RE-Tree: An Efficient Index Structure for Regular Expressions

Chee-Yong Chan, Minos Garofalakis, Rajeev Rastogi
Bell Labs, Lucent Technologies
{cychan,minos,rastogi}@research.bell-labs.com

Abstract

Due to their expressive power, Regular Expressions (REs) are quickly becoming an integral part of language specifications for several important application scenarios. Many of these applications have to manage huge databases of RE specifications and need to provide an effective matching mechanism that, given an input string, quickly identifies the REs in the database that match it. In this paper, we propose the RE-tree, a novel index structure for large databases of RE specifications. Given an input query string, the RE-tree speeds up the retrieval of matching REs by focusing the search and comparing the input string with only a small fraction of REs in the database. Even though the RE-tree is similar in spirit to other tree-based structures that have been proposed for indexing multi-dimensional data, RE indexing is significantly more challenging since REs typically represent infinite sets of strings with no well-defined notion of spatial locality. To address these new challenges, our RE-tree index structure relies on novel measures for comparing the relative sizes of infinite regular languages. We also propose innovative solutions for the various RE-tree operations, including the effective splitting of RE-tree nodes and computing a "tight" bounding RE for a collection of REs. Finally, we demonstrate how sampling-based approximation algorithms can be used to significantly speed up the performance of RE-tree operations. Our experimental results with synthetic data sets indicate that the RE-tree is very effective in pruning the search space and easily outperforms naive sequential search approaches.

1 Introduction

Regular expressions (REs) provide an expressive and powerful formalism for capturing the structure of messages, events, and documents. Consequently, they have been used extensively in the specification of a number of languages for important application domains, including the XPath pattern language for XML documents [7], the policy language of the *Border Gateway Protocol* (BGP) for propagating routing information between autonomous systems in the Internet [21], and the UNIX shell's `grep` utility. Many of these applications have to manage large databases of RE specifications and need to provide an effective matching mechanism that, given an input string, quickly identifies all the REs in the database that match it. This "RE retrieval" problem is important for a variety of software components in the middleware and networking infrastructure of the Internet. We list some of these application domains below.

• **XML Filtering.** Information dissemination applications extensively use document-filtering techniques to avoid flooding users with unnecessary information. The key idea is to maintain, for each user, a profile that captures the user's interests/preferences, and transmit to the user only documents that match the profile. While most existing systems for selective dissemination of information typically use simple keywords to specify profiles, the growing momentum of XML as the standard for information exchange on the Internet has recently prompted proposals that use more powerful RE-based languages for expressing user profiles [1]. The primary reason for this is that XML documents are structured, and thus REs provide a succinct syntax for creating more accurate and focused profiles. More specifically, REs can be used to specify matching constraints on the sequence of elements along specific paths in the XML-document tree. Systems for filtering XML documents [1, 5, 11] represent user interests using XPath expressions [7] which incorporate a restricted form of REs.

• **XML Routing.** XML routers route XML documents based on their content. By directing XML traffic to the least loaded-application server that is capable of processing a request, an XML router can balance load and boost Web site performance. Again, an RE-based language often provides a succinct and convenient tool for expressing the set of rules for routing incoming XML traffic. As an example, the Intel NetStructure XML Accelerator product [9] uses XPath 1.0 for specifying the rules for directing XML transactions to the appropriate application servers.

• **XML Classification.** As XML becomes a popular standard for exchanging data on the Web, the volume of XML-encoded documents on the Web is expected to grow rapidly in coming years. Document Type Descriptors (DTDs) are RE-based expressions that serve the role of schemas spec-

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

ifying the internal structure of XML documents. However, DTDs are *not* mandatory and an XML document may not always have an accompanying DTD. Thus, in many cases, it becomes necessary to identify the subset of a "universe" of known DTDs that match a new XML document. Identifying matching DTDs for XML documents, besides enabling document typing and classification, is also crucial for the efficient storage and effective querying of XML data [10].

• **BGP Routing.** In the BGP4 Internet routing protocol [21], routers transmit advertisements to neighboring routers; each advertisement contains a destination IP address reachable from the router as well as the sequence of routing systems on the path from the router to the destination. In case a router receives multiple advertisements (from different neighbors) for the same destination, the BGP4 policy language allows for *priorities* to be assigned to advertisements based on the corresponding routing-system sequences. (The advertisement with the highest priority eventually determines the route to the destination). The BGP4 policy language essentially allows for network administrators to specify a set of REs (on routing-system sequences) and associate priorities with each RE. Advertisements containing routing-system sequences that match a specific RE are then assigned the corresponding priority.

Another application involves finding the root-cause of a fault in a network by matching the sequence of events (generated when a network fault occurs) to a database of RE patterns that capture the association between high-level fault patterns and their root causes. Clearly, the above-mentioned applications have a critical need for a scalable and efficient structure that can index large numbers of REs and that can quickly retrieve all REs matching a given input string. For the XML-filtering application, the input string is a path in the XML document and the REs are user profiles, while for BGP routing, the input string is the routing system sequence in an advertisement and the REs are the BGP policies.

In this paper, we propose the *RE-tree*, a novel index structure for performing fast retrievals of REs that match a given input string. The RE-tree, to the best of our knowledge, is the *first* truly scalable index structure that can handle the storage and retrieval of REs in their full generality. The only prior work along these lines that we are aware of are indexing schemes for filtering XML documents based on XPath expressions [1, 5, 11]. However, while the XPath language allows rich patterns with tree structure to be specified, it lacks the full expressive power of REs (e.g., XPath does not permit the RE operators $*$, $|$ and $\cdot$ to be arbitrarily nested), and thus extending XFilter to handle general REs will most likely be difficult. Further, these XPath-based methods are designed for indexing main-memory resident data; in contrast, REs in the RE-tree are organized hierarchically (in a manner similar to the R-tree [14]), and thus the RE-tree is well-suited for disk-resident data.

The task of indexing REs is challenging because REs typically represent *infinite* sets with no well-defined notion of spatial locality. While indexing of documents (which are essentially a finite set/bag of elements) has been extensively studied by the IR community [3], we are not aware of any indexing techniques for infinite sets. The RE-tree employs a number of novel and sophisticated techniques for indexing infinite regular languages, which we list below.

- **Novel Measures for Comparing the Relative Sizes of Infinite Regular Languages.** We develop two novel measures for the size of an infinite regular language – the first counts the number of strings in the language that are less than or equal to a certain size, and the second *information-theoretic* measure is based on the cost (in bits) of encoding random samples in the language. The latter measure is inspired by a general observation in information theory that the cost of encoding a random element of a set is proportional to the set's size.
- **Novel Algorithms for Splitting and Generalizing a Set of REs.** Similar to the R-tree, when an RE-tree index node overflows, the set of REs in the node are split and two new compact parent REs that are more general than the two subsets of REs (due to the split), are computed. We show that both splitting and generalizing a set of REs are NP-hard problems, and present heuristics for these operations in the context of the RE-tree.
- **Novel Sampling-Based Approximation Algorithms for Speeding RE-tree Operations.** Specifically, we show how random samples of RE languages can be used to efficiently compute *approximate* counts (of the number of strings with a fixed length in the language) and samples for unions/intersections of regular languages. These counts and samples are important for the RE-tree split and generalize operations. Also, we devise a novel practical algorithm that employs a combination of dynamic programming and sampling to compute counts/samples for an RE using its non-deterministic finite automaton representation. Previous algorithms computed these after converting the RE to a deterministic finite automaton (see [17]), which can be fairly expensive.

To measure the effectiveness of the RE-tree in retrieving the set of REs that match an input query string, we conducted an extensive experimental study with synthetic data sets. Our experimental results indicate that the RE-tree can drastically reduce the number of RE-comparison operations and easily outperforms naive sequential-search approaches, improving the overall search performance by a factor of approximately 2 on the average.

The remainder of this paper is organized as follows. We first present an overview of the RE-tree in Section 2 and identify the key design issues. Section 3 describes some fundamental algorithms for counting and sampling regular languages. Building on top of these algorithms, we propose two different measures for comparing the relative sizes of infinite regular languages in Section 4. Section 5 then describes in detail our algorithms for the various RE-tree operations. In Section 6, we present the results of our experimental study comparing the RE-tree against sequential-search approaches. Finally, Section 8 concludes the paper with some directions for future research.

Due to space constraints, we do not include proofs of theorems in the body of the paper; the complete details can be found in the full version of this paper [6].

2 Overview of RE-trees

An RE-tree indexes a large collection of REs such that, given an arbitrary input string w, REs in the collection that match w can be retrieved quickly and efficiently. In this section, we first present an overview of the RE-tree index structure. Although the design principles behind the RE-tree are similar in spirit to those in the R-tree spatial index [14], the application of these principles to indexing REs actually reveals a number of interesting algorithmic issues and tradeoffs that require novel techniques. We identify these new design issues in Section 2.2, deferring the details of our solutions to Section 5. Table 1 summarizes some of the key notation used in this paper.

Symbol	Description
Σ	Alphabet over which REs are defined
$\delta()$	Automaton transition function
$\|M\|$	Number of states in automaton M
$L(M)$	Language of automaton M (i.e., strings accepted by M)
$L_i(M)$	Set of length-i strings accepted by automaton M
$\|L(M)\|$	Measure for size of $L(M)$
$M_1 \cup M_2$	Automaton that accepts strings in $L(M_1) \cup L(M_2)$
$M_1 \cap M_2$	Automaton that accepts strings in $L(M_1) \cap L(M_2)$
m	Minimum occupancy for each RE-tree node
α	Maximum number of states for a bounding automaton

Table 1: Notation.

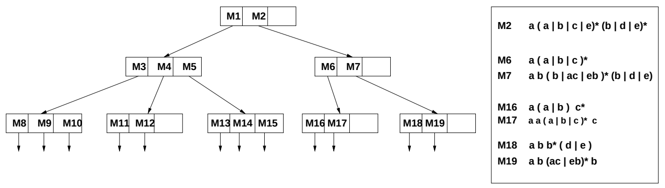
2.1 Index Structure

An RE-tree is a dynamic, height-balanced, hierarchical index structure where leaf nodes contain data entries corresponding to the indexed REs, and internal nodes contain "directory" entries that point to nodes at the next level of the index. Specifically, each leaf node entry is of the form (id, M), where id is the unique identifier of an RE R and M is a *finite automaton* representing R [15]. Each internal node stores a collection of finite automata; and each node entry is of the form (M, ptr), where M is a finite automaton and ptr is a pointer to some node N (at the next level) such that the following *containment property* is satisfied: If for a set of automata $\mathcal{M}$, $L(\mathcal{M}) = \bigcup_{M_i \in \mathcal{M}} L(M_i)$, and $\mathcal{M}(N)$ denotes the collection of automata contained in node N, then $L(\mathcal{M}(N)) \subseteq L(M)$. We refer to the automaton M as the *bounding automaton* for $\mathcal{M}(N)$. The containment property is key to improving the search performance of hierarchical index structures like RE-trees: if a query string w is not contained in $L(M)$, then it follows that $w \notin L(M_i)$ for all $M_i \in \mathcal{M}(N)$. As a result, the entire subtree rooted at N can be pruned from the search space. Clearly, the closer $L(M)$ is to $L(\mathcal{M}(N))$, the more effective this search-space pruning will be.

In general, there are an infinite number of bounding automata for $\mathcal{M}(N)$ with different degrees of precision from the least precise bounding automaton with $L(M) = \Sigma^*$ to the most precise bounding automaton, referred to as the *minimal bounding automaton*, with $L(M) = L(\mathcal{M}(N))$.

Since the storage space for an automaton is dependent on its complexity (in terms of the number of its states and transitions), there is a space-precision tradeoff involved in the choice of a bounding automaton for each internal node entry[1]. Thus, even though minimal bounding automata result in the best pruning due to their tightness, it may not be desirable (or even feasible) to always store minimal bounding automata in RE-trees since their space requirement can be too large (possibly exceeding the size of an index node), thus resulting in an index structure with a low fan-out. Therefore, to maintain a reasonable fan-out for RE-trees, we impose a space constraint on the maximum number of states (denoted by α) permitted for each bounding automaton in internal RE-tree nodes.

The automata stored in RE-tree nodes are, in general, Non-deterministic Finite Automata (NFAs) with a minimum number of states [15]. Also, for space efficiency, we require that each individual RE-tree node contains at least m entries. An example RE-tree is illustrated in Figure 1, where only the top three levels of internal nodes are shown. Each internal node has between 2 and 3 entries, with each M_i representing a bounding automaton; the details of some of these automata are shown in the table on the right in the form of REs (for conciseness). Note that each pair of parent-child node entries satisfies the containment property.



M2	a (a | b | c | e)* (b | d | e)*
M6	a (a | b | c)*
M7	a b (b | ac | eb)* (b | d | e)
M16	a (a | b) c*
M17	a a (a | b | c)* c
M18	a b b* (d | e)
M19	a b (ac | eb)* b

Figure 1: RE-tree Example.

2.2 Design Issues

At a high level, RE-trees are conceptually similar to other hierarchical, spatial index structures, like the R-tree [14]; this is also true for RE-tree search and maintenance algorithms. RE-tree search simply proceeds top-down along (possibly) multiple paths whose bounding automaton accepts the input string; RE-tree updates try to identify a "good" leaf node for insertion and can lead to node splits (or, node merges for deletions) that can propagate all the way up to the root (details can be found in [6]). There

[1]This is in contrast to R-trees, where the space of a bounding rectangle is independent of its precision, and therefore the bounding rectangles in R-trees are all minimal bounding rectangles.

is, however, a fundamental difference between the RE-tree and the R-tree in the indexed data types: regular languages are much more complex objects than multi-dimensional rectangles. This difference mandates the development of novel algorithmic solutions for the core RE-tree operations. Our main goal for the RE-tree is to optimize search performance, and the key guiding principle for achieving this goal is to keep each bounding automaton M in every internal node as "tight" as possible. Thus, if M is the bounding automaton for $\mathcal{M}(N)$, then $L(M)$ should be as close to $L(\mathcal{M}(N))$ as possible. More specifically, the three core new problems that we need to address in the RE-tree context can be defined as follows (the algorithms shown in the parentheses correspond to our solutions and are described in detail later in the paper).

(P1) Selection of an optimal insertion node (Algorithm CHOOSEBESTFA**).** The goal here is to choose an insertion path for a new RE that leads to "minimal expansion" in the bounding automaton in each internal node. Thus, given the collection of automata $\mathcal{M}(N)$ in an internal index node N and a new automaton M, we need to find the optimal $M_i \in \mathcal{M}(N)$ to insert M such that $|L(M_i) \cup L(M)| - |L(M_i)|$ is minimum (or equivalently, $|L(M_i) \cap L(M)|$ is maximum).

(P2) Computing an optimal node split (Algorithm SPLITFA**).** When splitting a set of REs during an RE-tree node-split, we seek to identify a partitioning that results in the minimal amount of "covered area" in terms of the languages of the resulting partitions. More formally, given the collection of automata $\mathcal{M} = \{M_1, M_2, \cdots, M_k\}$ in an overflowed index node, we want to find the optimal partition of $\mathcal{M}$ into two disjoint subsets $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $|\mathcal{M}_1| \geq m$, $|\mathcal{M}_2| \geq m$ and $|L(\mathcal{M}_1)| + |L(\mathcal{M}_2)|$ is minimum.

(P3) Computing an optimal generalized automaton (Algorithm GENERALIZEFA**).** During insertions, node-splits, or node-merges, we need to be able to identify a bounding automaton for a set of REs that does not cover too much "dead space". Thus, given a collection of automata $\mathcal{M}$, we seek to find the optimal generalized automaton M such that $|M| \leq \alpha$, $L(\mathcal{M}) \subseteq L(M)$ and $|L(M)|$ is minimum.

The goal of the above operation specifications is to maximize the pruning during search by keeping bounding automata tight. In (P1), the automaton M_i for which $L(M_i)$ expands the least due to the insertion of M is chosen to insert M. The set of automata $\mathcal{M}$ are split into two tight clusters in (P2), while in (P3), we are interested in finding the most precise automaton covering the set of automata in $\mathcal{M}$ and with no more than α states. As discussed earlier, the restriction on the number of states is imposed to keep the fan-out of each node high since this is critical to improving search performance. Essentially, (P3) represents the space-precision tradeoff for bounding automata in RE-trees.

Note that while (P3) is unique to RE-trees, both (P1) and (P2) have their equivalents in R-trees. Examples of heuristics that have been proposed for (P1) in R-trees include minimizing the increase in the area of the minimum bounding rectangle (MBR) [14], and minimizing the overlap among the MBRs within the node [4]. The heuristics for (P2) in R-trees are similar to those for (P1); examples include minimizing the total area of the two MBRs [14], and minimizing the area of the intersection between the two MBRs [4]. The aim of all these heuristics is to minimize the number of visits to nodes that do not lead to any qualifying data entries.

Although the "MBRs" in RE-trees (which correspond to regular languages) are very different from the MBRs in R-trees, the intuition behind minimizing the area of MBRs (total area or overlapping area) in R-trees should be effective for RE-trees as well. The counterpart for area in an RE-tree is $|L(M)|$, the size of the regular language for M. However, since a regular language is generally an infinite set, we need to develop new measures for the size of a regular language or for comparing the sizes of two regular languages.

3 Counting and Sampling $L_n(M)$

Before defining size measures for infinite regular languages in the next section, we first describe two fundamental algorithms for counting and sampling that form the basis of our definitions. The first algorithm counts the number of length-n strings accepted by an automaton M (i.e., $|L_n(M)|$), while the second algorithm generates a random sample of $L_n(M)$. We begin by presenting the algorithms, originally proposed in [17], for the simpler case when M is a DFA. We then present novel practical counting and sampling algorithms for the case when M is an NFA.

Let $\beta(s, n)$ denote the number of distinct length-n paths that can be generated using an automaton M (with start state s_0) from state s to any accepting state.

3.1 Algorithms for DFAs

For the case where M is a DFA, $|L_n(M)| = \beta(s_0, n)$. Clearly, for $n = 0$, we have $\beta(s, 0) = 1$ if s is an accepting state, and 0 otherwise. For $n \geq 1$, $\beta(s, n)$ can be computed recursively as follows [17]: $\beta(s, n) = \sum_{x \in \Sigma,\ t = \delta(s,x)} \beta(t, n-1)$. Thus, dynamic programming can be used to compute $\beta(s, i)$ for all states s, first for $i = 0$ and then for successively increasing values of i (until $i = n$) using the computed results for $i - 1$. Since each $\beta(s, i)$ is computed by considering all states reachable from state s with a single transition, each $\beta(\cdot)$ value can be computed in $O(\min\{|\Sigma|, |M|\})$ time. Thus, since there are $|M|n$ values in $\beta(\cdot)$, it follows that there is an $O(n|M|\min\{|\Sigma|, |M|\})$ algorithm to compute $|L_i(M)|$ for all $1 \leq i \leq n$.

We now explain how to generate a random string from $L_n(M)$ based on the computed values of $\beta(\cdot)$. Consider the collection of all length-n strings that can be generated from some state s to any accepting state. Then the probability that a randomly selected string from this collection has $x \in \Sigma$ as the first symbol is given by $\frac{\beta(t,n-1)}{\beta(s,n)}$, where $t = \delta(s, x)$. Thus, a uniformly random string from $L_n(M)$ can be generated iteratively by randomly choosing a transition at each state beginning from the start state as

```
Algorithm GenRandomString ($M, s, n$)
Input:   $M$ is a DFA, $s$ is a state in $M$.
         $n$ is a length parameter, $n \geq 1$.
Output:  A random string in $L_n(M)$.
1)  for $i := n$ downto $1$ do
2)      Select transition $x$ out of $s$ from $\Sigma$ with probability
        $\frac{\beta(t,i-1)}{\beta(s,i)}$, where $t = \delta(s,x)$;
3)      Let $y_i$ be the selected transition;
4)      $s := \delta(s, y_i)$;
5)  return $y_n \cdot y_{n-1} \cdots y_1$;
```

Figure 2: Algorithm for Generating a Random String in $L_n(M)$.

shown in Figure 2. Algorithm GenRandomString in the figure when invoked with input parameter $s = s_0$ returns a random string from $L_n(M)$. It is straightforward to show that if $s_0, s_n, s_{n-1}, \ldots, s_1$ denote the sequence of states visited by the algorithm, then the probability that a string $w \in L_n(M)$ is returned by Algorithm GenRandomString is $\frac{\beta(s_n,n-1)}{\beta(s_0,n)} \times \frac{\beta(s_{n-1},n-2)}{\beta(s_n,n-1)} \times \cdots \times \frac{\beta(s_1,0)}{\beta(s_2,1)} = \frac{1}{\beta(s_0,n)} = \frac{1}{|L_n(M)|}$. Note that the time complexity of the algorithm is $O(n)$ if the function $\beta(\cdot)$ has been precomputed. Also, a uniform random sample of $L_n(M)$ of size k can be generated by repeatedly invoking GenRandomString until it has returned k distinct strings.

3.2 Algorithms for NFAs

For the case where M is an NFA[2], $\beta(s_0, n) \geq |L_n(M)|$ since there can be multiple accepting paths in an NFA for a given string. However, the problem of computing $|L_n(M)|$ for an NFA M is #P-complete.

An unbiased estimator for $|L_n(M)|$ can be computed as follows. Let p be a uniformly generated accepting path of length n in M, and let w be the string labelling p. Then $|L_n(M)| \approx \frac{\beta(s_0,n)}{q}$, where q is the number of accepting paths of w in M. Essentially, the unbiased estimator is computed by scaling down the total number of length-n accepting paths in M with the count of the number of accepting paths for a randomly generated length-n string. Note that since there is a one-to-one correspondence between an accepting string and an accepting path in a DFA, Algorithm GenRandomString (in Figure 2) can be used to generate the path p. The number of accepting paths for w can be derived by traversing M with w.

However, as pointed out in [17], the above estimator can have a very large standard deviation. Although [17] proposes a more accurate, randomized algorithm for approximating $|L_n(M)|$, it is not very useful in practice due to its high time complexity of $O(n^{log(n)})$.

In Figure 3, we present a novel and practical algorithm for approximately counting strings in $L_n(M)$ for an NFA M that has a lower time complexity of $O(n^2|M|^2 \min\{|\Sigma|, |M|\})$. Similar to earlier approaches, the algorithm uses dynamic programming to compute counts $\beta(s, i)$. However, instead of computing in each $\beta(s, i)$, the number of length-i accepting paths from state

[2]The ϵ-transitions in M are assumed to be eliminated.

```
Algorithm CountStrings ($M, n$)
Input:   $M$ is an NFA, $n$ is a length parameter, $n \geq 1$.
Output:  Counts $\beta$ of accepting strings of length less than or equal
         to $n$ for each state in $M$.
1) for each state $s \in M$ do $\beta(s, 0) := 0$;
2) for each accepting state $s \in M$ do $\beta(s, 0) := 1$;
3) for $i := 1$ to $n$ do
4)   for each state $s \in M$ do
5)     $\beta(s, i) := 0$;
6)     for each symbol $x$ such that there is a transition out of $s$ on $x$ do
7)       Let $t_1, \ldots, t_l$ be the resulting states for transitions out of $s$
         on symbol $x$;
8)       $\beta(s, i) := \beta(s, i) + \beta(t_1, i-1)$;
9)       for $j := 2$ to $l$ do
10)        Let $r$ be a random sample of $L_{i-1}(M, t_j)$ (generated by
           repeatedly invoking GenRandomString ($M, t_j, i-1$)
           a fixed number of times);
11)        $\beta(s,i) := \beta(s,i) + \beta(t_j, i-1) \times \frac{|r - (\cup_{k=1}^{j-1} L_{i-1}(M,t_k))|}{|r|}$;
12) return $\beta()$;
```

Figure 3: Algorithm for Computing Approximate Count of $L_n(M)$.

s, the algorithm adjusts this total count of accepting paths by eliminating duplicate paths (recall that in an NFA, there can be multiple accepting paths that correspond to the same string). Thus, in our algorithm, each $\beta(s, i)$ captures $|L_i(M, s)|$, the number of length-i accepting strings in M from state s, more accurately.

When computing $\beta(s, i)$ in Steps 5–11, Algorithm CountStrings subtracts from the total count of accepting paths $\sum_{x \in \Sigma,\ t \in \delta(s,x)} \beta(t, i-1)$ (due to the dynamic programming relationship)[3], the number of paths for the same string that are counted multiple times. Thus, the key problem is estimating the number of these duplicate length-i accepting paths from s, which we solve as follows. First, observe that it is not possible for two paths to be identical if they are due to transitions associated with different symbols – thus, duplicate paths can only result due to transitions out of s on the same symbol. Consequently, we only need to perform duplicate elimination from among the set of paths whose first transitions have the same symbol.

Suppose that $t_1, t_2, \ldots, t_l$ are the states due to transitions out of s on symbol x. Note that each $\beta(t_j, i-1)$ is an estimate of $|L_{i-1}(M, t_j)|$ which is the number of distinct accepting paths of length $i-1$ from state t_j. However, for a pair of states t_j, t_k, it is possible that $L_{i-1}(M, t_j)$ and $L_{i-1}(M, t_k)$ have strings in common. The two paths from t_j and t_k for each common string are identical, and with the two transitions from s to t_j and t_k (on symbol x), result in duplicate paths from s. Our strategy for eliminating such duplicates is to, for each t_j, subtract from $\beta(t_j, i-1)$, the number of strings in $L_{i-1}(M, t_j)$ that have already been counted earlier in $L_{i-1}(M, t_1), L_{i-1}(M, t_2), \ldots, L_{i-1}(M, t_{j-1})$. In the algorithm, we estimate the number of these strings by generating a random sample r of $L_{i-1}(M, t_j)$ and scaling $\beta(t_j, i-1)$ by the fraction of r not contained in $\cup_{k=1}^{j-1} L_{i-1}(M, t_k)$. Thus, $\beta(t_j, i-1) * \frac{|r - (\cup_{k=1}^{j-1} L_{i-1}(M,t_k))|}{|r|}$

[3]Note that for an NFA M, $\delta(s, x)$ is a set of states.

is a good estimate of the count of strings in $L_{i-1}(M, t_j)$ not previously counted in $\cup_{k=1}^{j-1} L_{i-1}(M, t_k)$, and is added to $\beta(s, i)$ in Step 11.

We must point out that each string returned by GENRANDOMSTRING (in Step 10) is based on the previously computed counts $\beta()$ which only estimate the number of accepting strings. Thus, the sample r may not be a uniform random sample. Further, GENRANDOMSTRING assumes that M is a DFA and so it needs to be modified when M is an NFA. Details of the required modifications for NFAs can be found in [6].

The overhead of generating the random sample r of length-$(i-1)$ strings for states $t_1, t_2, \ldots, t_l$ increases the time complexity of the dynamic programming algorithm by a factor of $n|M|$ in the worst case, resulting in a worst-case time complexity of $O(n^2|M|^2 \min\{|\Sigma|, |M|\})$ for the algorithm. This is because $i \leq n$ and $l \leq |M|$. Note that the algorithm can also handle DFAs very efficiently and its time complexity reduces to $O(n|M| \min\{|\Sigma|, |M|\})$ if M is a DFA. The reason for this is that for a DFA, l is always 1 and as a result, random samples r do not need to be generated.

4 Size Measures for Infinite Regular Languages

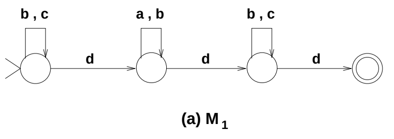
Estimating the size of a regular language $L(M)$ is much more difficult than computing $|L_n(M)|$ since unlike $L_n(M)$, $L(M)$ can be an infinite set. In this section, we define two different measures that attempt to capture the size of infinite regular languages. We note that the techniques presented here have applications beyond indexing REs and can also be used for estimating the selectivities of REs, clustering REs, etc.

4.1 Definitions

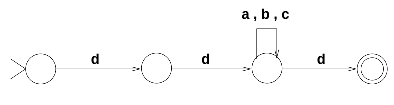
Before we proceed to define our measures for the size of $L(M)$, we identify certain desirable properties for such measures. First, we note that it does not make sense to actually count the number of strings in an infinite regular language. However, we do know that while it is impossible to assign a precise integer size to an infinite language, not all infinite languages are equal with respect to size. For example, the language for RE $(a|b)^*$ is *larger than* the language for $a(a|b)^*$ (since the latter is a proper subset of the former). Thus, we would like to define a measure for the size of a language that reflects our intuition of the "larger than" relationship between languages. We denote this measure of the size of $L(M)$ by $|L(M)|$.

We can formalize our intuition of the "larger than" relationship between regular languages as follows.

Definition 4.1: *For a pair of automata M_i, M_j, we say that $L(M_i)$ is larger than $L(M_j)$ iff there exists a positive integer N such that for all $k \geq N$, $\sum_{l=1}^{k} |L_l(M_i)| > \sum_{l=1}^{k} |L_l(M_j)|$.* ■



(a) M_1



(b) M_2

Figure 4: Examples of Automata.

The various definitions for measure $|L(M)|$ of the size of $L(M)$ that we present in the following subsections attempt to capture this "larger than" relationship. Specifically, for a pair of automata M_i, M_j, if $L(M_i)$ is larger than $L(M_j)$, then $|L(M_i)| > |L(M_j)|$.

4.2 Max-Count Measure

An obvious measure for the size of a regular language $L(M)$ is to count $L(M)$ up to some maximum length λ, i.e., $|L(M)| = |L_1(M)| + |L_2(M)| + \cdots + |L_\lambda(M)|$. Clearly, the larger we choose λ to be, the more effective we can expect the Max-Count measure to be at reflecting the larger than relationship. However, for very large values of λ, the Max-Count measure may not be practical. Thus, a good compromise is to set λ to be equal to or slightly greater than $|M|$. This ensures that strings due to all accepting states and traversing a fair proportion of paths/cycles in M are counted in $|L(M)|$. This measure is particularly useful for applications where the maximum length of the query strings is known and its value is not too large. For such cases, λ can be set to the maximum query length.

4.3 Minimum Description Length (MDL) Based Measure

For applications where information about the maximum length of the query strings is not known apriori, using the Max-Count measure can be problematic due to its sensitivity to the maximum length parameter value λ.

Example 4.2: Consider the two automata M_1 and M_2 in Figure 4. Even though $|L_n(M_1)|$ is greater than $|L_n(M_2)|$ for small values of n, $L(M_2)$ is larger than $L(M_1)$ since $|L_n(M_2)|$ grows much faster than $|L_n(M_1)|$ as n increases. In fact, one can show that $|L_{n+3}(M_1)| = (n+1)(n+2)2^{(n-1)}$ and $|L_{n+3}(M_2)| = 3^n$ for $n > 0$. The crossover point for $|L(M_1)|$ and $|L(M_2)|$ using the Max-Count measure occurs for a maximum length of $\lambda = 16$, that is, $\sum_{l=1}^{\lambda} |L_l(M_2)| > \sum_{l=1}^{\lambda} |L_l(M_1)|$ for $\lambda \geq 16$. Thus, if the value of λ for the Max-Count measure is set to be less than 16, then $L(M_1)$ would be considered, incorrectly, to be larger than $L(M_2)$. ■

To obtain a more robust measure of the size of infinite languages, we present an alternate metric that attempts to address some of the shortcomings of the Max-Count measure and is based on using Rissanen's *Minimum Descrip-*

tion Length (MDL) principle [20]. The MDL principle essentially provides an information-theoretic definition of the optimal "theory/model" that can be inferred from a set of data examples; and it has been applied in a variety of problems (e.g., constructing decision trees [19], learning common patterns in a set of strings [18], inferring DTDs from a collection of XML data [13]). An important observation made in [13] is that for two given REs R_i and R_j, if R_i defines a larger language than R_j (i.e., R_i is less precise than R_j with respect to the same collection of input data sequences), then the cost of encoding an input data sequence using R_i is likely to be higher than using R_j. This observation is consistent with information theory since, in general, more bits are needed to specify an item that comes from a larger collection of items.

Our use of the MDL principle to define a measure of the size of a regular language is also inspired by a similar observation and is based on the following intuition: Given two DFAs M_i and M_j, if $L(M_i)$ is larger than $L(M_j)$, then the *per-symbol-cost* of an MDL-based encoding of a random string in $L(M_i)$ using M_i is very likely to be higher than that of a string in $L(M_j)$ using M_j. The *per-symbol-cost* of encoding a string $w \in L(M)$ is essentially the ratio of the cost of an MDL-based encoding of w using M (defined below) to the length of w. Therefore, a reasonable measure for the size of a regular language $L(M)$ is the expected per-symbol-cost of an MDL-based encoding for a random sample of strings in $L(M)$. Let $\mathcal{MDL}(M, w)$ denote the cost of an MDL-based encoding of a string $w \in L(M)$ using M and let r be a random sample of $L(M)$. Then, the MDL-based measure for the size of $L(M)$ is as follows[4].

$$|L(M)| = \frac{1}{|r|} \sum_{w \in r} \frac{\mathcal{MDL}(M, w)}{|w|} \qquad (1)$$

We next define the cost of an MDL-encoding of w using M. Suppose that $w = w_1.w_2.\cdots.w_n \in L(M)$ and $s_0, s_1, \ldots, s_n$ is the unique sequence of states visited by w in M. Then,

$$\mathcal{MDL}(M, w) = \sum_{i=0}^{n-1} log_2(n_i) \qquad (2)$$

where each n_i denotes the number of transitions out of state s_i, and $log_2(n_i)$ is the number of bits required to specify the transition out of state s_i. Since the above formulation is based on counting the number of transitions, to obtain an accurate measure of the size of an infinite language, it is important that M does not contain any "non-essential" transitions (i.e., transitions that are not part of an accepting path that involves at least one cycle[5]). Thus, M should be a minimal-state DFA without any non-essential transitions. The following example illustrates how the MDL-based encoding costs are computed.

Example 4.3: Consider the two automata M_1 and M_2 in Figure 4, and two length-10 strings $w_1 = bbbdbbdbbd \in L(M_1)$ and $w_2 = ddbbbbbbbd \in L(M_2)$. The per-symbol-costs of encoding w_1 and w_2 using M_1 and M_2, respectively, are as follows:

$$\frac{\mathcal{MDL}(M_1, w_1)}{|w_1|} = \frac{10 \times log_2(3)}{10} \approx 1.5850 \text{ and}$$

$$\frac{\mathcal{MDL}(M_2, w_2)}{|w_2|} = \frac{(2 \times 0) + (8 \times log_2(4))}{10} \approx 1.6000$$

The computed per-symbol-costs correctly indicate that $L(M_2)$ is larger than $L(M_1)$. ∎

[4] $|w|$ denotes the length string w.

[5] For example, in Figure 7(a), if the state labeled 5 does not have a a self-loop transition, then both the a-transition from state 2 to state 5 as well as the b-transition from state 5 to state 6 are non-essential. The removal of such non-essential transitions from an automaton does not affect the infiniteness of its language.

We still need to address the issue of generating the random sample r of $L(M)$ to encode. This can be carried out by selecting two values λ and θ, both slightly greater than $|M|$ and for a desired random sample of size k, generating $\frac{k|L_i(M)|}{\sum_{l=\lambda}^{\lambda+\theta-1}|L_l(M)|}$ random strings from each $L_i(M)$, $\lambda \leq i \leq \lambda + \theta - 1$ (using Algorithm GENRANDOMSTRING). By sampling from each $L_i(M)$ in proportion to its count, we ensure that paths in the automaton that are traversed more frequently by accepting strings have a greater likelihood of being included in the sample r.

5 Algorithms for RE-Tree Operations

Now that we have robust measures for comparing the relative sizes of infinite regular languages, we are in a position to present details of the three RE-tree operations for choosing the best automaton, splitting a set of automata and computing a generalized automaton (problems (P1), (P2) and (P3) from Section 2.2). The algorithms for the RE-tree operations presented in this section perform a number of standard operations on automata like union, intersection and conversion of an NFA to a DFA. Efficient algorithms for these automata operations can be found in [15]. Note that while it is possible to compute the union/intersection of a pair of automata M_i, M_j in time proportional to $|M_i||M_j|$, the worst-case time complexity of converting an NFA M to a DFA can be exponential in $|M|$.

5.1 Algorithm CHOOSEBESTFA

From among the automata in $\mathcal{M}(N)$ contained in a node N, Algorithm CHOOSEBESTFA returns the automaton M_i for which $|L(M_i) \cap L(M)|$ is maximum. For each $M_i \in \mathcal{M}(N)$, the algorithm constructs the DFA $M \cap M_i$ and computes $|L(M \cap M_i)|$ using either the Max-Count or the MDL metric from the previous section. The automaton M_i for which the language $L(M \cap M_i)$ is the largest is chosen for inserting M into the RE-tree.

5.2 Algorithm SPLITFA

For a set of automata $\mathcal{M} = \{M_1, \ldots, M_k\}$, Algorithm SPLITFA partitions $\mathcal{M}$ into $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $|\mathcal{M}_1| \geq m$, $|\mathcal{M}_1| \geq m$ and $|L(\mathcal{M}_1)| + |L(\mathcal{M}_2)|$ is minimum. Unfortunately, it can be shown that even if $L(M_i)$ for every $M_i \in \mathcal{M}$ is finite, the problem of optimally splitting $\mathcal{M}$ is NP-hard (reduction from clique). Thus, one can expect that for infinite languages, the problem is even more difficult.

```
Algorithm SPLITFA ($\mathcal{M}$)
Input: $\mathcal{M}$ is a set of automata to be split.
Output: Two sets of automata $\mathcal{M}_1$ and $\mathcal{M}_2$ resulting from the split.
1) Let $M_i, M_j \in \mathcal{M}$ be the pair of automata such that $|L(M_i \cup M_j)|$
      $-|L(M_i \cap M_j)|$ is maximum;
2) $\mathcal{M}_1 := \{M_i\}, \mathcal{M}_2 := \{M_j\}$;
3) $\mathcal{M} := \mathcal{M} - \{M_i, M_j\}$;
4) while ($\mathcal{M} \neq \emptyset$) do
5)     if ($|\mathcal{M}| = m - |\mathcal{M}_1|$) then
6)         $\mathcal{M}_1 := \mathcal{M}_1 \cup \mathcal{M}$;
7)         break;
8)     if ($|\mathcal{M}| = m - |\mathcal{M}_2|$) then
9)         $\mathcal{M}_2 := \mathcal{M}_2 \cup \mathcal{M}$;
10)        break;
11)    Let $M_i \in \mathcal{M}$ be the automaton for which $\min\{|L(\mathcal{M}_1 \cup \{M_i\})|$
           $-|L(\mathcal{M}_1)|, |L(\mathcal{M}_2 \cup \{M_i\})| - |L(\mathcal{M}_2)|\}$ is minimum;
12)    if ($|L(\mathcal{M}_1 \cup \{M_i\})| - |L(\mathcal{M}_1)| \leq |L(\mathcal{M}_2 \cup \{M_i\})| -$
       $|L(\mathcal{M}_2)|$) then
13)        $\mathcal{M}_1 := \mathcal{M}_1 \cup \{M_i\}$;
14)    else
15)        $\mathcal{M}_2 := \mathcal{M}_2 \cup \{M_i\}$;
16)    $\mathcal{M} := \mathcal{M} - \{M_i\}$;
17)    return ($\mathcal{M}_1, \mathcal{M}_2$);
```

Figure 5: Algorithm for Splitting a Set of Automata.

Theorem 5.1: *Given finite sets of elements $S_1, \ldots, S_n$, the problem of partitioning them into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ (each containing at least m elements) such that $|(\cup_{S_i \in \mathcal{S}_1} S_i)| + |(\cup_{S_i \in \mathcal{S}_2} S_i)|$ is minimum, is NP-hard.* ∎

Since the problem of computing the optimal partitioning of $\mathcal{M}$ is NP-hard, we resort to heuristics to generate the two sets $\mathcal{M}_1$ and $\mathcal{M}_2$. Figure 5 contains the steps of the SPLITFA algorithm for splitting $\mathcal{M}$ into two compact well-separated subsets. Algorithm SPLITFA is similar in spirit to the Quadratic Split algorithm from [14] – however, instead of trying to minimize the area of MBRs, our splitting algorithm attempts to reduce the size of regular languages. The algorithm begins by picking as seeds (in Step 1), the two automata from $\mathcal{M}$ whose languages have large non-overlapping portions, and then greedily assigns each remaining automaton M_i in $\mathcal{M}$ to the set whose language increases the least due to the addition of M_i.

Note that Step 1 requires $O(|\mathcal{M}|^2)$ automata intersection, union and size measure computations to be performed, one for each pair of automata in $\mathcal{M}$. For efficiency purposes, the algorithm caches DFAs for $\mathcal{M}_1$ and $\mathcal{M}_2$ as well as $|L(\mathcal{M}_1)|$ and $|L(\mathcal{M}_2)|$ between successive iterations of the while loop. Thus, in Step 11, the algorithm performs $2(|\mathcal{M}| - i)$ automata union and size measure computations in the i^{th} iteration of the while loop, two for each of the remaining automata in $\mathcal{M}$.

5.3 Algorithm GENERALIZEFA

Recall from Section 2.2 that the objective of the GENERALIZEFA algorithm is to generate for a set of automata $\mathcal{M}$, an automaton M with no more than α states such that $L(\mathcal{M}) \subseteq L(M)$ and $|L(M)|$ is minimum. This problem can be shown to be NP-hard by reducing the partition problem [12] to it.

Theorem 5.2: *Given a set of automata $\mathcal{M}$, the problem of computing a DFA M with no more than α states and for which $L(\mathcal{M}) \subseteq L(M)$ and $|L(M)|$ is minimum, is NP-hard.* ∎

```
Algorithm GENERALIZEFA ($\mathcal{M}, \alpha$)
Input:  $\mathcal{M}$ is a set of automata to be generalized.
        $\alpha$ is maximum number of states in generalized automaton.
Output: The generalized automaton for $\mathcal{M}$.
1) Compute DFA $M$ for $\cup_{M_i \in \mathcal{M}} M_i$
2) while ($|M| > \alpha$) do
       /* $M_{(s_i,s_j)}$ is the resulting automaton when states $s_i, s_j$ in
       $M$ are merged */
3)     Let $s_i, s_j$ be the pair of states in $M$ for which $|L(M_{(s_i,s_j)})|$
       is minimum among all pairs of states in $M$;
4)     Set $M$ to be equal to the DFA for $M_{(s_i,s_j)}$;
5) return $M$;
```

Figure 6: Algorithm for Generalizing a Set of Automata.

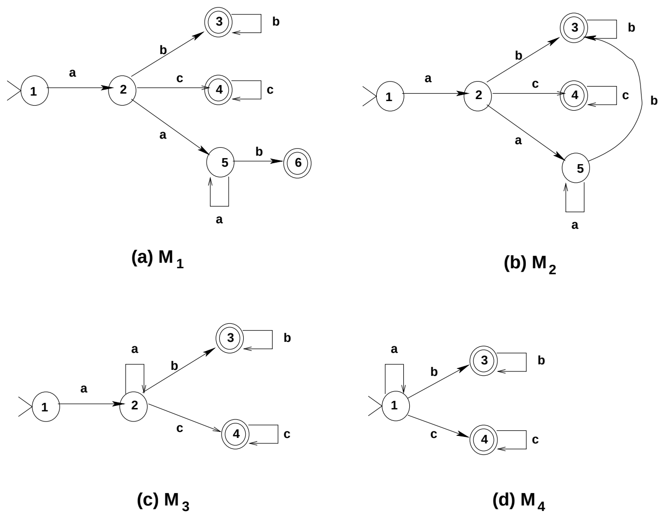


Figure 7: Automaton Generalization Example.

The algorithm for generalizing a set of automata $\mathcal{M}$ is shown in Figure 6. The algorithm greedily merges pairs of states (that result in the smallest regular languages) in the union automaton M until $|M| \leq \alpha$. Note that the automaton $M_{(s_i,s_j)}$ resulting from merging states s_i and s_j in M may not be a DFA. Thus, if we want to compute accurate counts and samples for $L_l(M_{(s_i,s_j)})$, we may need to convert $M_{(s_i,s_j)}$ into a DFA first. However, an option with lower overhead, if approximate counts and samples are acceptable, is to directly apply to $M_{(s_i,s_j)}$, the counting and sampling algorithms for NFAs described in Section 3. This would avoid converting NFA $M_{(s_i,s_j)}$ to a DFA, an operation which could be potentially expensive and could also cause the number of states in the automaton to increase.

A further optimization is to not consider all pairs of states s_i, s_j in M as candidates for merging in Step 3. One heuristic is to consider as candidates only those state pairs s_i, s_j that have similar incoming and outgoing transitions. The rationale here is that merging such similar states is more likely to result in automata with compact languages.

Example 5.3: Suppose we want to generalize the three automata for the REs abb^*, acc^* and aa^*b. The DFA for the union of the three automata is depicted in Figure 7(a). Suppose $\alpha = 3$, that is, we would like the final generalized automaton to contain at most 3 states. We trace the steps of Algorithm GENERALIZEFA on the union automaton M_1. The first pair of states merged by the algorithm are states 3 and 6 since the language for the resulting automaton M_2 in Figure 7(b) is $aa^*bb^*|acc^*$, which is smaller than the resulting languages when other pairs of states are merged. For instance, merging states 3 and 4 results in the language $(a(b|c)(b|c)^*)|aa^*b$, while $aa^*(bb^*|cc^*)$ is the language when states 2 and 5 are merged. In the next iteration, states 2 and 5 are merged to yield automaton M_3 in Figure 7(c) whose language is $aa^*(bb^*|cc^*)$. And in the final iteration, states 1 and 2 are merged, and the final automaton M_4 in Figure 7(d) containing 3 states is returned by the algorithm ($L(M_4)$ is $a^*(bb^*|cc^*)$). ∎

5.4 Optimizations

The RE-tree operations CHOOSEBESTFA and SPLITFA described in previous subsections required frequent computations of $|L(M_i \cap M_j)|$ and $|L(M_i \cup M_j)|$ to be performed for pairs of automata M_i, M_j. These computations can adversely affect RE-tree performance since construction of the intersection and union automaton M can be expensive. Further, the final automaton M may have many more states than the two initial automata M_i and M_j. This could be a problem since computing $|L(M)|$ using any of the size measures from Section 4 requires $|L_l(M)|$ for a range of l values to be calculated using Algorithm COUNTSTRINGS. (The MDL metric also requires random samples for each $L_l(M)$ to be generated).

In this subsection, we show how sampling can be used to speed up the performance of the RE-tree operation CHOOSEBESTFA. Due to lack of space, we defer details of our optimization techniques for SPLITFA to [6].

The CHOOSEBESTFA algorithm requires $|L(M \cap M_i)|$ to be estimated, where M is the automaton being inserted into the tree and M_i is an automaton in the current RE-tree node. In the following, we show how a random sample of $L_l(M)$ and knowledge of $|L_l(M)|$ for a range of l values can be used to compute $|L(M \cap M_i)|$ very fast for the two size measures proposed earlier. Thus, for an automaton M being inserted into the RE-tree index, our algorithm only requires us to generate counts and samples for $L_l(M)$ for specific values of l. We make use of the following two theorems (see [8]) that suggest how counts and samples for the intersection of two sets S_1 and S_2 can be generated using the count and sample information for one of them.

Theorem 5.4: *Suppose r_1 is a uniform random sample of set S_1. Then $\frac{|r_1 \cap S_2||S_1|}{|r_1|}$ is an unbiased estimator of the size of $S_1 \cap S_2$.* ∎

Theorem 5.5: *Suppose r_1 is a uniform random sample of set S_1. Then $r_1 \cap S_2$ is a uniform random sample of $S_1 \cap S_2$ with size $|r_1 \cap S_2|$.* ∎

Max-Count. For the Max-Count measure, we need to compute $|L_l(M \cap M_i)|$ for l values in the range $[1, \lambda]$. We assume that for these l values, we have already computed $|L_l(M)|$ and $\hat{L}_l(M)$, a uniform random sample of $L_l(M)$. From Theorem 5.4, it follows that $\frac{|\hat{L}_l(M) \cap L(M_i)||L_l(M)|}{|\hat{L}_l(M)|}$ is an unbiased estimate of $|L_l(M \cap M_i)|$. Here $\hat{L}_l(M) \cap L(M_i)$ can be computed efficiently by simply checking for each string in sample $\hat{L}_l(M)$, if it is accepted by M_i. Thus, with our sampling approach, we only need to compute counts and samples for $L_l(M)$ (the automaton being inserted) once at the beginning. There is no need to either construct the automaton for each $M \cap M_i$ or count $L_l(M \cap M_i)$ using the dynamic programming algorithm COUNTSTRINGS.

MDL. For computing $|L(M \cap M_i)|$ using the MDL measure, we need to generate random samples of $L_l(M \cap M_i)$ for $\lambda \leq l \leq \lambda + \theta - 1$. Suppose that $\hat{L}_l(M)$ denotes the random sample of $L_l(M)$. Then, due to Theorem 5.5, $\hat{L}_l(M) \cap L(M_i)$ is a uniform random sample of $L_l(M \cap M_i)$. Thus, using our sampling-based approach, sample $\hat{L}_l(M)$ needs to be computed only once in the beginning. In addition, we also need to construct the intersection automaton for each $M \cap M_i$ to encode the strings in the sample. However, we completely eliminate the need to count $L_l(M \cap M_i)$ using Algorithm COUNTSTRINGS or generate new samples for $L_l(M \cap M_i)$ using Algorithm GENRANDOMSTRING.

6 Experimental Evaluation

To determine the effectiveness of the RE-tree, we compare its performance against the sequential file approach which stores the REs in a flat file and searches the entire file sequentially for each search query. As we noted in Section 1, we are not aware of any disk-based indexing method for indexing REs (in their full generality). Our experimental results (based on synthetic data sets) indicate that the RE-tree approach offers a significant performance improvement (by factors ranging from 2 to 3) over the sequential search approach.

Data Sets: Each synthetic data set comprises of clusters of similar REs that are generated using a synthetic RE generator, where the number of clusters and the size of each cluster are controlled by the input parameters c_{num} and c_{size}, respectively. The REs in each cluster are similar in terms of both their content (i.e., symbol distribution) as well as their structure (i.e., parse tree). Content-wise, each cluster is associated with some subset of the alphabet $\Sigma' \subseteq \Sigma$, referred to as its *hot alphabet*, such that each RE symbol is drawn from Σ' with a probability of ρ and drawn from $(\Sigma - \Sigma')$ with a probability of $(1 - \rho)$. The parameter ρ applies to all the clusters and is referred to as the *hot probability*. Each cluster is mapped to its hot alphabet as follows: First, the alphabet Σ is arbitrarily partitioned into n_σ disjoint subsets $\Sigma = \cup_{i=1}^{n_\sigma} \Sigma_i$ such that each subset Σ_i

consists of $\frac{|\Sigma|}{n_\sigma}$ symbols[6], where n_σ is another input parameter. Each cluster is then randomly mapped to one of the n_σ subsets as its hot alphabet.

In terms of structural similarity, each cluster of similar REs is generated by first randomly generating a *seed RE* R_i and then using R_i to derive the other REs in the cluster by making a random modification to the parse tree of R_i. Note that in a parse tree for an RE, the leaf nodes correspond to symbols in Σ while the internal nodes correspond to one of the three operators: union, concatenation, or kleene star. The structure of each seed RE is controlled by two parameters θ_E and θ_e such that each seed RE is of the form $R_i = E_1.E_2.\cdots.E_n$, where n is a random integer between 1 and θ_E and each E_j has one of the following four forms: (1) $(e_1|e_2|\cdots|e_{n_j})$, (2) $(e_1.e_2.\cdots.e_{n_j})$, (3) $(e_1|e_2|\cdots|e_{n_j})^*$, or (4) $(e_1.e_2.\cdots.e_{n_j})^*$; where n_j is a random integer between 1 and θ_e. A similar RE is derived from a seed RE R_i by adding a new internal node to its parse tree (corresponding to either the kleene star operator or union operator) at a randomly selected location in the parse tree. For instance, when adding a new union operator, one of the operands of the union operator is one of the existing nodes in the parse tree while the other operand is a new leaf node corresponding to a randomly generated symbol.

Queries: For each data set, we generate a set of 1000 random queries, where each query $w \in \Sigma^*$ is generated as follows. First, randomly select an RE R from the data set, and randomly generate an integer n between 5 and 10. Next, generate a random length-n string w that matches R; if no such string exists, we repeat the generation process with another randomly chosen pair of values for R and n.

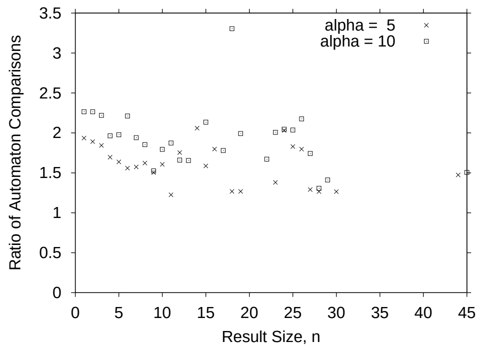
Algorithms: For the sequential file approach, a file of automata is created for each data set by packing the collection of automata into as few pages as possible. To find matching REs for a query string w requires sequentially checking against each automaton in the file. For the RE-tree approach, our implementation is based on the algorithms presented in Sections 2 through 5. We use the MDL measure for comparing sizes of regular languages since in our experiments, we found it to be more effective than the Max-Count metric. The RE-tree operations for the most part are implemented as described in Section 5; however, our implementation does not incorporate any of the sampling-based optimizations discussed in Section 5.4. We intend to conduct further experiments with these optimization techniques on larger data sets as part of our future work.

The experiments were conducted on a 700 MHz Intel Pentium III machine with 512 MB memory running FreeBSD 4.1. Both the sequential file and RE-tree methods were implemented as Unix files on a 10GB SAMSUNG-SV1022D disk. For each query and each method, we measured both the evaluation time (including both CPU and I/O times) as well as the number of automaton comparisons (i.e., the number of automata that were checked against the query). Each query was run 10 times (for each method) to compute its average measurement values.

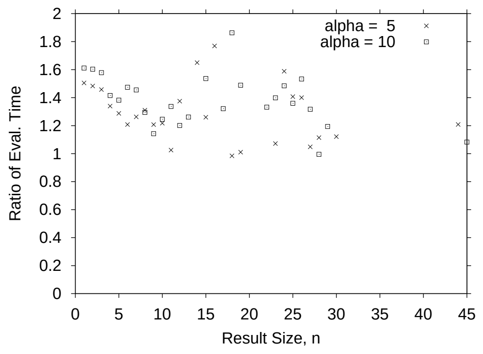
[6]For simplicity, assume that $|\Sigma|$ is a multiple of n_σ.

Param	Meaning	Value
P	Page Size (in KB)	4
$\|\Sigma\|$	Size of alphabet	20
θ_E	Max. number of first-level expressions (per RE)	3
θ_e	Max number of second-level symbols (per first-level expression)	3
n_σ	Number of alphabet subsets	5
α	Maximum number of states for bounding automata	5, 10
c_{num}	Number of RE clusters	50, 100
c_{size}	Size of RE cluster	25, 50
D	Size of data set given by $c_{num} \times c_{size}$	1250, 5000
ρ	Probability that a symbol belongs to the "hot" alphabet subset	0.5, 0.75, 1.0

Table 2: Experimental Parameters and Values.



(a) Ratio of Automaton Comparisons, $\frac{N_{seq}(n)}{N_{rt}(n)}$.



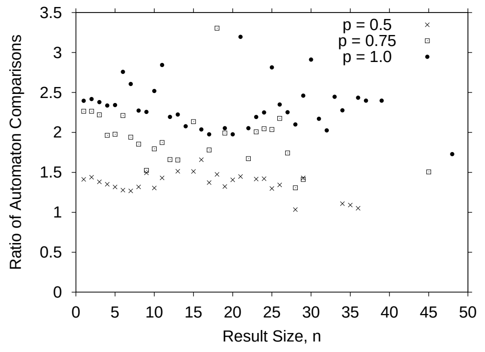
(b) Ratio of Evaluation Time, $\frac{T_{seq}(n)}{T_{rt}(n)}$.

Figure 8: Varying α, $\rho = 0.75$, $c_{num} = 50$, $c_{size} = 25$.

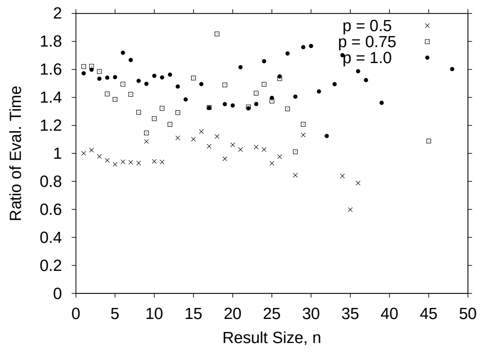
6.1 Experimental Results

This section presents experimental results that compare the performance of the two approaches by varying three main parameters: (1) α, the maximum size of a bounding automaton; (2) ρ, the hot probability; and (3) the size of the data set (number of REs) given by $D = c_{num} \times c_{size}$. Table 2 summarizes the values of the parameters used in our experiments.

The performance results are presented in terms of two

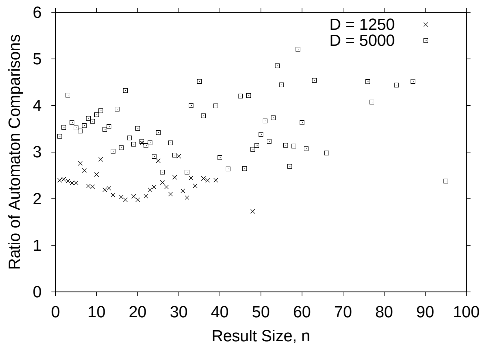


(a) Ratio of Automaton Comparisons, $\frac{N_{seq}(n)}{N_{rt}(n)}$.

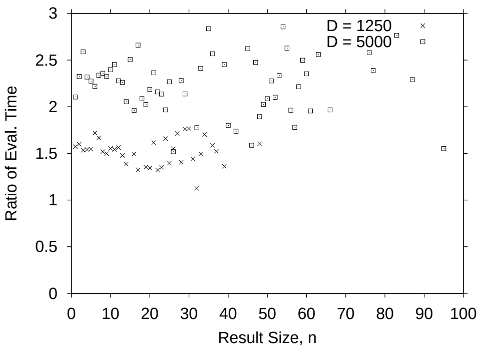


(b) Ratio of Evaluation Time, $\frac{T_{seq}(n)}{T_{rt}(n)}$.

Figure 9: Varying ρ, $\alpha = 10$, $c_{num} = 50$, $c_{size} = 25$.



(a) Ratio of Automaton Comparisons, $\frac{N_{seq}(n)}{N_{rt}(n)}$.



(b) Ratio of Evaluation Time, $\frac{T_{seq}(n)}{T_{rt}(n)}$.

Figure 10: Varying D, $\alpha = 10$, $\rho = 1.0$.

ratios, $\frac{N_{seq}(n)}{N_{rt}(n)}$ and $\frac{T_{seq}(n)}{T_{rt}(n)}$; where $N_{seq}(n)$ ($N_{rt}(n)$) is the average number of automaton comparisons incurred by the sequential (RE-tree) approach for queries with a result size (that is, number of matching REs) of n, and $T_{seq}(n)$ ($T_{rt}(n)$) is the average evaluation time incurred by the sequential (RE-tree) approach for queries with a result size of n. Note that the number of automaton comparisons incurred by the sequential fi le approach $N_{seq}(n)$ is always equal to D, the size of the data set.

Figure 8 shows the performance results as α is varied. The graphs indicate that RE-trees outperform the sequential fi le approach by up to a factor of 3 and 2, respectively, for the number of automaton comparisons and search time. Our results indicate that both the number of automaton comparisons (Figure 8(a)) and running time (Figure 8(b)) improve with larger values of α. This is because as α increases, the precision of the bounding automata generally becomes higher, thereby resulting in better pruning of “false-drops” (i.e., path traversals that do not lead to any qualifying REs) and hence fewer number of automaton comparisons and index page accesses.

Figure 9 depicts the performance results as ρ is varied. Since the similarity of the REs in each cluster becomes higher with a larger value of ρ, the RE-tree approach is able to improve its fi ltering (with tigher bounding automata) thereby resulting in less false drops and hence improved query evaluation. Therefore, the performance of RE-trees improves as ρ increases.

Note that for the experimental results in Figures 8 and 9, the average evaluation time incurred by the sequential fi le approach is about 6500 ms, and the space requirement of the RE-tree approach is about 1.5 times more than that of the sequential fi le approach.

Figure 10 illustrates the performance results as D is varied. We note that the performance gain of RE-trees increases with a larger value of D.

7 Related Work

Although several indexing methods have been proposed to speed up the search of textual data with REs (e.g., bit-parallel implementation of NFA [22] and suffi x trees [2]), there is very little work on the reverse indexing problem involving the retrieval of REs that match an input string. Besides the recent main-memory indexes proposed for fi ltering XML documents using XPath expressions [1, 5, 11], which is a specialized class of REs, we are not aware of any work on (disk-based) indexes for general REs.

Recently, several indexing methods [1, 5, 11] have been proposed for fi ltering incoming XML documents against a collection of user profi les expressed as XPath queries, where each XPath query specifies patterns pertaining to paths in the document graph. Our work differs from these indexing schemes in two important aspects. First, the class of REs supported by XPath queries is more specialized as both the union and kleene operators are to be used only with the entire alphabet (i.e., as Σ and Σ^*, respectively). Consequently, the associated finite automata are simpler as there are no cycles (besides self-loops) and each state has at most two outgoing transitions: a self-loop transition to itself and a transition to one other state (labelled with either a single letter of the alphabet or the entire alphabet). This is in contrast to our work that deals with REs in their full generality. Second, as opposed to our hierarchical disk-based index approach, the existing XPath-based indexing schemes are designed for main-memory resident data.

A related problem is that of indexing sets of objects, which has been addressed in various contexts, including document retrieval (where each document is characterized by a set of keywords) [3] and evaluation of set predicates (involving set-valued attributes) [16]. Inverted lists and signature files are two well-known techniques that have been developed for this problem [3]. However, since these methods are targeted for indexing finite sets of objects, they are not appropriate for indexing infinite regular languages.

8 Conclusions

In this paper, we presented the RE-tree, which is a novel index structure for performing fast retrievals of REs that match a given input string. In order to overcome the challenge of indexing the infinite regular sets (corresponding to REs) with no well-defined notion of spatial locality, we developed novel measures for comparing the relative sizes of infinite regular languages; these range from "rate-of-growth" estimates to encoding costs of sample strings using the corresponding REs. We also proposed innovative solutions for the various RE-tree operations, including the effective splitting of RE-tree nodes and computing a "tight" bounding RE (under a space constraint) for a collection of REs. Finally, we showed how sampling-based approximation algorithms can be used to significantly speed up the performance of RE-tree operations.

Our experimental results with synthetic data sets clearly demonstrate that the RE-tree index is significantly more effective than performing a sequential search for matching REs, and in a number of cases, outperforms sequential search by factors as high as 3. As part of our future work, we are conducting more experiments with the sampling-based approximation algorithms on larger data sets to further explore their tradeoffs and fine-tune the design of RE-trees. Another direction that we are actively investigating is the application of the counting, sampling and size estimation techniques for regular languages that we developed in the paper to other problem domains like selectivity estimation of REs, clustering REs, etc.

References

[1] M. Altinel and M.J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of VLDB*, pages 53–64, Cairo, Egypt, 2000.

[2] R. Baeza-Yates and G. Gonnet. Fast Text Searching for Regular Expressions or Automaton Searching on a Trie. *Journal of the ACM*, 43(6):915–936, November 1996.

[3] R. Baeza-Yates, B. Ribeiro-Neto, and G. Navarro. Indexing and Searching. In *Modern Information Retrieval*, chapter 8, pages 191–228. ACM Press, 1999.

[4] N. Beckmann, H-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of SIGMOD*, pages 322–331, Atlantic City, New Jersey, 1990.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 235–244, San Jose, California, February 2002.

[6] C.-Y. Chan, M. Garofalakis, and R. Rastogi. "RE-Tree: An Efficient Index Structure for Regular Expressions". Bell Laboratories Tech. Memorandum, February 2002.

[7] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, http://www.w3.org./TR/xpath, November 1999.

[8] W. G. Cochran. *Sampling Techniques*. John Wiley & Sons, 1977.

[9] The Intel Corporation. Intel NetStructure XML Accelerators. http://www.intel.com/netstructure/products/xml_accelerators.htm, 2000.

[10] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1999.

[11] Y. Diao, P. Fischer, M.J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 341–342, San Jose, California, February 2002.

[12] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[13] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *Proc. of SIGMOD*, pages 165–176, Dallas, Texas, May 2000.

[14] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of SIGMOD*, pages 47–57, Boston, Massachusetts, 1984.

[15] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[16] Ishikawa, Y. and Kitagawa, H. and Ohbo, N. Evaluation of Signature Files as Set Access Facilities in OODBs. In *Proceedings of SIGMOD*, pages 247–256, Washington, D.C., May 1993.

[17] S. Kannan, Z. Sweedyk, and S. Mahaney. Counting and Random Generation of Strings in Regular Languages. In *Proceedings of SODA*, pages 551–557, 1995.

[18] P. Kilpelainen, H. Mannila, and E. Ukkonen. MDL Learning of Unions of Simple Pattern Languages from Positive Examples. In *Proceedings of EuroCOLT*, 1995.

[19] J.R. Quinlan and R.L. Rivest. Inferring Decision Trees Using the Minimum Description Length Principle. *Information and Computation*, 80:227–248, 1989.

[20] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[21] J. W. Stewart. *BGP4, Inter-Domain Routing in the Internet*. Addison Wesley, 1998.

[22] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10):83–91, October 1992.