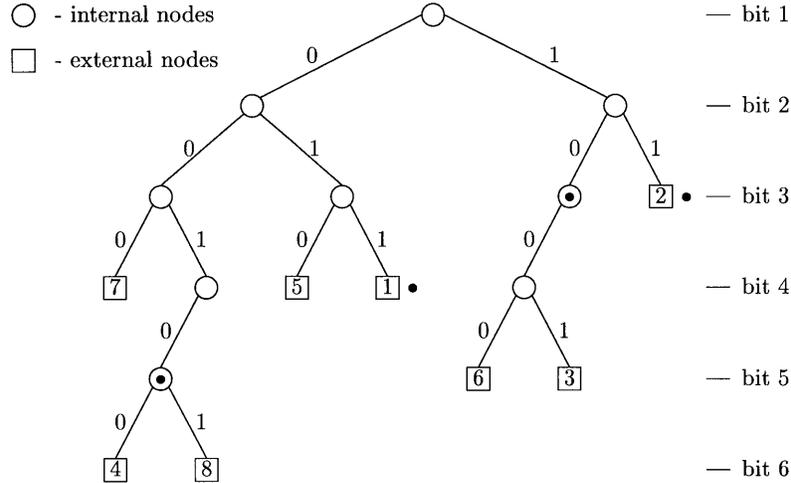


FIG. 1. Binary trie (external node label indicates position in the text) for the first eight sistrings in “01100100010111 . . .”.

bounded by the height of the trie. On average, the height of a trie is logarithmic for any square-integrable probability distribution [Devroye 1982]. For a random uniform distribution, we have Régnier [1981]:

$$\mathcal{H}(n) = 2 \log_2(n) + o(\log_2(n))$$

for a binary trie containing n strings. The main disadvantage of plain tries is that the number of internal nodes could be $O(n^2)$ in the worst case.

2.4. PRACTICAL ASPECTS. In this section, we briefly discuss how to implement a binary trie in practice. One possible solution to reduce the number of internal nodes is to use a Patricia tree. A *Patricia tree* [Morrison 1968] is a trie with the additional constraint that single-descendant nodes are eliminated.¹ A counter is kept in each node to indicate which is the next bit to inspect.

For n sistrings, such an index has n external nodes (the n positions of the text) and $n - 1$ internal nodes. Each internal node consists of a pair of pointers plus some counters. Thus, the space required is $O(n)$. For many applications, it is not necessary to store all the possible sistrings. For example, it may be sufficient to store sistrings that start at the beginning of a word. This reduces the space by a factor proportional to the average word size, which for large databases may be a significant saving. Also, note that n does not change by the use of a binary representation of the underlying alphabet (typically ASCII).

It is possible to build the index in $O(n\mathcal{H}(n))$ time, where $\mathcal{H}(n)$ denotes the height of the tree. As for tries, the expected height of a Patricia tree is logarithmic (and at most the height of the binary trie). The expected height of a Patricia tree is $\log_2 n + o(\log_2 n)$ [Pittel 1986]. We refer the reader to Gonnet and Baeza-Yates [1991] for updated algorithms on a Patricia tree.

¹ This name is an acronym for “Practical Algorithm To Retrieve Information Coded In Alphanumeric”.

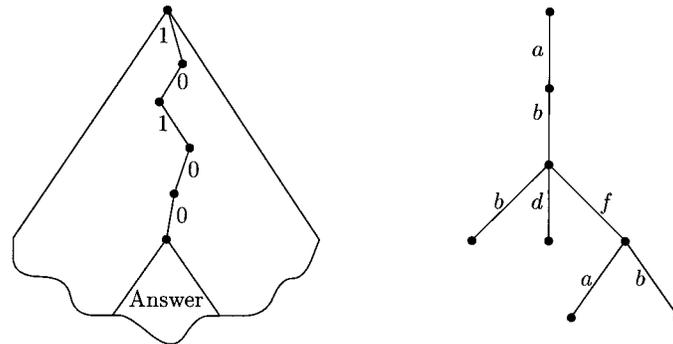


FIG. 2. (a) Prefix searching for "10100". (b) Complete prefix trie of the words *abb*, *abd*, *abfa*, *abfb*, *abfaa*, and *abfab*.

A trie built using the sistrings of a string is also called *suffix tree* [Aho et al. 1974]. Similarly, a Patricia tree is called a *compact suffix tree*. In a compact suffix tree, instead of maintaining the next bit to inspect, we have pointers to a piece of text that represent the collapsed path. The disadvantage of compact suffix trees is that in general they need more space in each internal node. Although a random trie (*independent* strings) is different from a random suffix tree (*dependent* strings) in general, both models are equivalent when the symbols are uniformly distributed [Apostolico and Szpankowski 1992]. Moreover, the complexity should be the same for both cases, because $\mathcal{H}_{\text{suffix tree}}(n) \leq \mathcal{H}_{\text{trie}}(n)$ [Apostolico and Szpankowski 1992]. Simulation results suggest that the asymptotic ratio of both heights converges to 1 [Szpankowski 1993].

For some applications, Patricia trees still need too much space. Tries can be simulated efficiently by using suffix arrays (also called PAT arrays) [Gonnet 1983; Manber and Myers 1990]. A suffix array just stores the leaves of the trie in lexicographical order according to the text contents. Following any pointer can be simulated on $O(\log n)$ time by using two binary searches. In this case, we do not have to use a binary alphabet. So all our results are valid for suffix arrays with that overhead.

3. Searching a Set of Strings

Efficient searching of all sistrings having a given prefix can be done using a Patricia tree [Knuth 1973]. To search for a string (a word, a sequence of words, etc.), we start at the root of the tree following the searched string. Since in the search we may skip bits, one final comparison with the actual text is needed to verify the result (this is not necessary in compact suffix trees, as already mentioned). If the search ends inside the tree, the subtree rooted at the last inspected node contains the complete answer (see Figure 2(a)). If the search string extends beyond the limits of the tree, then at most one position in the text matches. Thus, a search is done in $|string|$ steps. Similarly, if we build a tree with reversed text (that is, sistrings to the left), we can have *suffix* searching.

We extend the prefix searching algorithm to the case of a finite set of strings. The simplest solution is to use prefix searching for every member of the set. However, we can improve the search time by recognizing common prefixes in our query.

Let the *complete prefix trie* (CPT) be the trie of the set of strings, such that

- there are no truncated paths, that is, every word corresponds to a complete path in the trie; and
- if a word w is a prefix of another word x , then only w is stored in the trie. In other words, the search for w is sufficient to also find the occurrences of x .

Figure 2(b) shows an example.

Note that the CPT is also an automaton that recognizes a set of strings, where the final states are the ending nodes. Now we present a first algorithm that simulates an automaton which is a tree on the trie. We extend this concept in the following sections. Hence, the new searching algorithm is

- Construct the complete prefix trie of the query using a binary alphabet.
- Traverse simultaneously, if possible, the complete prefix trie and the Patricia tree of sistrings in the text (the “index”). That is, follow at the same time a 0 or 1 edge, if they exist in both the trie and the CPT. All the subtrees in the index associated with terminal nodes in the complete prefix trie are the answer. As in prefix searching, because we may skip bits while traversing the index, a final comparison with the actual text is needed to verify the result.

Let $|S|$ be the sum of the lengths of the set of strings in the query. The construction of the complete prefix trie requires time $O(|S|)$. Similarly, the number of nodes of the trie is $O(|S|)$, thus, the searching time is of the same order.

An extension of this idea leads to the definition of prefixed regular expressions.

4. Prefixed Regular Expressions

A *prefixed regular expression* (PRE for short) is defined recursively by

- \emptyset is a PRE and denotes the empty set.
- ϵ (empty string) is a PRE and denotes the set $\{\epsilon\}$.
- For each symbol a in Σ , a is a PRE and denotes the set $\{a\}$.
- If p and q are PREs denoting the regular sets P and Q , respectively, r is a regular expression denoting the regular set R such that $\epsilon \in R$, and $x \in \Sigma$, then the following expressions are also PREs:
 - $p + q$ (union) denotes the set $P \cup Q$.
 - xp (concatenation with a symbol on the left) denotes the set xP .
 - pr (concatenation with an ϵ -regular expression on the right) denotes the set PR , and
 - p^* (star) denotes the set P^* .

For example $ab(bc^* + d^+ + f(a + b))$ is a PRE, while $a\Sigma^*b$ is not a PRE. Note that some regular expressions corresponding to finite languages are not PRE, such as $(a + b)(c + d)(b + c)$.

The main property of a PRE is that there exists a unique finite set of words in the language, called *prewords*, such that:

TABLE I. MEMBERSHIP TESTING AND PREWORDS OF A PRE-TYPE QUERY

<i>Query</i>	<i>PRE(Query)</i>	<i>prewords(Query)</i>
\emptyset	true	\emptyset
ϵ	true	$\{\epsilon\}$
$x \in \Sigma$	true	$\{x\}$
$r + s$	<i>PRE(r)</i> and <i>PRE(s)</i>	<i>prewords(r)</i> \cup <i>prewords(s)</i>
$r s$	(<i>r</i> is a string and <i>PRE(s</i>)) or (<i>PRE(r)</i> and ($\epsilon \in L(s)$))	<i>prewords(r)</i> <i>prewords(s)</i>
r^*	true	$\{\epsilon\}$

- for any other string in the language, one of these words is a proper prefix of that string,
- the *prefix property* [Hopcroft and Ullman 1979, page 121] holds; that is, no word in this set is a proper prefix of another word in the set.

In other words, the automaton that recognizes a PRE can be seen as being a graph having a tree where all the final states lie.

Note that the size of the complete prefix trie of the prewords is linear in the size of the query. For example, the prewords of $ab(bc^* + d^+ + f(a + b))$ are abb , abd , $abfa$, and $abfb$. The complete prefix trie for this set is shown in Figure 2(b).

We use the notation $PRE(r) = \text{true}$ if r is a PRE, and $prewords(r)$ to denote the set of prewords of r . Table I gives descriptive recursive techniques to test if a query is a PRE and to find the prewords of a query. In both cases, the total time needed is linear in the size of the query.

To search a PRE query, we use the algorithm to search for a set of strings, using the prewords of the query. Because the number of nodes of the complete prefix trie of the prewords is $O(|query|)$, the search time is also $O(|query|)$. Formally:

THEOREM 4.1. *It is possible to search a PRE query in time $O(|query|)$ independently of the size of the answer.*

5. General Automaton Searching

In this section, we present the algorithm that can search for arbitrary regular expressions in time sublinear in n on the average. For this, we simulate a DFA in a binary trie built from all the sistrings of a text.

The main steps of the algorithm are:

- (a) Convert the regular expression passed as a query into a minimized DFA that may take exponential space/time with respect to the query size but is independent of the size of the text [Hopcroft and Ullman 1979].
- (b) Next, eliminate outgoing transitions from final states (see justification in step (e)). This may induce further minimization.
- (c) Convert character DFAs into binary DFAs using any suitable binary encoding of the input alphabet. Each state will then have at most two outgoing transitions, one labelled 0 and one labelled 1.

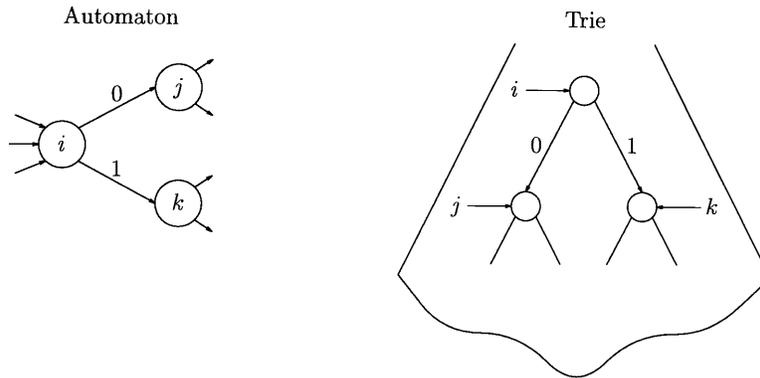


FIG. 3. Simulating the automaton on a binary digital tree.

- (d) Simulate the binary DFA on the binary trie from all sistrings of text using the same binary encoding as in step (b). That is, associate the root of the tree with the initial state, and, for any internal node associated with state i , associate its left descendant with state j if $i \rightarrow j$ for a bit 0, and associate its right descendant with state k if $i \rightarrow k$ for a bit 1 (see Figure 3).
- (e) For every node of the index associated with a final state, accept the whole subtree and halt the search in that subtree. (For this reason, we do not need outgoing transitions in final states). That is, we stop when we find the shortest string matching the query starting at a given position.
- (f) On reaching an external node, run the remainder of the automaton on the single string determined by this external node.

To adapt the algorithm to run on a Patricia tree, wherever we skip bits, it is necessary to read the corresponding text to determine the validity of each “skipped” transition: the sistring starting at any position in the current subtree may be used. A depth-first traversal to associate automaton states with trie nodes ensures $O(\mathcal{H}(n))$ space for the simulation.

With some changes, the same algorithm is suitable for nondeterministic finite automata. In this event, for each node in the tree we have at most $O(|query|)$ active states during the simulation and ϵ -transitions are handled in a special manner (for the state expansion the ϵ -closure of the corresponding transition is used). Hence, we need $O(|query|\mathcal{H}(n))$ space for the simulation. A compromise in complexity is achieved by using a non-deterministic finite automaton with no ϵ -transitions. In this case, the NFA has only $O(|query|)$ states and transitions [Hopcroft and Ullman 1979]. Hence, the potential exponential size of the DFA is avoided.

The average case analysis of the above algorithm is not simple. For that reason, we present it in Appendix B, and we give here an account of the proof by using an approximate analysis and presenting our main results.

Let \mathbf{H} be the incidence matrix of the graph that represents the DFA. Suppose that the trie is perfectly balanced and let $\tilde{N}(n)$ be the expected number of internal nodes visited by the algorithm. Then, we can express one step of the simulation with

$$\tilde{N}(n) = \mathbf{H}\tilde{N}\left(\frac{n}{2}\right).$$

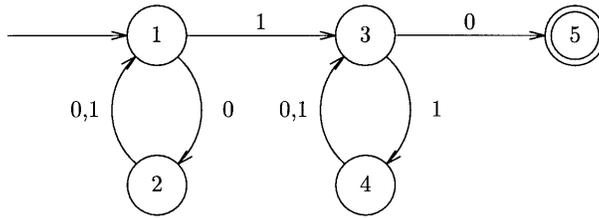


FIG. 4. Deterministic finite automaton for $(0(1 + 0))^*1(1(1 + 0))^*0$.

Iterating the recurrence, we obtain $\vec{N}(n) = \mathbf{H}^{\log_2 n} \vec{N}(1)$. The power of a matrix times a vector can be written as a linear combination of its eigenvectors. Let λ be the largest eigenvalue of \mathbf{H} . Then, we obtain that $N_1(n) = O(n^{\log_2 \lambda})$. We are able to prove that $|\lambda| < 2$ for any regular expression (because we stop at final states).

The total running time of the algorithm is proportional to the number of internal nodes visited plus the time spent searching the text associated to final states. The number of final states is proportional to $N_1(n)$, and using the same idea, we show that the expected searching time on just one string is bounded by a constant. Although a Patricia tree is not balanced, we are able to show that the complexity is almost the same as the balanced case. In fact, in Appendix B, we show that:

THEOREM 5.1. *The expected number of comparisons performed by a DFA for a query q represented by its incidence matrix \mathbf{H} while searching the trie of n random strings is sublinear, and given by*

$$O((\log_2 n)^t n^r),$$

where $r = \log_2 \lambda < 1$, $\lambda = \max_i (|\lambda_i|)$, $t = \max_i (m_i - 1)$, such that $|\lambda_i| = \lambda$, and the λ_i 's are the eigenvalues of \mathbf{H} with multiplicities m_i .

In Appendix B, we also give the time complexity for NFAs, the relation with suffix trees and the application of our analysis techniques to other problems. The complexity for Patricia trees or suffix arrays is also sub-linear, as in the worst case, only a constant factor (Patricia tree) or a $O(\log n)$ factor (suffix arrays) is included. Now we give an example of the algorithm execution:

Example 1. Let A be the DFA of the regular expression $(0(1 + 0))^*1(1(1 + 0))^*0$ (see Figure 4).

Suppose that we want to search this regular expression in the digital trie of Figure 1. By simulating the DFA with the trie as input (i.e., using as input all possible paths of the trie, the algorithm stops at the internal nodes and leaves marked with a \bullet in Figure 1. The final answer is all the text positions of the subtrees and leaves involved. The average case analysis for this example is given in Appendix B.

As a corollary of the theorem above, it is possible to characterize the complexity of different types of regular expressions according to the DFA structure. For example, DFAs having only cycles of length 1, have a largest eigenvalue equal to 1, but with multiplicity proportional to the number of cycles, obtaining a complexity of $O(\text{polylog}(n))$.

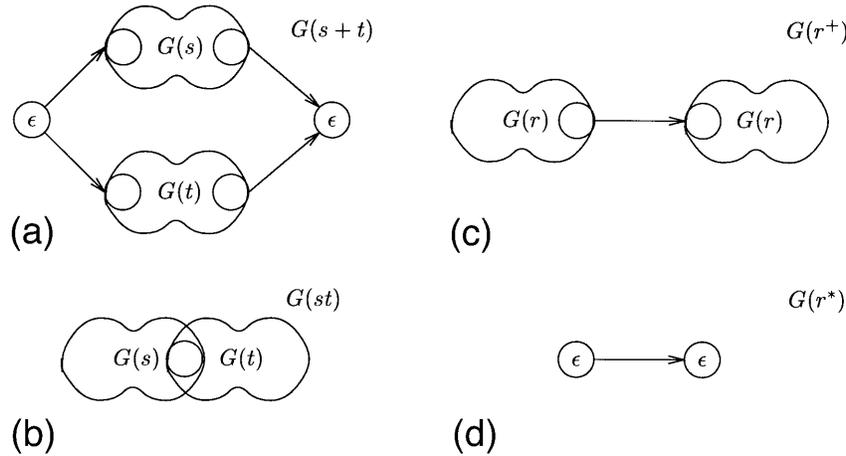


FIG. 5. Recursive definition of a substring graph.

6. Substring Analysis for Query Planning

In this section, we present a general heuristic, which we call *substring analysis*, to plan what algorithms and order of execution should be used for a generic pattern matching problem, which we apply to regular expressions.

Even when queries are represented by regular expressions, with query planning sometimes we can do much better than automata searching. For example, if the query is $a \Sigma^* b$ (“a” followed by “b”) we have two choices. One is to search for an a , and then follow every possible path in the index trying to find all possible b ’s. The second is to go to the text after each a , and sequentially search for b . In both cases, this is useless if there are no b ’s at all in the text. The aim of this section is to find from every query a set of necessary conditions that have to be satisfied. Hence, for many cases, this heuristic can save processing time later on. For this we introduce a graph representation of a query.

We define the *substring graph* of a regular expression r to be an acyclic directed graph such that each node is labelled by a string. The graph is defined recursively by the following rules:

- $G(\epsilon)$ is a single node labelled ϵ .
- $G(x)$ for any $x \in \Sigma$ is a single node labelled with x .
- $G(s + t)$ is the graph built from $G(s)$ and $G(t)$ with an ϵ -labelled node with edges to the source nodes and an ϵ -labelled node with edges from the sink nodes, as shown in Figure 5(a).
- $G(st)$ is the graph built from joining the sink node of $G(s)$ with the source node of $G(t)$ (Figure 5(b)), and relabelling the node with the concatenation of the sink label and the source label.
- $G(r^+)$ are two copies of $G(r)$ with an edge from the sink node of one to the source node of the other, as shown in Figure 5(c).
- $G(r^*)$ is two ϵ -labelled nodes connected by an edge (Figure 5(d)).

Figure 6(a) shows the substring graph for the query $(ca^+s + h(a + e^*)m)(a + e)$. Each path in the graph represents one of the sequence of substrings that must be found for the pattern to match.

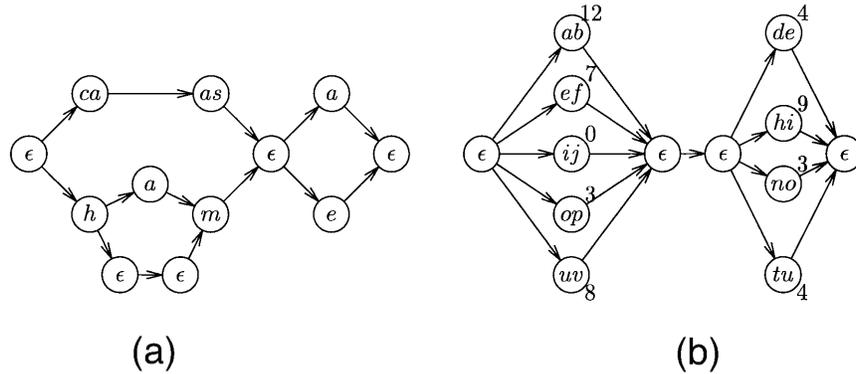


FIG. 6. Substring graph of two queries.

It is possible to construct this graph in time $O(|query|)$, using the above recursive definition.

Given a regular expression r , let $source(r)$ be the unique source node of $G(r)$ (no incoming edges). Similarly, let $sink(r)$ be the unique sink node of $G(r)$ (node without outgoing edges). All the node labels in a path between $source(r)$ and $sink(r)$ are substrings of a subset of words in $L(r)$. For example, in Figure 6(a) the strings ca , as and a are substrings of the subset ca^+sa of $L(r)$. Therefore, each label in $G(r)$ is a *necessary condition* for a nonempty subset of words in $L(r)$. In other words, any word in the regular set described by r contains all the labels of some path in $G(r)$ as substrings.

We explain the use of the substring graph through an example. Suppose that we have the query $q = (ab + ef + ij + op + uv)\Sigma^*(de + hi + no + tu)$. The substring graph $G(q)$ is shown in Figure 6(b).

After building $G(q)$, we search for all node labels in $G(q)$ in our index of sistrings, determining whether or not that string exists in the text. This step requires $O(|q|)$ time, because the sum of the label lengths is $O(|q|)$. For example, the number of occurrences for each label in our example for a text sample is given beside each node in Figure 6(b).

For all nonexistent labels, we remove:

- the corresponding node,
- adjacent edges, and
- any adjacent nodes (recursively) from which all incoming edges or all outgoing edges have been deleted.

In the example of Figure 6(b), we remove the node labelled by ij and the adjacent edges. This reduces the size of the query. Such reduction happens in practice when we have long labels or erroneous queries (misspellings).

Second, from the number of occurrences for each label we can obtain an upper bound on the size of the final answer to the query. For adjacent nodes (serial, or *and*, nodes) we multiply both numbers, and for parallel nodes (*or* nodes) we add the number of occurrences.

At this stage, ϵ -nodes are simplified and treated in a special way:

- consecutive serial ϵ -nodes are replaced by a single ϵ -node (for example, the lowest nodes in Figure 6(a)),
- chains that are parallel to a single ϵ -node, are deleted (e.g., the leftmost node labelled with a in Figure 6(a)), and

—the number of occurrences in the remaining ϵ -nodes is defined as 1 (after the simplifications, ϵ -nodes are always adjacent to non- ϵ -nodes, since ϵ was assumed not to be a member of the query).

In our example, the number of occurrences for Figure 6(b) can be bounded by 600. Similar to search strategies in inverted files [Knuth 1973; Batory 1979], this bound is useful in deciding future searching strategies for the rest of the query.

7. Final Remarks

We have shown that using a trie or Patricia tree, we can search for many types of string searching queries in logarithmic average time, independently of the size of the answer. We also show that automaton searching in a trie is sublinear in the size of the text on average for any regular expression, this being the first algorithm found to achieve this complexity. Similar ideas have been used since for approximate string searching by simulating dynamic programming over a digital tree [Gonnet et al. 1992; Ukkonen 1993], also achieving sublinear time on average. In particular, Gonnet et al. [1992] have used this algorithm for protein matching.

In general, however, the worst case of automata searching is linear. For some regular expressions and a given algorithm it is possible to construct a text such that the algorithm must be forced to inspect a linear number of characters. The pathological cases consist of periodic patterns or unusual pieces of text that, in practice, are rarely found. An implementation of this algorithm using the Oxford English Dictionary showed that most real queries coming from users as regular expressions were resolved in about $O(\sqrt{n})$ node inspections. This is very efficient for all practical purposes.

Finding an algorithm with logarithmic search time for any RE query is still an open problem [Galil 1985]. Another open problem is to derive a lower bound for searching REs in preprocessed text. Besides this, an interesting question is whether there exist efficient algorithms for the bounded “followed-by” problem (i.e., $s_1 \Sigma^{\leq k} s_2$ for fixed k). For solutions to the case $s_1 \Sigma^* s_2$ see Baeza-Yates [1989] and Manber and Baeza-Yates [1991].

Appendix A. Query Examples

All the examples given here arise from the OED, where tags are used to represent the structure. We use $\langle \text{tag} \rangle$ and $\langle / \text{tag} \rangle$ to denote starting and ending tags.

- (1) All citations to an author with prefix Scot followed by at most 80 arbitrary characters then by works beginning with the prefix Kenilw or Discov:

$$\langle \text{A} \rangle \text{Scot } \Sigma^{\leq 80} \langle \text{W} \rangle (\text{Kenilw} + \text{Discov}).$$

Here, tag A is used for authors, W for work titles.

- (2) All first citations accredited to Shakespeare between 1610–11:

$$\langle \text{EQ} \rangle (\langle \text{LQ} \rangle) ? \langle \text{Q} \rangle \langle \text{D} \rangle 161(0 + 1) \langle / \text{D} \rangle \langle \text{A} \rangle \text{Shak}$$

Here EQ is used for all citations (if there is only one could be the last citation: LQ), Q for each citation, and D for dates.

(3) All references to author W. Scott:

$$\langle A \rangle ((S\text{ir}b)?W)?b\text{Scott}b? \langle /A \rangle,$$

where b denotes a space.

Appendix B. Exact Analysis

We analyze the algorithm for a random trie under the independent model. We already know that the independent model is equivalent to the dependent model for the random uniform case as pointed out in Section 2.3.

Let the initial state of the DFA be labelled 1. Let \mathbf{H} be the incidence matrix of the DFA (that is, $h_{ij} \in \{0, 1, 2\}$ is the number of symbol transitions from state i to state j) and \vec{F} be a constant vector such that F_i is 1 for all i .

LEMMA B.1. *Let $\vec{N}(n)$ be the vector $[N_1(n), \dots, N_s(n)]$, where $N_i(n)$ is the expected number of internal nodes visited when a DFA using state i as initial state is simulated on a binary trie of n random strings, and s is the number of states of the DFA. Then we have*

$$\vec{N}(n) = \sum_{k \geq 0} \tau_{n,k} \mathbf{H}^k \vec{F},$$

where

$$\tau_{n,k} = 1 - \left(1 - \frac{1}{2^k}\right)^n - \frac{n}{2^k} \left(1 - \frac{1}{2^k}\right)^{n-1}.$$

PROOF. For a nonfinal state i , the expected number of internal nodes visited starting from it in a binary trie of n random strings can be expressed as ($n > 1$)

$$N_i(n) = 1 + \frac{1}{2^n} \sum_{k=0}^n \binom{n}{k} (N_j(k) + N_\ell(n - k)),$$

where $\delta(0, i) = j$ and $\delta(1, i) = \ell$. This is similar to the analysis of the expected number of nodes of a binary trie [Knuth 1973]. For final states, we have $N_f(n) = 1$ for $n > 1$. For undefined states (when $\delta(x, i)$ is not defined) we define $N_{undef}(n) = 0$. The initial conditions are $N_i(0) = N_i(1) = 0$ for any state i .

Introducing exponential generating functions in the above equation, that is,

$$N_i(z) = \sum_{n \geq 0} N_i(n) \frac{z^n}{n!},$$

we obtain

$$N_i(z) = \exp\left(\frac{z}{2}\right) \left(N_j\left(\frac{z}{2}\right) + N_\ell\left(\frac{z}{2}\right) \right) + \exp(z) - 1 - z.$$

Writing all the equations as a matrix functional equation, we have

$$\tilde{N}(z) = \exp\left(\frac{z}{2}\right) \mathbf{H} \tilde{N}\left(\frac{z}{2}\right) + f(z) \tilde{F},$$

where $f(z) = e^z - 1 - z$.

This functional equation may be solved formally by iteration [Flajolet and Puech 1986], obtaining

$$\tilde{N}(z) = \sum_{k \geq 0} \exp(z(1 - 1/2^k)) f\left(\frac{z}{2^k}\right) \mathbf{H}^k \tilde{F}.$$

From here, it is easy to obtain $\tilde{N}(n) = n! [z^n] \tilde{N}(z)$ using the series expansion of $\exp(x)$, where $[x^n]P(x)$ denotes the coefficient in x^n of the polynomial $P(x)$, from which the result follows. \square

From the previous lemma, we can obtain the exact number of nodes visited by the DFA for any fixed n . Note that the i th element of $\mathbf{H}^k \tilde{F}$ represents all possible paths in the DFA of length k that finish in state i . The next step is to obtain the asymptotic value of $N_1(n)$. This is based on the analysis of partial match queries in k - d -tries of Flajolet and Puech [1986]. In fact, their analysis is similar to a DFA with a single cycle of length k . We extend that analysis to any DFA.

LEMMA B.2. *The asymptotic value of the expected number of internal nodes visited is*

$$N_1(n) = \gamma_1(\log_2 n)(\log_2 n)^{m_1-1} n^{\log_2 |\lambda_1|} + O((\log_2 n)^{m_1-2} n^{\log_2 |\lambda_1|} + \log_2 n),$$

where $2 \geq |\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_t| \geq 0$, $\lambda_i \neq \lambda_j$ are the eigenvalues of \mathbf{H} with multiplicities m_1, \dots, m_t , s is the order of \mathbf{H} (number of states = $\sum_i m_i = s$), and $\gamma_1(x)$ is an oscillating function of x with period 1, constant mean value and small amplitude. We later show that $|\lambda_1| < 2$ strictly.

PROOF. Decomposing \mathbf{H} in its upper normal Jordan form [Gantmacher 1959], we have

$$\mathbf{H} = \mathbf{P} \mathbf{J} \mathbf{P}^{-1},$$

where \mathbf{J} is a block diagonal matrix of the form

$$\mathbf{J} = \begin{bmatrix} J_1 & 0 & \dots & \dots \\ 0 & J_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & 0 & J_t \end{bmatrix},$$

J_i is a $m_i \times m_i$ square matrix of the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & \dots \\ 0 & \lambda_i & 1 & 0 \\ \dots & \dots & \dots & 1 \\ \dots & \dots & 0 & \lambda_i \end{bmatrix},$$

and \mathbf{P} has as columns the respective eigenvectors. Then

$$\mathbf{H}^k = \mathbf{P}\mathbf{J}^k\mathbf{P}^{-1},$$

where \mathbf{J}^k is the block diagonal matrix $[J_i^k]$, and each J_i^k is of the form [Gantmacher 1959]:

$$J_i^k = \begin{bmatrix} \lambda_i^k & k\lambda_i^{k-1} & \frac{k(k-1)}{2}\lambda_i^{k-2} & \dots & \binom{k}{m_i-1}\lambda_i^{k+1-m_i} \\ 0 & \lambda_i^k & k\lambda_i^{k-1} & \dots & \binom{k}{m_i-2}\lambda_i^{k+2-m_i} \\ 0 & 0 & \lambda_i^k & \dots & \binom{k}{m_i-3}\lambda_i^{k+3-m_i} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & 0 & \lambda_i^k \end{bmatrix}.$$

Therefore

$$\tilde{N}(n) = \sum_{k \geq 0} \tau_{n,k} \mathbf{H}^k \vec{F} = \mathbf{P} \left(\sum_{k \geq 0} \tau_{n,k} \mathbf{J}^k \right) \mathbf{P}^{-1} \vec{F}.$$

Let

$$S_j = \sum_{k \geq 0} \tau_{n,k} \binom{k}{j} \lambda^{k-j}.$$

The convergence of S_j is guaranteed by the fact that, for fixed n and large k we have

$$\tau_{n,k} \approx 1 - \exp\left(-\frac{n}{2^k}\right) - \frac{n}{2^k} \exp\left(-\frac{(n-1)}{2^k}\right) = O\left(\frac{n^2}{4^k}\right).$$

The asymptotic value of $S_0(\lambda > 1)$ has already been obtained by Flajolet and Puech [1986] as

$$S_0 = \gamma(\log_2 n) n^{\log_2 \lambda} + O(1),$$

where $\gamma(x)$ is a periodic function of x with period 1, mean value depending only on λ , and with a small amplitude. For details about the exact computation of the function γ using Mellin transform techniques, we refer the reader to Flajolet and Puech [1986].

The asymptotic value of $S_j(\lambda > 1)$ is obtained in a similar way by using Mellin transform techniques, obtaining

$$S_j = \frac{\gamma(\log_2 n)}{j!} \left(\frac{\log_2 n}{\lambda}\right)^j n^{\log_2 \lambda} + O(S_{j-1}).$$

If $\lambda = 1$, then $S_j = O((\log_2 n)^{j+1})$. Then, for λ_i we have that the dominant term is

$$\gamma_i(\log_2 n)(\log_2 n)^{m_i-1} n^{\log_2 \lambda_i},$$

where we include the constant

$$\frac{1}{(m_i - 1)! \lambda_i^{m_i-1}}$$

as part of the function γ_i . The highest order term of $N_1(n)$ is given by λ_1 . In the case that there is more than one eigenvalue with the same modulus, λ_1 is the one with largest multiplicity. \square

Example 1 (Analysis). Let \mathcal{A} be the DFA of the regular expression $(0(1 + 0))^*1(1(1 + 0))^*0$ (see Figure 4). The incidence matrix for \mathcal{A} (state 1 is the initial state, and state 5 is the final state) is

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The eigenvalues are $\sqrt{2}$ and $-\sqrt{2}$, each with multiplicity 2, and 0. The Jordan normal form for \mathbf{H} is:

$$\mathbf{J} = \begin{bmatrix} \sqrt{2} & 1 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & -\sqrt{2} & 1 & 0 \\ 0 & 0 & 0 & -\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

and \mathbf{P} is the matrix where column i is an eigenvector associated to the eigenvalue in \mathbf{J}_{ii} . Namely

$$\mathbf{P} = \begin{bmatrix} \sqrt{2}/2 & 1/2 & -\sqrt{2}/2 & 1/2 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & \sqrt{2} & 0 & -\sqrt{2} & 0 \\ 0 & 2 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

Computing $\tilde{N}(n) = \mathbf{P}\mathbf{J}^k\mathbf{P}^{-1}\tilde{F}$ we obtain

$$N_1(n) = ((1 + \sqrt{2})\gamma_1(\log_2 n) + (1 - \sqrt{2})\gamma_2(\log_2 n)) \frac{\sqrt{n} \log_2 n}{8} + O(\sqrt{n}),$$

where $\gamma_1(x)$ and $\gamma_2(x)$ are two oscillating functions with period x , average value 1, and small amplitude.

Example 2. For the regular expression $(0(0 + 1)0)^*1$, the eigenvalues are

$$2^{1/3}, -\frac{1}{2}(2^{1/3} - 3^{1/2}2^{1/3}i), -\frac{1}{2}(2^{1/3} + 3^{1/2}2^{1/3}i), \text{ and } 0,$$

and the first three have the same modulus. The solution in this case is $N_1(n) = O(n^{1/3})$.

Using the fact that the DFA is connected, and that final states do not have outgoing transitions, the next lemma gives some characteristics of the eigenvalues of \mathbf{H} .

LEMMA B.3. $1 \leq \max_k(|\lambda_k|) < 2$.

PROOF. Decompose the DFA (a directed graph) into its strongly connected components, C_i , (cycles), and all other states, R . For each state in R , we have an eigenvalue with value 0. To each strongly connected component, C_i , we have an irreducible nonnegative matrix, H_i , as incidence matrix. In each one of these matrices, there is a real eigenvalue r of maximal modulus with a positive associated eigenvector (Perron–Frobenius Theorem [Gantmacher 1959]). Moreover, if there are m eigenvalues of modulus r , all of them are distinct and are the roots of the equation

$$\lambda^m - r^m = 0.$$

In other words, $\lambda_k = r e^{i(2\pi k/m)}$ for $k = 0, \dots, m - 1$.

Because the whole automaton (graph) is weakly connected, for each H_i there is at least a row in which some $h_{ij} = 1$ ($i \neq j$) and $h_{ik} = 0$ for all $k \neq j$. In other words, there is a transition to a state not in C_i .

The maximal eigenvalue of a $n \times n$ nonnegative irreducible matrix \mathbf{A} is bounded by

$$\lambda_{\max} \leq \frac{p(1 - \epsilon)\rho_p + n\epsilon\rho}{p(1 - \epsilon) + n\epsilon},$$

due to a result of Ostrowski and Schneider [1960], where

$$\epsilon = \left(\frac{\kappa}{R - \text{Min}} \right)^{n-1},$$

$\text{Min} = \min_i(A_{ii})$, $\kappa = \min_{i \neq j}(A_{ij} | A_{ij} > 0)$, $R = \max_i r_i$, $\rho_i = \sum_{j=1}^i r_j$, $r_j = \max_k A_{jk}$, and p is the smallest integer i , such that

$$\sum_{j=1}^i (r_j - r_{i+1}) \geq \epsilon \sum_{j=i+2}^n (r_{i+1} - r_j).$$

For each H_i , let $d_i \geq 1$ be the number of states with only one transition, and $s_i \geq 2$ the total number of states ($s_i > d_i$). Therefore, in our case $\text{Min} = 0$, $\kappa =$

1, $R = 2$, $\epsilon = 1/2^{s_i-1}$ and $p = n - d_i$. Replacing these values in the bound, and taking the maximum over all possible values of i , we have that

$$\max_k (|\lambda_k|) \leq 2 - \min_i \left(\frac{d_i}{(s_i - d_i)2^{s_i-1} + d_i} \right) < 2.$$

A similar result gives the lower bound $\max_k (|\lambda_k|) \geq 1$ [Ostrowski and Schneider 1960]. \square

The previous lemma shows that the average time is strictly sublinear and gives an upper bound for the largest eigenvalue (this bound is not tight; for example, in Example 1 we obtain an upper bound of $5/3$). Nevertheless, we can construct DFA's that visit $O(n^{1-\delta})$ nodes, for arbitrarily small δ . For example, for $1(\Sigma^m)^*1$, we have $\delta = 1/m$, although queries of this type are not common in practice.

LEMMA B.4. *The expected number of external nodes visited by the DFA is $O(N_1(n))$.*

PROOF. Starting from a nonfinal state i , the expected number of external nodes of a binary trie visited can be expressed as

$$N_i(n) = \frac{1}{2^n} \sum_{k=0}^n \binom{n}{k} (N_j(k) + N_\ell(n-k)) \quad (n > 1),$$

where $\delta(0, i) = j$ and $\delta(1, i) = \ell$ [Knuth 1973]. For final states, we have $N_f(n) = 1$ for $n > 1$. For undefined states (when $\delta(x, i)$ is not defined), we define $N_{undef}(n) = 0$. The initial conditions are $N_i(1) = 1$ for any state i . The solution for this case is of the same order of the solution of the recurrence in Lemma B.2. \square

The total time spent by the DFA is proportional to the number of nodes visited (internal and external) plus the number of comparisons performed to check the remainder of the single string corresponding to each external node. Thus, now we find the expected number of comparisons performed by the DFA when an external node is reached. As before, the DFA does not have outgoing transitions from final states.

LEMMA B.5. *The expected number of comparisons, $C(n)$, performed by a DFA while searching a random string of length ℓ is bounded by a constant that is independent of ℓ .*

PROOF. The average number of comparisons in a random string of length ℓ starting in a non-final state i , can be expressed as

$$C_i(\ell) = 1 + \frac{1}{2} (C_j(\ell-1) + C_k(\ell-1)) \quad (\ell > 0),$$

where $\delta(0, i) = j$ and $\delta(1, i) = k$ [Knuth 1973]. For final states we have $C_f(\ell) = 1$ for $\ell > 0$. For undefined states (when $\delta(x, i)$ is not defined), we define $C_{undef}(\ell) = 0$. The initial conditions are $C_i(0) = 0$ for any state i .

In a matrix form, we have

$$\vec{C}(\ell) = \frac{1}{2} \mathbf{H} \vec{C}(\ell - 1) + \vec{F}.$$

Solving by iteration, we obtain

$$\vec{C}(\ell) = \sum_{j=0}^{\ell-1} \frac{1}{2^j} \mathbf{H}^j \vec{F}.$$

Using the same decomposition of Lemma B.2 for \mathbf{H} , we have

$$\vec{C}(\ell) = \mathbf{P} \left(\sum_{j=0}^{\ell-1} \frac{1}{2^j} \mathbf{J}^j \right) \mathbf{P}^{-1} \vec{F}.$$

Therefore, we have summations of the form

$$S_j = \sum_{k \geq 0} \binom{k}{j} \frac{\lambda^{k-j}}{2^k}.$$

This geometric sum converges to a constant since $|\lambda| < 2$ (Lemma B.3). For λ_1 , we have that the constant is proportional to

$$\left(\frac{2}{2 - \lambda_1} \right)^{m_1},$$

where m_1 is the multiplicity of λ_1 . Moreover, because the norm of \mathbf{H} is less than 2 (the maximum eigenvalue) [Gantmacher 1959], we have

$$\vec{C}(\ell) \leq \left(\mathbf{I} - \frac{1}{2} \mathbf{H} \right)^{-1} \vec{F}. \quad \square$$

In Example 1, it is possible to bound $C_i(n)$ by 8 for all i . In the worst case, the number of comparisons could be exponential in the size of the query.

From the previous lemmas, we obtain our main result:

THEOREM B.6. *The expected number of comparisons performed by a minimal DFA for a query q represented by its incidence matrix \mathbf{H} while searching the trie of n random strings is sublinear, and given by*

$$O((\log_2 n)^t n^r),$$

where $r = \log_2 \lambda < 1$, $\lambda = \max_i(|\lambda_i|)$, $t = \max_i(m_i - 1, \text{such that } |\lambda_i| = \lambda)$, and the λ_i s are the eigenvalues of \mathbf{H} with multiplicities m_i .

PROOF. The number of comparisons is proportional to the number of internal nodes plus the number of external nodes visited. In each external node, a constant number of comparisons is performed on average. \square

For NFAs, the same result holds multiplied by the length of the query because we have at most $O(|query|)$ states per active node. However, in this case the incidence matrix is computed using the ϵ -closure of each state. Similarly, the complexity for Patricia trees is the same, as the number of internal nodes is

strictly less (and a subset of the equivalent trie) and checking against the text is equivalent at most to the internal nodes that are skipped.

Because the independent and dependent models are equivalent for random binary tries and random binary suffix trees [Apostolico and Szpankowski 1992], the complexity for a random suffix tree is the same. In practice we use a compact suffix tree. The asymptotic complexity of the algorithm in a compact suffix tree is also the same, because

- the asymptotic complexity of the algorithm in a complete balanced trie with the same number of sistrings is of the same order.
- for every suffix tree, the equivalent compact suffix tree has at most the same number of nodes.

The analysis technique used merges matrix theory with generating functions. By using matrix algebra, it is possible to simplify the analysis of a set of recurrence equations. For example, consider the following problem:

$$\begin{aligned} f_1(n) &= f_1\left(\frac{n}{2}\right) + f_2\left(\frac{n}{2}\right) + c_1, & f_1(1) &= 0, \\ f_2(n) &= \alpha f_2\left(\frac{n}{2}\right) + f_1\left(\frac{n}{2}\right) + c_2, & f_2(1) &= 0, \end{aligned}$$

for n a power of 2. Suppose that $\alpha = 0$, then we can reduce the problem to

$$f_1(n) = f_1\left(\frac{n}{2}\right) + f_1\left(\frac{n}{4}\right) + c_1 + c_2, \quad f_1(2) = f_1(1) = 0.$$

Even this particular case appears to be difficult to solve. However, writing the problem as a matrix recurrence, we obtain

$$\vec{f}(n) = \mathbf{H}\vec{f}\left(\frac{n}{2}\right) + \vec{c},$$

and this recurrence can be solved by simple iteration

$$\vec{f}(n) = \sum_{k=0}^{\log_2 n - 1} \mathbf{H}^k \vec{c}.$$

Decomposing \mathbf{H} in its Jordan normal form, we obtain the exact solution for $\vec{f}(n)$. In fact, we hide all the dependencies inside the matrix until the last step, where known summations that depend on the eigenvalues of \mathbf{H} have to be solved. We have already applied this technique to other problems, like the solution to a fringe analysis type recurrence [Baeza-Yates and Gonnet 1989b; Baeza-Yates 1995].

ACKNOWLEDGMENTS. We are grateful for the helpful comments of Alberto Apostolico, Wojciech Szpankowski, Frank Tompa and the anonymous referees.

REFERENCES

- AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.
- APOSTOLICO, A., AND SZPANKOWSKI, W. 1992. Self-alignments in words and their applications. *J. Algorithms* 13, 446–467.
- BAEZA-YATES, R. A. 1989. Efficient text searching. Ph.D. dissertation. Dept. of Computer Science, Univ. Waterloo, Waterloo, Ont., Canada.
- BAEZA-YATES, R. A. 1995. Fringe analysis revisited. *ACM Comput. Surv.* 27 (Mar.), 109–119.
- BAEZA-YATES, R., AND GONNET, G. H. 1989a. Efficient text searching of regular expressions. In *Proceedings of ICALP'89*. G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., Lecture Notes in Computer Science, vol. 372. Springer-Verlag, New York, pp. 46–62.
- BAEZA-YATES, R. A., AND GONNET, G. H. 1989b. Solving matrix recurrences with applications. Tech. Rep. CS-89-16. Department of Computer Science, Univ. Waterloo.
- BATORY, D. S. 1979. On searching transposed files. *ACM Trans. Datab. Syst.* 4, 531–544.
- DE LA BRIANDAIS, R. 1959. File searching using variable length keys. In *AFIPS Western JCC* (San Francisco, Calif.), pp. 295–298.
- DEVROYE, L. 1982. A note on the average depth of tries. *Computing* 28, 367–371.
- FLAJOLET, P., AND PUECH, C. 1986. Partial match retrieval of multidimensional data. *J. ACM* 33, 2 (Apr.), 371–407.
- FREDKIN, E. 1960. Trie memory. *Commun. ACM* 3, 9 (Sept.), 490–499.
- GALL, Z. 1985. Open problems in stringology. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, eds., volume F12 of NATO ASI Series. Springer-Verlag, New York, pp. 1–3.
- GANTMACHER, F. R. 1959. *The Theory of Matrices*, (2 vols.). Chelsea Publishing Company, New York.
- GONNET, G. H. 1983. Unstructured data bases or very efficient text searching. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23). ACM, New York, pp. 117–124.
- GONNET, G. H. 1988. Efficient searching of text and pictures (extended abstract). Tech. Rep. OED-88-02. Centre for the New OED., Univ. Waterloo, Waterloo, Ont., Canada.
- GONNET, G. H., AND BAEZA-YATES, R. 1991. *Handbook of Algorithms and Data Structures—In Pascal and C*, 2nd ed., Addison-Wesley, Wokingham, UK.
- GONNET, G., COHEN, M., AND BENNER, S. 1992. The all-against-all matching of a major protein sequence database. *Science* 256, 1443–1445.
- HOPCROFT, J. E., AND ULLMAN, J. D. 1979. *Introduction to Automata Theory*. Addison-Wesley, Reading, Mass.
- KNUTH, D. E. 1973. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, Mass.
- MANBER, U., AND BAEZA-YATES, R. 1991. An algorithm for string matching with a sequence of don't cares. *Inf. Proc. Lett.* 37 (Feb.), 133–136.
- MANBER, U., AND MYERS, G. 1990. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, Calif., Jan.). ACM, New York, pp. 319–327.
- MORRISON, D. R. 1968. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. *J. ACM* 15, 4 (Oct.), 514–534.
- OSTROWSKI, A. M., AND SCHNEIDER, H. 1960. Bounds for the maximal characteristic root of a non-negative irreducible matrix. *Duke Math J.* 27, 547–553.
- PITTEL, B. 1986. Paths in a random digital tree: Limiting distributions. *Adv. Appl. Prob.* 18, 139–155.
- RÉGNIER, M. 1981. On the average height of trees in digital search and dynamic hashing. *Inf. Proc. Lett.* 13, 64–66.
- RIVEST, R. 1977. On the worst-case behavior of string-searching algorithms. *SIAM J. Comput.* 6, 669–674.
- SZPANKOWSKI, W. 1993. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comput.* 22, 1176–1198.
- THOMPSON, K. 1968. Regular expression search algorithm. *Commun. ACM* 11, 6 (June), 419–422.
- UKKONEN, E. 1993. Approximate string matching over suffix trees. In *CPM'94* (Padova, Italy, June). Lecture Notes in Computer Science, vol. 520. Springer-Verlag, New York, pp. 240–248.