q-Gram Matching Using Tree Models

Prahlad Fogla and Wenke Lee, *Member*, *IEEE Computer Society*

Abstract—q-gram matching is used for approximate substring matching problems in a wide range of application areas, including intrusion detection. In this paper, we present a tree-based model to perform fast linear time q-gram matching. All q-grams present in the text are stored in a tree structure similar to Trie. We use a tree redundancy pruning algorithm to reduce the size of the tree without losing any information. We also use suffix links for fast q-gram search during query matching. We compare our work with the Rabin-Karp-based hash-table technique, commonly used for multiple q-gram search. We present results of experiments on system call sequence data used for intrusion detection.

Index Terms—Intrusion detection, q-gram matching, pattern matching, search problems, string matching, suffix tree, trees, tree data structure, word processing.

1 Introduction

Given a text string, T, and a query string, Q, the problem of q-gram matching is to find all the substrings of length q in the query which are also present in the text. This problem can be easily extended to multiple text strings. The length of the text string is assumed to be very large and the length of the query is much smaller than the text. q-gram matching has been used extensively in many application areas including information retrieval, signal processing, pattern recognition, and computational biology. In computational biology, genomic sequences show a high level of matching for small length substrings. In information retrieval, filtration-based approximate string matching algorithms require efficient search of all the q-gram shared by query and text. In some word processors, q-gram matching is performed for approximate matching and finding misspellings.

Substring matching has also been used extensively in intrusion detection. For example, consider the problem of detecting anomalous program behavior (in anomaly detection problem). Normal behavior of a program can be observed via its interaction with the underlying operating system, which can be characterized by the sequence of system calls (along with other auxiliary information such as call stack information at the instance when the system call is made) generated by the program. For example, it has been observed that small length substrings are very consistent throughout different normal executions of a program [9]. We can build a normal profile of a program using the substrings generated by the sample executions of the program. While monitoring, we can observe the substrings generated by the program and check if they match with the stored substrings. If a substring does not have a match, an alarm is raised. Similarly, many intrusion detection systems also rely on substring matching of network traffic or host activities with normal patterns or attack patterns. One main requirement of an intrusion detector is that it should be fast, so that one can detect any possible attack as early as possible. One of the main factors for the speed of an intrusion detector is the speed of the substring matching algorithm it uses. Since the set of normal or attack patterns needs to be extensive, there are usually a huge number of text patterns in intrusion detection.

Existing string or substring matching algorithms are not suitable for the q-gram matching problem and do not have good runtime efficiency. The expected runtime complexity of the best existing string matching algorithms is sublinear to the length of the text. They do not apply well to multiple q-gram matching with huge text size. Some string matching algorithms with multiple texts have complexity of length of the query. When used for multiple q-gram matching, they take $q \times m$ time, where m is the length of query.

A simple solution for the problem will be to record all the unique q-grams present in the text and store them in a hash-table. While matching, we can get each of the q-grams in the query and check if it is present in the hash-table. If the calculation of the hash function takes time t, then the total time to match the query will be $O(mt)$, where m is the length of the query. The Rabin-Karp algorithm follows the similar idea and uses an efficient method to calculate hashes. The Rabin-Karp algorithm is the only existing algorithm that we are aware of which works well for the given problem. In this paper, we present a new q-gram matching algorithm that is more efficient than the Rabin-Karp algorithm. Although our work was motivated by applications in intrusion detection, the algorithm is general and is applicable to other domains.

1.1 Solution Outline and Contribution

The solution we propose makes use of interpretability and efficiency of tree structure to develop a linear time algorithm for the q-gram matching problem. In the preprocessing step, all the q-grams present in the text are extracted and stored in a tree structure. This tree structure is similar to the Trie structure used for storing dictionaries. We can add suffix links [14] for efficient runtime substring matching. This tree may become very big depending on the character size, the structure of text, and the value of q. We also propose a tree redundancy pruning algorithm to reduce the size of the tree. The proposed solution has the same linear time complexity as a hash-table but with a smaller constant. It also uses less storage space than a hash-table. In addition, our approach can be used to perform q-gram matching for all the different values of q ($1 \leq q \leq$ Max Tree Depth). A hash-table method, on the other hand, can be used for only one value of q.

• *The authors are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {prahlad, wenke}@cc.gatech.edu.*

Manuscript received 3 Feb. 2005; revised 8 July 2005; accepted 4 Oct. 2005; published online 17 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0045-0205.

1.2 Paper Organization

In Section 2, we discuss some of the earlier work related to substring matching. In Section 3, we show the use of a tree structure for substring matching. Next, we present the idea of tree redundancy pruning and how to use suffix links to make the solution space and time efficient. Section 5 shows some results for intrusion detection application. We present our conclusion and future work in Section 6.

2 Related Work

The problem of substring matching has been studied extensively by researchers in several fields. Initial research was mainly focused on complete string matching, required for keyword search in a database. But, later the focus shifted to substring matching and approximate string matching. In approximate string matching, a query matches a text with at most k-mismatches.

Suffix trees are used extensively for different string processing problems. A suffix tree can be used for linear time search of a single string in a text. Various algorithms were suggested by Weiner [23], McCreight [14], and Ukkonen [22] for efficient suffix tree generation. Chen and Seiferas [6] proposed a simple technique to efficiently generate a tree to store all the subwords of a word.

Knuth et al. (KMP) [12] proposed a string searching algorithm. The algorithm preprocesses the query to compute a *shift* function, which is used in the search phase later. The search phase has runtime complexity of $O(m+n)$, where n is the length of text and m is the length of query. Boyer and Moore (BM) [3] suggested a faster algorithm which is now widely implemented for string searching in shell commands (grep) and some editors. The query preprocessing phase of the BM algorithm is of $O(m)$ complexity and the matching phase has the worst case of $O(mn)$ time complexity. But, for English text, the average runtime complexity of the BM algorithm is much smaller than the worst case and is around three times better than the KMP algorithm. Sunday [18] improved on the BM algorithm by improving the BM's shift function and adding a similar shift function used in the KMP algorithm. Sunday's algorithm has the worst case of $O(m+n)$ and its expected runtime is smaller than the BM algorithm.

The string matching algorithm with multiple texts was proposed by Aho and Corasick (AC) [2]. The AC algorithm preprocesses the texts to create a deterministic finite automaton (DFA), which is similar to Trie structure. The algorithm reads the successive characters in the query and makes state transitions based on the next character in the query. The algorithm takes linear time to the length of query. Coit et al. [7] proposed a fast string matching algorithm, the *AC_BM* algorithm, for intrusion detection. The algorithm stores the texts in a tree similar to Aho and Corasick. Its matching process uses techniques similar to Boyer and Moore.

The above algorithms are based on Boyer and Moore and work well for large queries with a small character size (σ). They do not perform very well for large character sizes or huge texts.

Approximate string matching [15] has also been studied extensively. Landau and Vishkin [13] proposed an $O(kn + km \log m)$ approximate string matching algorithm where k is the number of allowed mismatches. The Lanadu-Vishkin algorithm was improved by the Galil-Ginacarlo algorithm [10], which has $O(kn + m \log m)$ time complexity. Tarhio and Ukkonen [20] proposed a Boyer and Moore algorithm-based approach for approximate string matching. Chole and Hariharan [8] presented an $O(n(1 + k^3/m))$ algorithm based on dynamic programming. Wu et al. [25] used DFA for approximate matching in $O(mn/\log s)$, where s is the number of states in DFA. No approximate string matching algorithm is known to have a worst-case complexity less than $O(kn)$.

A lot of work has been done on filtration-based approximate string matching. Filtration-based algorithms first choose a set of positions in the text which are potentially similar to the pattern. Then, each preselected position is verified for a match using an accurate string matching technique. Karp and Rabin [11] first described a filtration-based approach for exact string matching. Navarro and Baeza-Yates ([16], [17]) proposed an $O(kn \log_\sigma m/\sigma)$ approximate string matching algorithm by improving on the idea developed by Wu and Manber [24]. Later, q-gram-based filtration algorithms ([21], [5], [19]) were developed for approximate matching. The q-gram filtration approach is based on observation that, if a pattern approximately matches a substring of the text, then they share a certain number of q-grams for sufficiently large q. Finding all q-grams shared by the pattern and the text is done by hashing. The algorithm proposed in our paper can be used to find all q-grams efficiently. Chang and Marr [5] proposed filtering-based algorithm with an average complexity of $O(\frac{n(k+\log_\sigma m)}{m})$. Furthermore, they proved that this is a lower bound on the average complexity. In practice, filtration-based string matching algorithms are the fastest, but their applicability is limited by the error level.

The Rabin-Karp algorithm [11] searches for a substring within a text by hashing. It is suitable for searching multiple patterns of the same length in the text. Suppose we are matching patterns of length q in a text T. First, we compute the hashes of the text patterns and store them in a hash-table. Next, we compute and match the hashes of all substrings of the query with the hashes of the text patterns. If two hashes match, then there is a very high probability that the two substrings match. To be completely sure, one can perform a naive matching on the two substrings. In case the hashes do not match, then we can say for sure that the substrings do not match. The main idea of Rabin-Karp is to efficiently compute hashes of successive substrings in constant time by using the hash of the previous substring. Using such a hashing function, matching can be done in expected time of $O(m)$.

QUASAR, proposed by Burkhardt et al. [4], finds all the substrings in a text which has a maximum of k mismatching q-grams as in query. *QUASAR* uses a suffix array to retrieve the positions of any given q-gram in the text. The searching of a given q-gram is done using a hash-table. The preprocessing phase of *QUASAR* takes $O(n \log n)$ time. The matching of a query string takes $O(nm)$. The space complexity of the matching algorithm is $O(5n + qS)$, where S is the number of unique q-grams in text. The high complexity of *QUASAR* is because it tries to find all positions in the text that has a possible matching alignment to the query string.

All of the above string matching algorithms are not suitable for multiple q-gram matching. The best known string matching algorithm takes $O(\frac{n \log_\sigma m}{m})$ expected time. When applied to multiple q-gram matching, the algorithm have the average complexity of $O(\frac{n \log_\sigma q}{q} m)$. This is not acceptable for situations where text size is huge. When

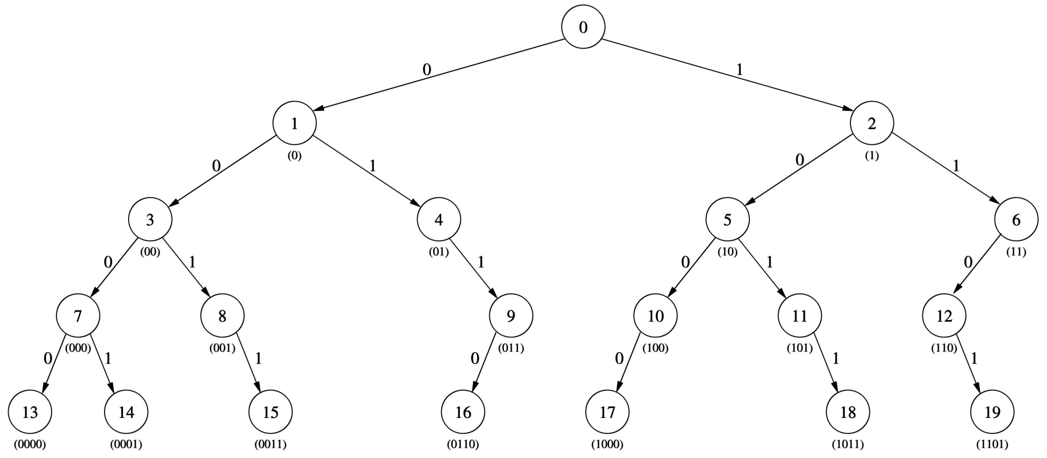


Fig. 1. Tree built from text 1000011011 for substrings of length 4. There are seven length-4 substrings (4-grams) present in the text, namely, $\{1000, 0000, 0001, 0011, 0110, 1101, 1011\}$. Each leaf node corresponds to one substring in the set.

applied to a q-gram matching problem, the Aho and Corasick algorithm takes $O(qm)$ time. The Rabin-Karp algorithm works in linear time of the query size but needs a lot of space to store the hash-table. Furthermore, it needs to create separate hash-tables for different values of q.

3 Tree Model

The tree structure provides a general framework for a systematic study of stream data. Trees are computationally efficient for storage, addition, and searching for sets of strings. Tree structures have been used extensively in many applications. For example, decision tree classifiers are successfully used in areas such as character/speech recognition, medical diagnosis, and remote sensing. Prefix trees are used to store and search through dictionaries. Suffix trees are a widely used data structure for text processing.

3.1 Notations

In the discussion that follows, we use x, y to represent arbitrary characters and X, Y to represent arbitrary sub strings. $X[i, \cdots, j]$ represents a substring of X that starts with the ith character and ends with the jth character. T represents the provided text and Q stands for the query string. A substring is said to be *accepted* by a substring matching algorithm if it was matched successfully. The substring is *rejected* if no match was found. For any string X, its length is denoted by $|X|$. $X[i]$ denotes the ith character of string X. xX and Xx mean string X is appended with character x at the start and at the end, respectively. $\mathcal{T}$ represents the tree used to store the q-grams of the text T. Fig. 1 shows an example tree to store length-4 substrings (or 4-grams) of text 1000011011. $\mathcal{T}_s$ is the tree with suffix link, $\mathcal{T}_p$ is the tree after pruning, and $\mathcal{T}_{sp}$ is the pruned tree with suffix link. The concepts of a pruned tree and a suffix link will be explained in later sections. q denotes the length of the matching substring. $N(l)$ is the number of nodes at depth l of the tree, and N_T is the total number of nodes in the tree. Each node of the tree corresponds to a substring present in the text. For convenience, we use the term *node* $X[i, \cdots, j]$ to denote the tree node corresponding to sub string $X[i, \cdots, j]$. For example, node 13 in the Fig. 1 is referred to as node $T[2, \cdots, 5]$ because it refers to substring $T[2, \cdots, 5]$ (0000). Node $X[i+1, \cdots, j]$ is called *immediate suffix node* of the node $X[i, \cdots, j]$. Node $X[i+l, \cdots, j]$ is called *longest suffix node* of the node $X[i, \cdots, j]$ if none of the nodes $X[i+l', \cdots, j]$, $1 \leq l' < l$, exists in the tree. Two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be *identical* if, for all nodes in $\mathcal{T}_1$, there is a corresponding node in $\mathcal{T}_2$ with the same path from the root, and vice versa. Tree $\mathcal{T}_1$ is *similar* to tree $\mathcal{T}_2$ if both trees are identical until depth d, where $d = min(depth(\mathcal{T}_1), depth(\mathcal{T}_2))$.

3.2 Tree Structure

A set of strings of length q can be efficiently represented using a depth q tree ($\mathcal{T}$). A node in the tree at depth l is associated with a substring of length l. Also, a link between a node at depth $l-1$ and a node at dept l is associated with the character at position l in the string. Fig. 1 shows an example sequence and the corresponding tree for sequence length 4. Constructing such a tree requires $q(|T| - q + 1)$ time.

Consider the substring matching problem with query $Q[1, \cdots, m]$. For each substring $Q[i, \cdots, i+q-1]$, $1 \leq i \leq |Q| - q + 1$ in the query, start from the root node and traverse the tree. At depth l, choose the link corresponding to character $Q[i+l+1]$. If at any depth-l node, there is no link corresponding to character $Q[i+l+1]$, then the substring is not present in the text. If we reach the end of the substring, then we have found a match and the substring is present in the text. A formal description of the q-gram matching algorithm using $\mathcal{T}$ is presented in Algorithm 1.

Algorithm 1 q-gram matching using $\mathcal{T}$

```
for all $q$-grams, $Q[i, \cdots, i+q-1]$, $i = 1$ to $|Q| - q + 1$ do
  Set current node as root of the tree $\mathcal{T}$
  for $j = 0$ to $q - 1$ do
    if the current node does not have a child for
      character $Q[i+j]$ then
      reject the current substring
      $Q[i, \cdots, i+q-1]$ and match the next substring
      starting from the root node
    else
      traverse to the child node for character $Q[i+j]$
    end if
  end for
  if $j = q - 1$ and we reach a leaf node then
    accept substring $Q[i, \cdots, i+q-1]$ and match the next
```

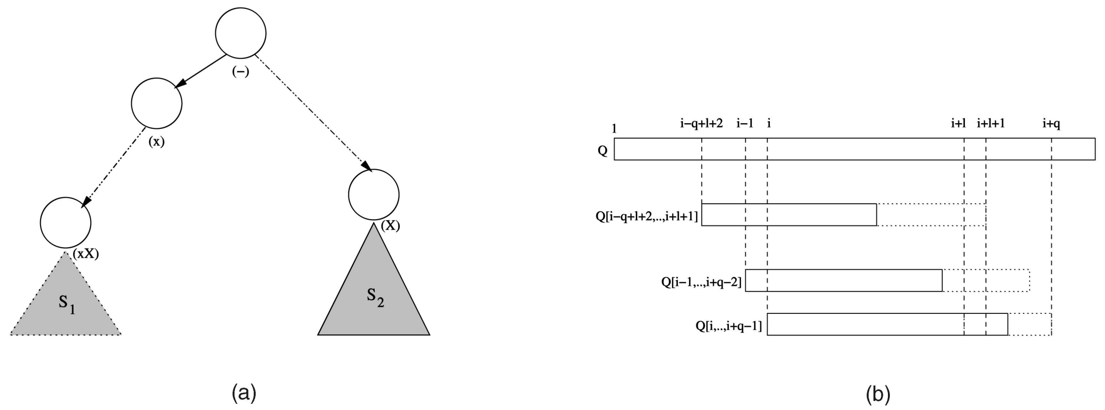


Fig. 2. (a) Tree redundancy pruning algorithm: subtrees S_1 and S_2 are similar. Remove S_1. (b) Matching using pruned tree: Marking previous substrings after finding a mismatch. In both figures, a path up to the solid and dash-dot lines are present in the pruned tree. The dotted part is pruned or absent in the tree.

```
      substring starting from the root node
    **end if**
  **end for**
```

Consider the problem of matching the query $Q = (011000)$ for the tree shown in Fig. 1 with $q = 4$. Substrings of length 4 are $(0110, 1100, 1000)$. For substring 0110, we will traverse nodes $(0, 1, 4, 9, 16)$, in the given order, and reach the end of the substring. But, when we try substring 1100, we will traverse nodes $(0, 2, 6, 12)$ and then stop at node-12 because there is no link for character 0. At this point, we can conclude that substring 1100 is not present in the text. Substring 1000 is matched successfully by traversing nodes $(0, 2, 5, 10, 17)$. The above matching process takes at most $O(q)$ steps to match each substring present in the query. Thus, $|Q|$ length query will take $O(q|Q|)$ time.

3.3 Tree Redundancy Pruning

The number of nodes in the tree increases with the number of unique substrings in the text and with the tree depth. The space required to store the tree may become unmanageable for a very large number of unique substrings. We have developed a tree redundancy pruning algorithm which removes redundant nodes present in a tree.

Let us revisit the tree constructed in Fig. 1. By looking at node-6, one can infer that substring 11 can be followed by only 01. Now, suppose we are matching query $(011xy)$. While matching $011x$, we can stop after reaching node-9. Then, we can check if x is equal to 0 or not while matching the next substring $11xy$. Thus, we can safely remove the child node (node-16) of node-9. This removal of node-16 should not affect the substring matching capability of the tree. Similarly, the child of node-11 can be removed because the existence of 1 after 101 can be checked while matching the next substring 01?? ("?" matches all characters.) Following the same idea, we can remove all the children of node-3.

Suppose we construct a tree $(\mathcal{T})$ with depth q as shown in Fig. 2a. If for any node xX at depth l, the $(q-l)$-depth subtree rooted at xX is similar[1] to the subtree rooted at node X, we remove the subtree of node xX and make node xX a leaf node. We repeat this for all the nodes. The final tree $(\mathcal{T}_p)$ will be the pruned version of the original tree $(\mathcal{T})$. A pruned version of the tree in Fig. 1 is shown in Fig. 3.

1. Recall the definition of tree similarity from Section 3.1

To prune a tree, for each node, we need to compare the subtree rooted at that node and the subtree rooted at its immediate suffix node. A comparison of two trees takes time linear to the number of nodes in the tree. Thus, the total time required to prune the tree is $O(N_T^2)$, where N_T is total number of nodes in the tree.

Consider the substring matching problem with query $Q[1, \cdots, m]$. While matching a substring $Q[i, \cdots, i+q-1]$, if we reach the end of the substring $Q[i, \cdots, i+q-1]$, it is successfully matched and is accepted. If we find a mismatch at character $Q[i+l+1]$, $0 \le l \le q-2$, (see Fig. 2b), the substrings

$$Q[i-q+l+2, \cdots, i+l+1],$$
$$Q[i-q+l+3, \cdots, i+l+2], \cdots, Q[i, \cdots, i+q-1]$$

are marked *rejected*. If any node in the path of $Q[i, \cdots, i+q-1]$ is pruned, we may reach the leaf node before the end of the substring. In such a case, we just mark it as *pending* and continue to match the next substrings. In the end, the substrings which are not *rejected* and marked *pending* are accepted. The detailed description of the matching algorithm using $\mathcal{T}_p$ is given in Algorithm 2.

```
**Algorithm 2** $q$-gram matching using $\mathcal{T}_p$
  **for all** $q$-grams, $Q[i, \cdots, i+q-1]$, $i = 1$ to $|Q|-q+1$ **do**
    Set current node as root of the tree $\mathcal{T}$
    **for** $j = 0$ to $q-1$ **do**
      **if** the current node is a leaf node **then**
        mark the current substring $Q[i, \cdots, i+q-1]$ as
        *pending* and match the next substring starting
        from the root node
      **else if** the current node does not have a child for
            character $Q[i+j]$ **then**
        **for** $k = 0$ to $q-j-1$ **do**
          **if** $Q[i-k, \cdots, i-k+q-1]$ is marked *pending*
            **then**
            mark $Q[i-k, \cdots, i-k+q-1]$ as *rejected*
          **else**
            continue matching the next substring starting
```

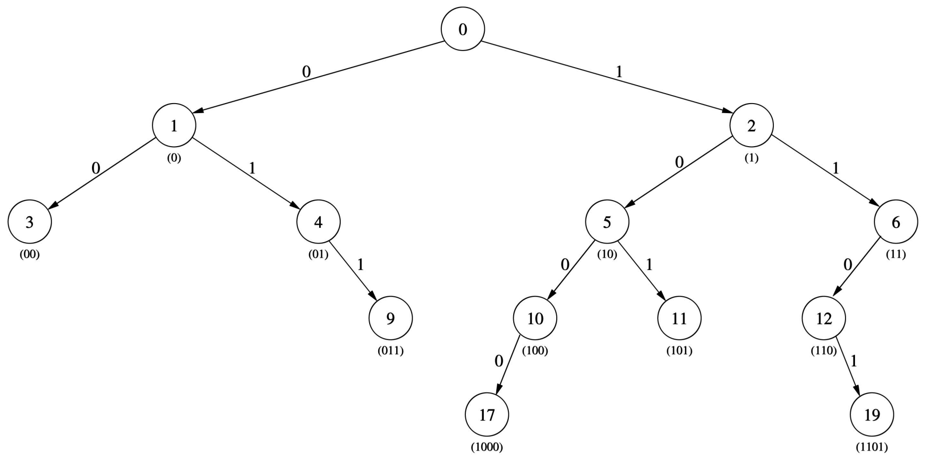


Fig. 3. The pruned version of the tree shown in Fig. 1. The subtrees of nodes 3, 9, and 11 were removed because they were similar to the subtrees of nodes 1, 6, and 4, respectively.

```
                from the root node
              **end if**
            **end for**
            break
        **else**
          traverse to the child node for character $Q[i + j]$
        **end if**
      **end for**
      **if** $j = q - 1$ and we reach a leaf node **then**
        mark the current substring $Q[i, \cdots, i + q - 1]$ as
        *accepted* and match the next substring starting from
        the root node
      **end if**
    **end for**
    mark all the *pending* substrings as *accepted*
```

The complexity of the matching algorithm depends on the effect of pruning on the text and also on the query string. The worst case can be $O(q|Q|)$, which is the same as the original tree matching algorithm. We will discuss more results on the effect of pruning in Section 5.

Theorem 3.1. *Substring matching using $\mathcal{T}_p$ is equivalent to substring matching using $\mathcal{T}$.*

Proof. We shall first prove that if a substring $Q[i, \cdots, i + q - 1]$ is accepted by the tree $\mathcal{T}$, then it should also be accepted by the pruned tree $\mathcal{T}_p$. If node $Q[i, \cdots, i + q - 1]$ is not pruned, then we will reach the end of the substring while matching it, and it will be accepted by the pruned tree. But, if the path to node $Q[i, \cdots, i + q - 1]$ is pruned at $Q[i, \cdots, i + j]$ (see Fig. 4a), then it will be marked *pending*. Consider matching the next $q - 1$ substrings ($Q[i + \alpha + 1, \cdots, i + \alpha + q]$, $\forall 0 \leq \alpha \leq q - 2$). We should not find any mismatch before character $Q[i + q]$ because $Q[i + \alpha + 1, \cdots, i + q - 1]$ are substrings of $Q[i, \cdots, i + q - 1]$. Thus, $Q[i, \cdots, i + q - 1]$ will not be marked as *rejected* because there is no future mismatch in the overlapping substrings and will be accepted by the pruned tree $\mathcal{T}_p$.

Now, we will prove the second part, that, if a substring is rejected by tree $\mathcal{T}$, then it will also be rejected by pruned tree $\mathcal{T}_p$. Suppose substring $Q[i, \cdots, i + q - 1]$ is not matched successfully in $\mathcal{T}$ at character $Q[i + l + 1]$, $1 \leq l \leq q - 2$. If the path is not pruned, then the substring will also be rejected by the pruned tree. Suppose the path is pruned at $Q[i, \cdots, i + j]$, for some $1 \leq j \leq l$. Suppose character $Q[i + l]$ is not pruned in the subtree of node $Q[i + l', \cdots, i + j]$, $1 \leq l' \leq j - 1$ (see Fig. 4b). All the subtrees $S_{i+\alpha}$, $\forall 0 \leq \alpha < l'$, are similar to the subtree $S_{i+l'}$. Since path $Q[i + j + 1, \cdots, i + l + 1]$ is not present in subtree S_i, the path will not be present in subtree $S_{i+l'}$. Thus, while checking for substring $Q[i + l', \cdots, i + l' + q - 1]$, we will find a mismatch at node $Q[i + l', \cdots, i + l]$. This is because character $Q[i + l + 1]$ is not supposed to follow character $Q[i + l]$ in subtree $S_{i+l'}$. At this point, substring $Q[i, \cdots, i + q - 1]$ will be marked *rejected* along with all the substrings $Q[i + k, \cdots, i + k + q - 1]$, $0 \leq k \leq l'$. □

3.4 Suffix Links

Let us consider the unpruned tree developed in Section 3.2. The matching process described for the tree involves duplicated checking. This is due to the overlap of consecutive substrings. We are checking the presence of subsubstring $Q[i + 1, \cdots, i + q - 1]$ while matching both substrings $Q[i, \cdots, i + q - 1]$ and $Q[i + 1, \cdots, i + q]$. To remove these redundant checks, we include suffix links in the tree structure. The idea of a suffix link was introduced by McCreight [14] to build a time efficient suffix search tree. The suffix link at each node points to the immediate suffix of the substring corresponding to that node. For example, the suffix link at node xX should point to node X. When we match substring xX, we can trace the suffix link and go directly to node X. To check for the next substring, we do not need to start again from the root. We only need to check the last character of the next substring.

To add suffix links in the unpruned tree, traverse to each node xX in the tree and create a suffix link from node xX to node X. The suffix link at the root node points to itself. Fig. 5 shows an example tree with suffix links. Creating a suffix link from a depth-l node requires us to traverse from a root to its suffix. This will take $O(l)$ time. Thus, to construct a suffix link of every node in the tree (with depth q) will take worst-case $O(qN_T)$ time, where N_T is total number of nodes in the tree.

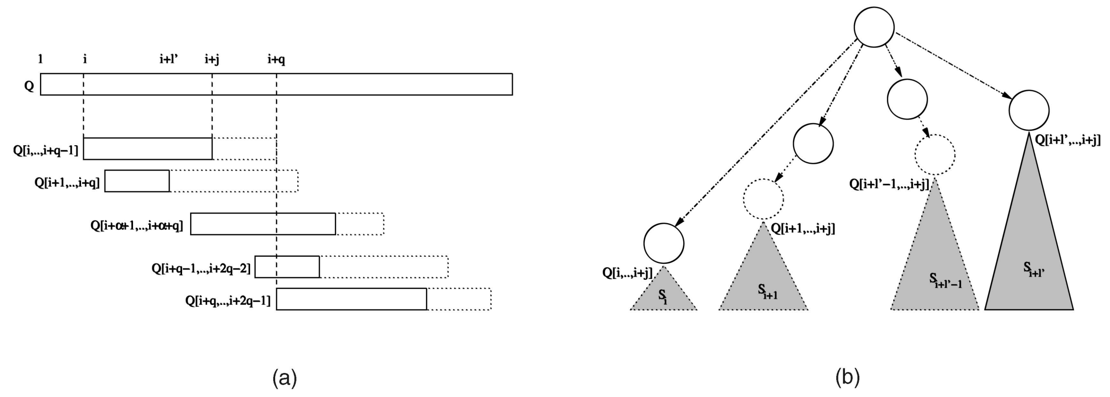


Fig. 4. Proof of correctness of matching algorithm using pruned tree. The paths up to the solid and dash-dot lines are present in the pruned tree. The dotted part is pruned or absent in the tree. (a) Successful matching of substring $Q[i,\cdots,i+q-1]$. There should be no mismatch before character $Q[i+q]$. (b) Subtrees $S_i,\cdots,S_{i+l'-1}$ are similar (identical up to length $q-j-1$) to subtree $S_{i+l'}$. All these subtrees except $S_{i+l'}$ are pruned.

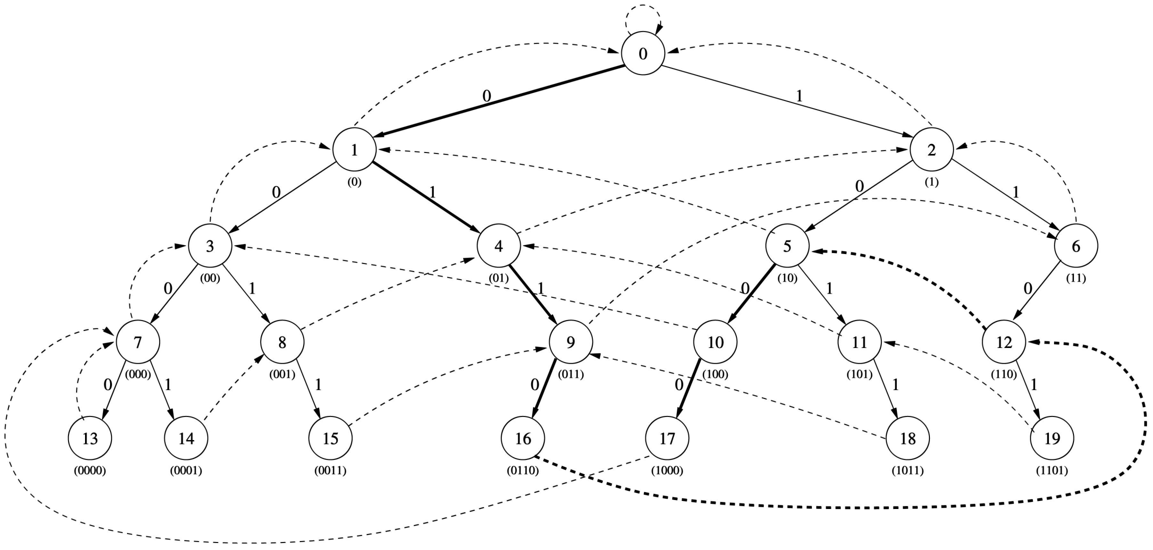


Fig. 5. Tree shown in Fig. 1 with suffix links. Solid lines are the regular links to child nodes. Dashed lines are suffix links. The path traced by the bold solid and dashed lines, (0,1,4,9,16,12,5,10,17), is the path followed while matching query 011000. A mismatch is found at node-12 where there is no child for character 0.

Consider the substring matching problem with query $Q[1,\cdots,m]$. For the first substring, we start from the root node and go down until we find a mismatch or reach a leaf node. If we reach a leaf node $Q[i,\cdots,i+q-1]$ after matching character $Q[i+q-1]$, we accept the substring $Q[i,\cdots,i+q-1]$. To match the next substring $Q[i+1,\cdots,i+q]$, we traverse the suffix link and try to match character $Q[i+q]$.

If we find a mismatch while trying to match character $Q[i+l+1]$ at node $Q[i,\cdots,i+l]$, we reject $Q[i,\cdots,i+q-1]$. To match the next substring $Q[i+1,\cdots,i+q]$, we traverse the suffix link to reach node $Q[i+1,\cdots,i+l]$ and continue the matching process by checking for character $Q[i+l+1]$. We can continue in a similar fashion until we reach the end of the query. A detailed description of substring matching using $\mathcal{T}_s$ is shown in Algorithm 3.

Algorithm 3 q-gram matching using $\mathcal{T}_s$

```
Set the current node as the root of tree $\mathcal{T}_s$
for $i = 1$ to $|Q| - q + 1$ do
    while the current node does not have a child for character $Q[i]$ and we have not reached the root node do
        suppose the current node is at depth $j$
        reject the current substring
            $Q[i-j,\cdots,i-j+q-1]$
        traverse the suffix link and match character $Q[i]$ for next substring
    end while
    if the current node is the root node and we still do not have a child for character $Q[i]$ then
        character $Q[i]$ is not in the tree, start matching with next character $Q[i+1]$
    end if
    traverse to the child node for character $Q[i]$
    if the current node is a leaf node then
        accept the current substring $(Q[i-q+1,\cdots,i])$
        traverse its suffix link
    end if
end for
```

The complete path traced while matching query (011000) is shown in bold in Fig. 5. Before proving the equivalence of substring matching with trees $\mathcal{T}$ and $\mathcal{T}_s$, we would like to state some properties that are hold during substring matching using $\mathcal{T}_s$.

Lemma 3.1. *If while matching query $Q[1,\cdots,m]$ using $\mathcal{T}_s$, we have processed up to $Q[i]$, then the current node is the longest suffix of $Q[1,\cdots,i]$ present in the tree $\mathcal{T}_s$.*

Proof: [by Induction]. For $i=1$, it is trivial to see that we will be at the longest suffix of string $Q[1]$ after processing $Q[1]$. If character $Q[1]$ is present in the tree, we will progress to node $Q[1]$. If $Q[1]$ is not present in the tree, we will be at the root node. Suppose the lemma holds for some $i=k<m$ and the current node $X=Q[k-l+1,\cdots,k]$ is the longest suffix of $Q[1,\cdots,k]$ present in the tree. Suppose the longest suffix of $Q[1,\cdots,k+1]$ present in the tree is $Q[k-l'+1,\cdots,k,k+1]$, $0\le l'\le l$. In this case, node $Q[k-l'+1,\cdots,k]$ will have child $Q[k+1]$. Also, none of $Q[k-j+1,\cdots,k]$, $l'<j<l$ will have child $Q[k+1]$; otherwise, $Q[k-j+1,\cdots,k+1]$ is the longest suffix of $Q[1,\cdots,k+1]$ present in the tree. We want to prove that after processing $Q[k+1]$ we will reach node $Q[k-l'+1,\cdots,k,k+1]$. While trying a match for $Q[k+1]$, starting from node X, we will traverse through $l-l'$ suffix links checking for child $Q[k+1]$. We will find a match only when we reach node $Q[k-l'+1,\cdots,k]$. At this point, we will traverse the child node of $Q[k-l'+1,\cdots,k]$ and reach node $Q[k-l'+1,\cdots,k+1]$. Thus, we will be at the longest suffix node of $Q[1,\cdots,k+1]$ after processing $Q[k+1]$. By induction, the lemma should hold for all $i\le m$. □

Theorem 3.2. *Substring matching using $\mathcal{T}_s$ is equivalent to substring matching using $\mathcal{T}$.*

Proof. To show the equivalence of the two matching algorithms, we will prove that, if a substring $Q[i,\cdots,i+q-1]$ in a query is accepted by $\mathcal{T}$, then it will be accepted by $\mathcal{T}_s$ and vice versa. If substring $Q[i,\cdots,i+q-1]$ is accepted by tree $\mathcal{T}$, then there exists a leaf node $Q[i,\cdots,i+q-1]$ in the tree which is also the longest possible suffix of $Q[1,\cdots,i+q-1]$ present in the tree. While matching the query using tree $\mathcal{T}_s$, after we process $Q[i+q-1]$, according to Lemma 3.1, we will reach the longest suffix node of $Q[1,\cdots,i+q-1]$ present in the tree. From the above, this node is the leaf node $Q[i,\cdots,i+q-1]$. Thus, tree $\mathcal{T}_s$ will also accept substring $Q[i,\cdots,i+q-1]$.

Also, if tree $\mathcal{T}_s$ accepts $Q[i,\cdots,i+q-1]$, it means that, after processing $Q[i+q-1]$, we will be at depth-q leaf node, say X. According to Lemma 3.1, X should also be the longest suffix of $Q[1,\cdots,i+q-1]$. But, length q suffix of $Q[1,\cdots,i+q-1]$ is $Q[i,\cdots,i+q-1]$. This means that node X corresponds to node $Q[i,\cdots,i+q-1]$. Since node $Q[i,\cdots,i+q-1]$ is present in the tree, substring $Q[i,\cdots,i+q-1]$ will also be accepted by the tree $\mathcal{T}$. □

The above matching algorithm is linear to the length of the query. For a traversal of a child link in the tree, we are successfully matching a character and progressing to the next character. For a suffix link traversal from a nonleaf node, we are successfully finding a substring mismatch and we can proceed to the next substring while rejecting the current substring. For a suffix link traversal from a leaf node, we are successfully finding a matching substring and we can proceed to the next substring while accepting the current substring. There are $|Q|$ characters in the query and $|Q|-q+1$ substrings. Thus, the total number of link traversals including child nodes and suffix links is at most $|Q|+(|Q|-q+1)=O(|Q|)$.

3.5 Pruned Trees with Suffix Links

As with the original tree, we would like to add suffix links in the pruned tree to make the matching process faster. But, we cannot make a direct link from a node xX to its immediate suffix node X because it is possible that the node X was deleted in the pruning process. Consider a node $Q[i,\cdots,i+j]$ present in the pruned tree. The longest suffix of $Q[i,\cdots,i+j]$ present in the tree is $Q[i+l',\cdots,i+j]$. Suppose we reach node $Q[i,\cdots,i+j]$ while matching a substring. While matching the next $l'-1$ substrings, we will not match beyond $Q[i+j]$ because the path is pruned before $Q[i+j]$ (see Fig. 4b). Since we have already matched up to $Q[i+j]$ successfully, there is no need to check these $l'-1$ substrings. Thus, when we reach node $Q[i,\cdots,i+j]$, we can directly go to node $Q[i+l',\cdots,i+j]$ and continue checking. Thus, the suffix link at the node $Q[i,\cdots,i+j]$ points to the node $Q[i+l',\cdots,i+j]$.

To add suffix links in a pruned tree, traverse to each node X in the tree and create a suffix link from node X to node Y, where Y is the longest suffix node of X present in the pruned tree. The suffix link at the root node points to itself. Fig. 6 shows a pruned tree with suffix links. Substring matching using tree $\mathcal{T}_{sp}$ is very similar to substring matching using tree $\mathcal{T}_s$.

Consider the substring matching problem with query $Q[1,\cdots,m]$. For the first substring, we start from the root node and go down until we find a mismatch or reach a leaf node. If we reach a depth-q node (a leaf node) $Q[i,\cdots,i+q-1]$ after matching character $Q[i+q-1]$, we accept the substring $Q[i,\cdots,i+q-1]$. To match the next substring $Q[i+1,\cdots,i+q]$, we traverse the suffix link and try to match character $Q[i+q]$.

Suppose we reach a leaf node $Q[i,\cdots,i+l]$, $l<q-1$, after matching character $Q[i+l]$. This means that the children of this node were pruned during the redundancy pruning. We mark substring $Q[i,\cdots,i+q-1]$ as *pending*. To match the next substring, we traverse the suffix link and try to match character $Q[i+l+1]$.

On the other hand, if we find a mismatch at node $Q[i,\cdots,i+l]$ for character $Q[i+l+1]$, then the substrings

$$Q[i-q+l+2,\cdots,i+l+1],$$
$$Q[i-q+l+3,\cdots,i+l+2],\cdots,Q[i,\cdots,i+q-1]$$

are marked *rejected*. After marking the substring, we traverse the suffix link of node $Q[i,\cdots,i+l]$ and continue matching $Q[i+l+1]$ for the next substring $Q[i+1,\cdots,i+q]$. At the end, all the substrings that are not marked *rejected* are *accepted*. The formal description of the matching algorithm is shown in Algorithm 4.

Algorithm 4 q-gram matching using $\mathcal{T}_{sp}$

```
Set current node as root of the tree $\mathcal{T}_{sp}$
for $i = 1$ to $|Q| - q + 1$ do
  while the current node does not have a child for
```

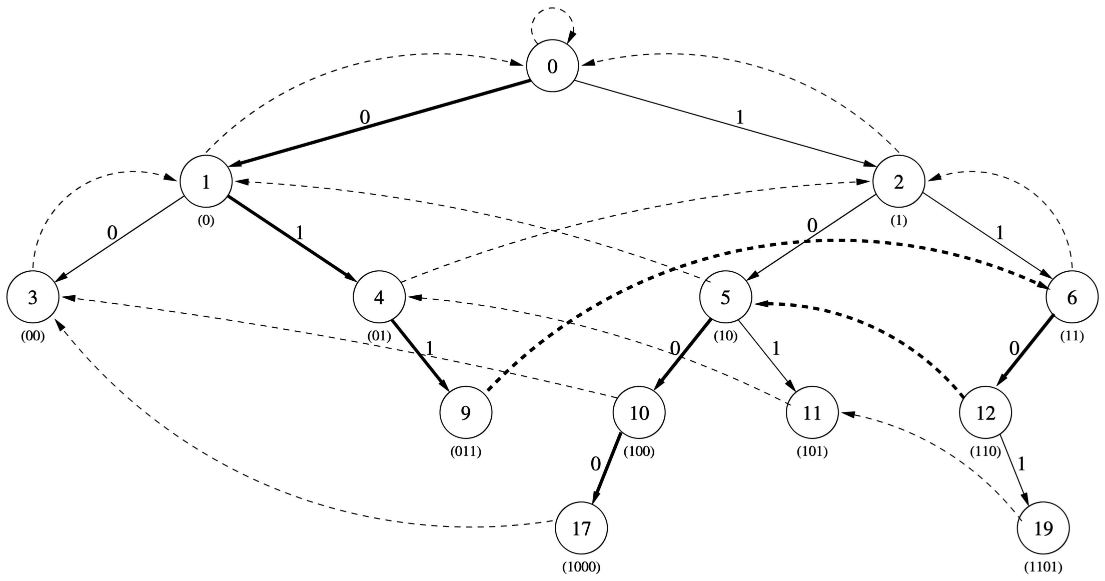


Fig. 6. Suffix links added to the pruned tree shown in Fig. 3. Solid lines are child links and dashed are suffix links. Node-17's immediate suffix node-7 was pruned from the tree. So, it points to immediate suffix node of node-7. Bold lines show the path traced by matching algorithm for query (011000). At node-12, we found a mismatch and substring 1100 was marked *rejected*.

```
  character $Q[i]$ or we have not reached the root node
  do
    /* A mismatch is found at current node for character
       $Q[i]$*/
    suppose the current node is at depth $j$
    for $k = 0$ to $q - j - 1$ do
      if substring $Q[i-j-k, \cdots, i-j-k+q-1]$ is
         marked $pending$ or is unmarked then
         mark substring $Q[i-j-k, \cdots, i-j-k+q-1]$
         as $rejected$
      else
         break
      end if
    end for
    traverse the suffix link
  end while
  if the current node is the root node and we still do not
     have a child for character $Q[i]$ then
     character $Q[i]$ is not in the tree, start matching with
     next character $Q[i+1]$
  end if
  traverse to the child node for character $Q[i]$
  if the current node is a leaf node then
    if the current node is at depth $q$ then
      mark the current substring $(Q[i-q+1, \cdots, i])$ as
      $accepted$
    else
      mark the current substring $(Q[i-q+1, \cdots, i])$ as
      $pending$
    end if
    traverse the suffix link
  end if
end for
mark all the $pending$ and unmarked substrings as
  $accepted$
```

The path traced while matching the query $Q = (011000)$ for the tree in Fig. 6 is shown in bold lines. Before proving the equivalence of substring matching with trees $\mathcal{T}_p$ and $\mathcal{T}_{sp}$, we would like to state some properties of substring matching using $\mathcal{T}_{sp}$.

Lemma 3.2. *If while matching query $Q[1, \cdots, m]$, we have processed up to $Q[i]$, then the current node is the longest suffix of $Q[1, \cdots, i]$ present in tree $\mathcal{T}_{sp}$.*

Proof: [by Induction]. For $i = 1$, it is trivial to see that we will be at the longest suffix of string $Q[1]$ after processing $Q[1]$. If character $Q[1]$ is present in the tree, we will progress to node $Q[1]$. If $Q[1]$ is not present in the tree, we will be at the root node. Suppose the lemma holds for some $i = k < m$ and the current node $X = Q[k-l+1, \cdots, k]$ is the longest suffix of $Q[1, \cdots, k]$ present in the tree. Suppose the longest suffix of $Q[1, \cdots, k+1]$ present in the tree is $Q[k-l'+1, \cdots, k, k+1]$, $0 \leq l' \leq l$. In this case, node $Q[k-l'+1, \cdots, k]$ will have child $Q[k+1]$ and nodes $Q[k-j+1, \cdots, k]$, $l' < j \leq l$ are either pruned or do not have child $Q[k+1]$. While trying a match for $Q[k+1]$, we will travel through suffix links checking for child $Q[k+1]$. We will find a match only when we reach node $Q[k-l'+1, \cdots, k]$. At this point, we will traverse child node of $Q[k-l'+1, \cdots, k]$ and reach node $Q[k-l'+1, \cdots, k+1]$. Thus, we will be at the longest suffix node of $Q[1, \cdots, k+1]$ after processing $Q[k+1]$. By induction, lemma should hold for all $i \leq m$. □

Theorem 3.3. *Substring matching using $\mathcal{T}_{sp}$ is equivalent to substring matching using $\mathcal{T}_p$.*

Proof. To show the equivalence of two matching algorithms, we will prove that if a substring $Q[i, \cdots, i+q-1]$ is marked *accepted* by $\mathcal{T}_p$, then it will be marked *accepted* by $\mathcal{T}_{sp}$, and vice versa. We will also prove that, if a mismatch is found while matching substring $Q[i, \cdots, i+q-1]$ by $\mathcal{T}_p$, then a mismatch will be found by $\mathcal{T}_{sp}$, and vice versa. From these two observations, it is implied that if a query is marked first as *pending* in $\mathcal{T}_p$ and accepted (or rejected) later, it will be marked as *pending* in $\mathcal{T}_{sp}$ and accepted (or rejected) later, and vice versa.

If substring $Q[i, \cdots, i+q-1]$ is marked *accepted* by tree $\mathcal{T}_{sp}$, then after processing $Q[i+q-1]$, we will be at depth-q leaf node, say X. According to Lemma 2, X should also be the longest suffix of $Q[1, \cdots, i+q-1]$. But, length q suffix of $Q[1, \cdots, i+q-1]$ is $Q[i, \cdots, i+q-1]$. This means that node X corresponds to node $Q[i, \cdots, i+q-1]$. Since node $Q[i, \cdots, i+q-1]$ is present in the tree, substring $Q[i, \cdots, i+q-1]$ will be accepted by tree $\mathcal{T}_p$.

If substring $Q[i, .., i+q-1]$ is marked *accepted* by tree $\mathcal{T}_p$, then there exists a leaf node $Q[i, \cdots, i+q-1]$ in the tree, which is also the longest suffix of $Q[1, \cdots, i+q-1]$ present in the tree. While matching query using tree $\mathcal{T}_{sp}$, after we process $Q[i+q-1]$, according to Lemma 2, we will reach the longest suffix node of $Q[1, \cdots, i+q-1]$ present in the tree. From the above, this node is a leaf node $Q[i, .., i+q-1]$. Thus, $\mathcal{T}_{sp}$ will accept substring $Q[i, .., i+q-1]$.

If $\mathcal{T}_{sp}$ finds a mismatch for substring $Q[i, \cdots, i+q-1]$ at node $Q[i, \cdots, i+j]$, then node $Q[i, \cdots, i+j]$ does not have child $Q[i+j+1]$. While matching $Q[i, \cdots, i+q-1]$ using $\mathcal{T}_p$, we will reach node $Q[i, \cdots, i+j]$, and find a mismatch while looking for child $Q[i+j+1]$.

Suppose $\mathcal{T}_p$ finds a mismatch for substring $Q[i, \cdots, i+q-1]$ at node $Q[i, \cdots, i+j]$. While matching using $\mathcal{T}_{sp}$, after we match character $Q[i+j]$, according to Lemma 2, we will be at the longest suffix node of $Q[1, \cdots, i+j]$. Suppose the longest suffix node is $Q[i-l, \cdots, i+j]$, $0 \leq l \leq q-j-1$. Since $Q[i+j+1]$ is not a child of $Q[i, \cdots, i+j]$, $Q[i+j+1]$ cannot be a child of $Q[i-l', \cdots, i+j]$, $\forall 0 \leq l' \leq l$. Thus, we will find a mismatch at node $Q[i-l, \cdots, i+j]$ while matching substring $Q[i-l, \cdots, i-l+q-1]$ and traverse the suffix link and again try to match $Q[i+j+1]$. We will continue to find mismatch, traverse suffix links, and will eventually reach node $Q[i, \cdots, i+j]$ and still cannot match character $Q[i+j+1]$. Thus, we have found a mismatch for $Q[i, \cdots i+q-1]$. □

Corollary 3.1. *Substring matching with $\mathcal{T}_{sp}$ is equivalent to substring matching using $\mathcal{T}$.*

Proof. This is followed immediately from Theorem 3.1 and Theorem 3.3. □

The above matching algorithm is linear to the length of the query. For every child link traversal, we successfully match a character and progress to the next character. For every suffix link traversal from a leaf node, we mark the current substring as *accepted* or *pending* and proceed to the next substring. For every suffix link traversal from a nonleaf node, we find a substring mismatch and proceed to match the next substring after marking the previous overlapping *pending* substrings as *rejected*. Finally, every substring is marked either once, *accepted* or *rejected*, or twice, first *pending* and then *accepted* or *rejected*. There are $|Q|$ characters in the query and $|Q|-q+1$ substrings. Thus, the total number of link traversals including child and suffix links is at most $|Q| + 2 \times (|Q|-q+1) = O(|Q|)$.

4 String Matching

4.1 Exact String Matching

If T is the text and Q is the query, then the exact string matching problem is to find if Q is a substring of T. We will prove that we can use the above q-gram matching for exact string matching. But first, we will prove some results on the number of nodes at a given depth of the tree. We will assume that the text is converted to an infinite length string by adding an infinite number of end symbol (\$) at the end.

Theorem 4.1. *For any depth l, $N(l) \leq N(l+1)$, where $N(l)$ is the number of nodes at depth l in tree $\mathcal{T}$.*

Proof. Every node X in the tree should have at least one child x. Otherwise, the text string will end after we encounter X in the text. But, this is a contradiction because X is assumed to be an infinite length string. Thus, for every node at depth l, we will have at least one node at depth $l+1$. Thus, the number of nodes at depth $l+1$ is at least $N(l)$. □

Theorem 4.2. $\exists l_0, s.t. N(l_0) = N(l_0+1)$.

Proof. Assume $N(l) < N(l+1)$ for all l. For $l = |T|+2$, where $|T|$ is the length of the original text without end symbols added at the end, we will have $N(l) > |T|+1$. But the number of unique substrings cannot be more than $|T|+1$. We have reached a contradiction. Thus, $\exists l_0, s.t. N(l_0) = N(l_0+1)$. □

Theorem 4.3. *If, for some depth $l_0, N(l_0+1) = N(l_0)$, then*

$$N(l+1) = N(l) = N(l_0), \forall l \geq l_0. \tag{1}$$

Proof. If we go down the tree, the number of nodes will increase if and only if there is at least one node with two or more children. Suppose the theorem does not hold for some $l \geq l_0$, that is, $N[l+1] > N[l]$ and node $X[1, \cdots, l]$ at depth l has two children x and x'. Since the substrings $X[1, \cdots, l]x$ and $X[1, \cdots, l]x'$ are present in the tree, their suffix $X[l-l_0, \cdots, l]x$ and $X[l-l_0, \cdots, l]x'$ should also be present in the tree. Then, the depth l_0 node $X[l-l_0, \cdots, l]$ should have two children for x and x'. Thus, $N(l_0+1) > N(l_0)$. This is a contradiction. Thus, we have proved the theorem. □

4.1.1 Exact String Matching Algorithm

Given a text T, make it an infinite length string by adding special end symbols at the end. Build a (l_0+1)-depth tree where the number of nodes stops increasing after depth l_0. Given a query Q, perform q-gram matching with $q = l_0+1$. If no mismatch is found, then the query is a substring of the text; otherwise, it is not.

Theorem 4.4. *Query Q is a substring of text T if and only if no mismatch is found while performing q-gram matching with $q = l_0+1$, where $N(l_0+1) = N(l_0)$.*

Proof. It is easy to see that if the query is a substring of the text, then there will be no mismatch. We need to prove that, if Q is not a substring of the text, then we will have at least one mismatching q-gram. Suppose the maximum prefix match of a query in the text starts at ith position in T and the length of matched prefix is l. If $l \leq l_0$, then substring $Q[i, \cdots, i+l_0]$ is not present in the text and

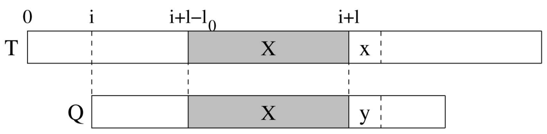


Fig. 7. String matching of text T and query Q. X is longest prefix match of Q present in T. Alphabets x and y are different.

while performing q-gram matching, we will find a mismatch for this substring. The case where $l > l_0$ is shown in Fig. 7. Alphabet x and y are different. Since the substring Xx is in the text, it should be present in the tree. Since node X is depth-l_0 node, it cannot have two children; otherwise, $N(l_0) < N(l_0 + 1)$. Thus, substring Xy cannot be present in the tree because, otherwise, depth l_0 node X will have two children, namely, x and y. So, we will find a mismatch at node X in the tree while matching character y from the query. Therefore, the exact matching problem is the same as q-gram matching. □

4.2 Variable Length Matching

The tree model can be used to perform q-gram matching where q is the same as the maximum depth of the tree. But in many cases, one does not know what sequence length is appropriate. Also, one might need to use different values of q depending on the policy and the situation. For example, in intrusion detection, the value of q is determined by the current required false alarm rate and detection rate. One may want to have larger q for a better detection rate and sometimes a smaller q for a small false alarm rate. We would not want to create a separate tree for each sequence length. This would require more resources and might be cumbersome to maintain.

Given a desired maximum sequence length (say L), we can construct a pruned tree with suffix links as proposed earlier. This tree can be used to perform substring matching of lengths smaller or equal to L. While matching for length $q < L$, we can assume that all the nodes at depth q are leaves and there are no nodes below depth q. Now, we can use the same algorithm proposed above for length q substring matching. Consider the problem of matching the text 1000011011 (the original tree is shown in Fig. 1) with query 011000 for sequence length 3. All 3-grams in the query are present in the text and we should not find any mismatch. Fig. 8 shows the path followed by the algorithm while matching 011000 for sequence length 3.

5 Experiments and Results

We use performance measurements to show the effectiveness of the pruning algorithm. We demonstrate the savings in space because of pruning. We also show the average time complexity of substring matching using the pruned tree $\mathcal{T}_p$. These two experiments also help us understand the use of tree redundancy pruning to reduce the matching overhead when using pruned tree $\mathcal{T}_p$.

5.1 Data Set

For our experimentation purpose, we use the system call sequence data sets made available by the University of New Mexico (UNM). These data sets were generated for the stide intrusion detection system developed at UNM [1]. The character size is the number of different possible system calls, which is equal to 182. In order to collect the data, the given program was started and requests were sent to this program. These requests were either sent by a real user or automatically generated by a program. During the processing of requests, the program makes some calls to the underlying operating system. We can use the *trace* command to record these system calls made by the program in the order they were invoked. The properties of the data sets including their sizes are shown in Table 1. Each data set consists of multiple sequences generated by multiple runs of the program. As the name suggests, *synthetic sendmail* and *synthetic ftp* data sets were generated at UNM using automatically generated synthetic requests. *Real Sendmail* data was generated by an MIT sendmail server with real user requests. Since the MIT system call trace was very large, only a small fraction (consisting of 5.3 million data points) was used in our experiments. Further information about the data set may be obtained from [1].

To observe the effect of tree redundancy pruning on other data sets, we also generated some random data. In particular, we generated random data with independent Bernoulli distribution and Markovian distribution. Mathematical formulation for Bernoulli distribution is shown in (2).

$$s_i = \begin{cases} 1, & \text{with probability } p; \\ 0, & \text{with probability } 1 - p, \end{cases} \tag{2}$$

where p is the probability of occurrence of 1 in data.

In a Markov model, the current output depends on the current state of the model. For simplicity, we are assuming a simple Markov model with binary output. The current output depends on values of the previous two outputs. The sequence satisfies the following stochastic distribution:

$$s_i = \begin{cases} 1, & \text{if } (s_{i-2}, s_{i-1}) = (0,0); \\ 0, & \text{if } (s_{i-2}, s_{i-1}) = (1,1); \\ 1 \text{ with prob. } p_1, & \text{if } (s_{i-2}, s_{i-1}) = (0,1); \\ 1 \text{ with prob. } p_2, & \text{if } (s_{i-2}, s_{i-1}) = (1,0), \end{cases}$$

where p_1 and p_2 are two prescribed probabilities.

The properties of Bernoulli and Markovian data sets are shown in Table 2. We used two separate Bernoulli data sets with different values of p.

Fig. 9 shows the number of unique sequences in the different data sets. As we expect, the number of unique sequences increases as we increase the sequence length. The MIT data set shows more diversity and the number of unique sequences increases rapidly. For the ftp and sendmail data set, the number of unique sequences remains very low and increases very slowly. For Markovian and Bernoulli data sets, the number of unique sequences increases very fast. This is due to the random nature of these data sets. For the Bernoulli data set, the number of unique sequences stops increasing rapidly after sequence length 24 when it approaches its maximum value of $|T| - q$, where $|T|$ is the number of data points and q is the sequence length.

5.2 Experiments

To determine the average time complexity of substring matching using a pruned tree $\mathcal{T}_p$, we calculate the average depth of leaf nodes in the tree. This gives us the expected time taken to match a substring. The average depth of a tree can be computed as

$$Avg.Depth = \frac{\sum_{x \in Leaves} depth(x)}{|Leaves|}, \tag{3}$$

where $Leaves$ is the set of all leaf nodes.

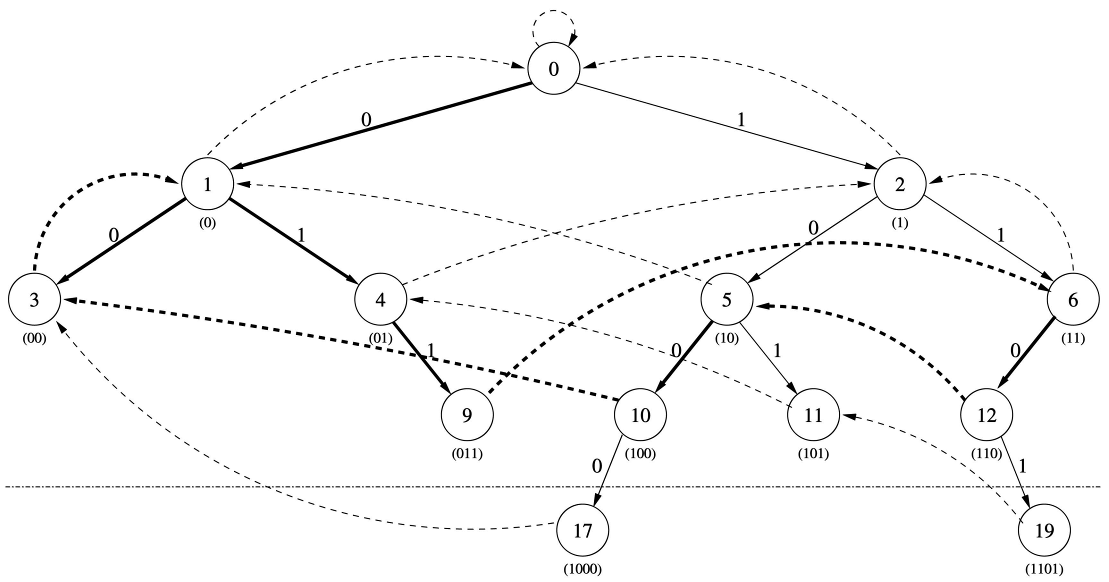


Fig. 8. Length-3 substring matching of query 011000 using the tree in constructed in Fig. 6. All the nodes at depth 3 are considered leaves and all nodes below them (i.e., nodes 17 and 19) are considered nonexistent. When we reach nodes 12 and 10, we do not try to go any further down to $\mathrm{depth\text{-}4}$ node. Instead, we traverse the suffix links and continue matching the next character for the next length-3 substring.

The space required to store a tree is directly proportional to the number of nodes and links in the tree. For each node in the tree, there is one incoming link from its parent and one outgoing suffix link. Thus, the number of links is twice the number of nodes. Thus, the space required to store the tree is roughly three times the number of nodes in the tree. In order to compare the space requirements of the unpruned and pruned trees, we calculate the number of nodes for both. We also compared the average space requirements for tree structures with those for hash-tables. Assuming the load factor of hash-table is 1, the minimum space required by the hash-table is $q \times \#(entries\ in\ hash\text{-}table)$, where q is the sequence length. The number of entries in the hash-table is equal to the number of unique substrings in the text.

$$Space_{tree} = 3 \times N_T, \tag{4}$$

$$Space_{hash} = q \times \#(\text{unique substrings of length q}). \tag{5}$$

5.3 Results

Fig. 10 shows the effect of pruning on the average depth of the tree for different sequence lengths. Every leaf node in the unpruned tree is at depth q, where q is the sequence length. Thus, the average depth of the unpruned tree is the same as the sequence length for all data sets. Compared with the unpruned tree, the average depth of the pruned tree increases slowly as we increase the sequence length. Thus, even if we increase the sequence length, there is a small increase in the amount of processing required to match a substring using pruned tree $\mathcal{T}_p$. The effect of pruning is more apparent for *ftp* and *sendmail.daemon* data as we increase the sequence length. Pruning reduces the average depth by very small value for the Markovian data set. The subtrees rooted at the bottom of the tree are similar to the subtrees rooted at their suffix nodes. Therefore, they are pruned during the redundancy pruning. Thus, the average depth reduces by a small value. For the Bernoulli data set, pruning works very well for small sequence lengths. This is because, for small q, Bernoulli data contains all the possible $\mathrm{length}\text{-}q$ substrings. Thus, after pruning, the tree reduces to a $\mathrm{depth\text{-}1}$ tree with just three nodes. For larger sequence lengths, there is not much gain from pruning for Bernoulli data because the data is random in nature. After sequence length 24 when the number of unique sequences stops increasing rapidly, Bernoulli data again shows significant reduction in average depth.

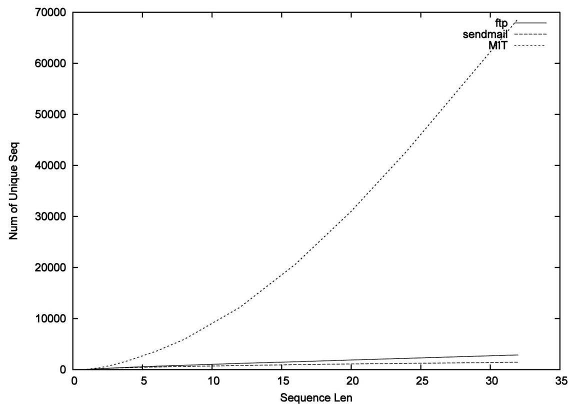
The space required to store the unpruned and pruned tree is shown in Fig. 11. It also shows the minimum space required for a hash-table based matching scheme. For a practical hash-tables with few collisions, we would require more space. The space requirement of the unpruned tree is slightly more than the hash-table scheme. But for the pruned tree, the space requirement is much less. This means that many of the nodes are being pruned during the

TABLE 1
System Call Data

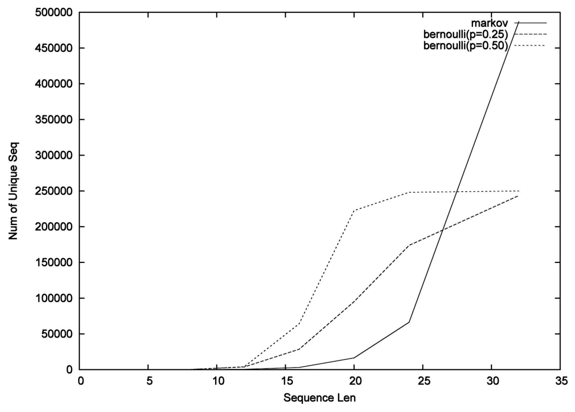
DataSet	Filename	Number of System Calls
Synthetic Sendmail	sendmail.daemon	1.556M
Synthetic FTP	nonself1	180,315
Real Sendmail	MIT	5.23M

TABLE 2
Bernoulli and Markovian Data Set

DataSet	Parameters	Number of data points
Bernoulli	p = 0.25	250,000
Bernoulli	p = 0.50	250,000
Markovian	p1 = 0.25, p2 = 0.75	5,000,000

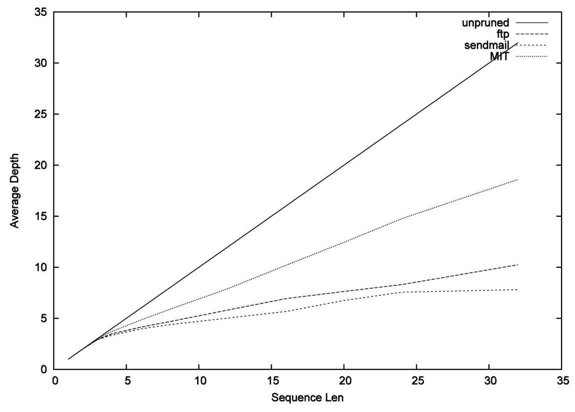


(a)

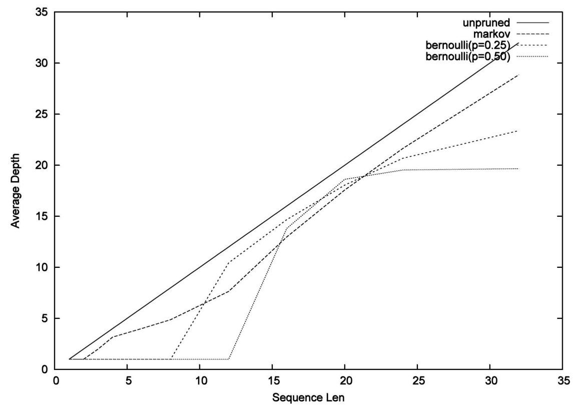


(b)

Fig. 9. Number of unique sequences in a data set. (a) System call data. (b) Bernoulli and Markovian data.



(a)



(b)

Fig. 10. Average depth as a function of sequence length. Solid lines in the figure corresponds to the average depth of the unpruned trees. The average depth of the unpruned trees is equal to the sequence length for all the data sets. (a) System call data. (b) Bernoulli and Markovian data.

pruning process. For larger sequence lengths, pruning reduces the space complexity by an order of 10. Thus, pruning yields a large space saving.

The space requirement for Markovian and Bernoulli data sets is shown in Fig. 11d, Fig. 11e, and Fig. 11f, respectively. For both data sets, the space requirement of the unpruned tree is smaller than the hash-table. For Bernoulli data sets, the space requirement of hash-table eventually becomes less than that of the unpruned tree for higher sequence lengths. As for the system call data, pruning shows a large savings in space and reduces the space requirement by four times. Change in space requirements is more visible for larger sequence lengths.

5.4 Comparison with Rabin-Karp

In addition to the above experiments, we also compared our tree model with the Rabin-Karp-based hashing technique. For this purpose, we implemented both algorithms in C++. For the given training data, we built a pruned tree with suffix links and performed matching for the testing data. For the Rabin-Karp algorithm, for the given training data, we generated hashes for all the given length substrings present in the data and stored them in the hash-table. While matching the test data, we efficiently generated the successive hashes and checked if that hash value was present in the hash-table. If the hash value was found, we considered it as a match. We did not perform further exact matching. If we did not find the value in the table, we got a mismatch. For hashing purposes, we considered a substring as a number in some base. The hash value of the substring was simply the number represented by the substring modulo a big prime. Equations (6) and (7) show the hashing and rehashing mechanism used in our implementation.

$$hash_{X_i} = (x_i b^{q-1} + \cdots + x_{i+q-1}) \bmod p, \tag{6}$$

where X is the string, X_i is the ith substring, x_i is the ith character in the string, p is a random prime, and b is the base.

$$hash_{X_{i+1}} = ((hash_{X_i} - b^{q-1}x_i) \times b + x_{i+q}) \bmod p. \tag{7}$$

The value of b^{q-1} can be precalculated once at the start. The above rehashing technique is efficient and takes constant time to calculate. Thus, the first hash can be calculated in time $O(q)$ and further hashes can be calculated in constant time using (7). Thus, the calculation of hashes of all the q-grams in the query will take $O(|Q|)$ time, which is linear to the length of the query. We used the same data sets as earlier for comparison. We computed the time taken for preprocessing of the data for both models and compared their runtime overhead for testing the data. For simplicity,

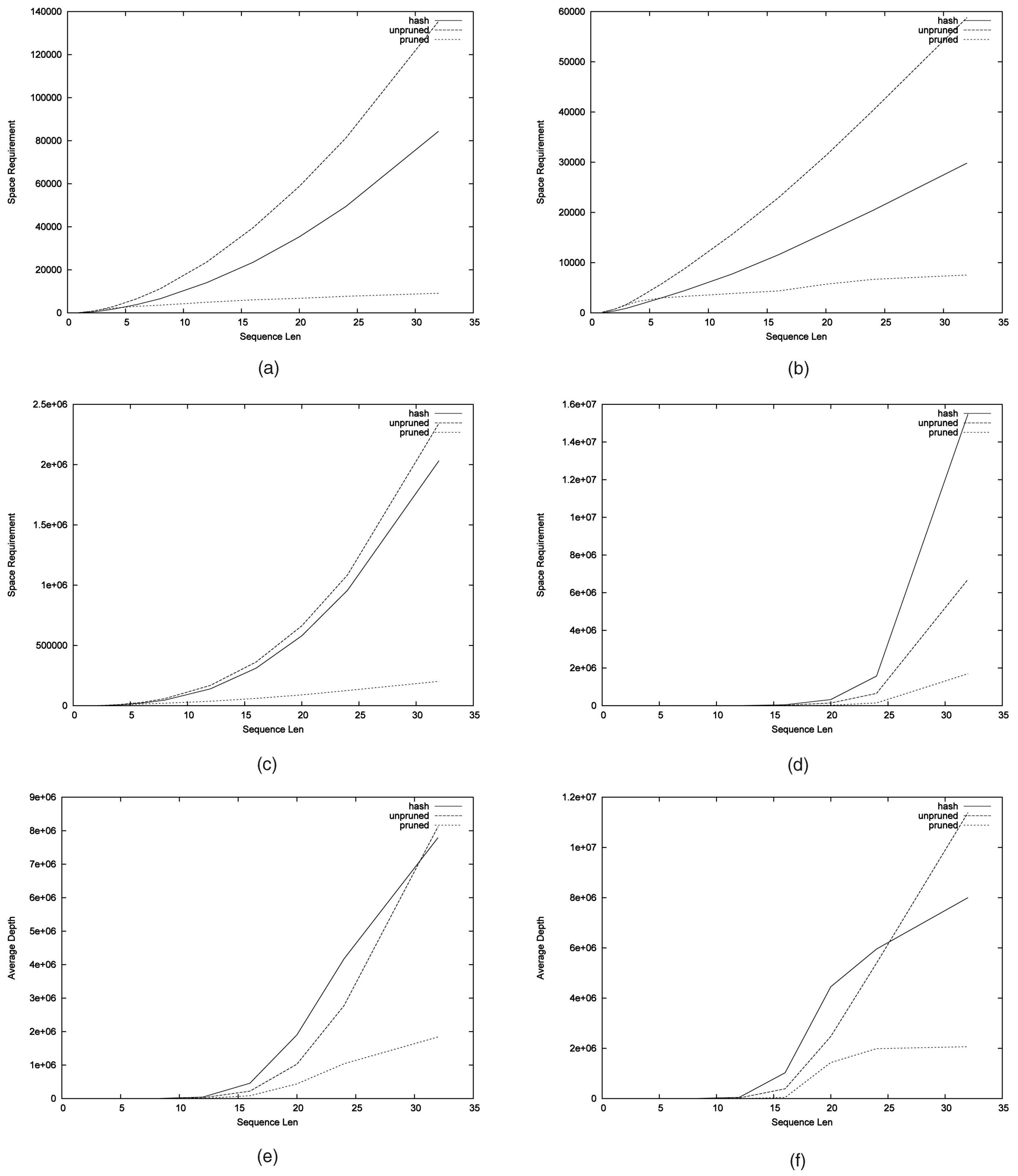


Fig. 11. Space complexity of different data sets. (a) FTP data. (b) Sendmail.daemon data. (c) MIT data. (d) Markovian data. (e) Bernoulli data (p = 0.25). (f) Bernoulli data (p = 0.50).

we used the same set of data for both training and testing. For efficiency, we set b to 256 and p to a 32-bit random prime.

The time taken to train and perform substring matching is shown in Fig. 12. For the tree-based algorithm, the training time is calculated as the sum of the time taken to record the unique sequences, build the tree, create suffix links, prune the suffix tree, and adjust the suffix links. The training time for a tree model is proportional to the square of the size of the tree. The training time of the Rabin-Karp algorithm depends on the time to compute the hash and the time to store the hash in a table. Computing hashes takes constant time. Thus, the training time of Rabin-Karp does not change much as we increase the sequence length. Compared with Rabin-Karp, the tree model takes more time to train, and the time increases with sequence length. This is acceptable because training is an offline process and is performed just once.

The matching time is almost constant for both algorithms and does not change with sequence length. This is because the matching time of both the Rabin-Karp algorithm and the tree model are independent of the sequence length. The

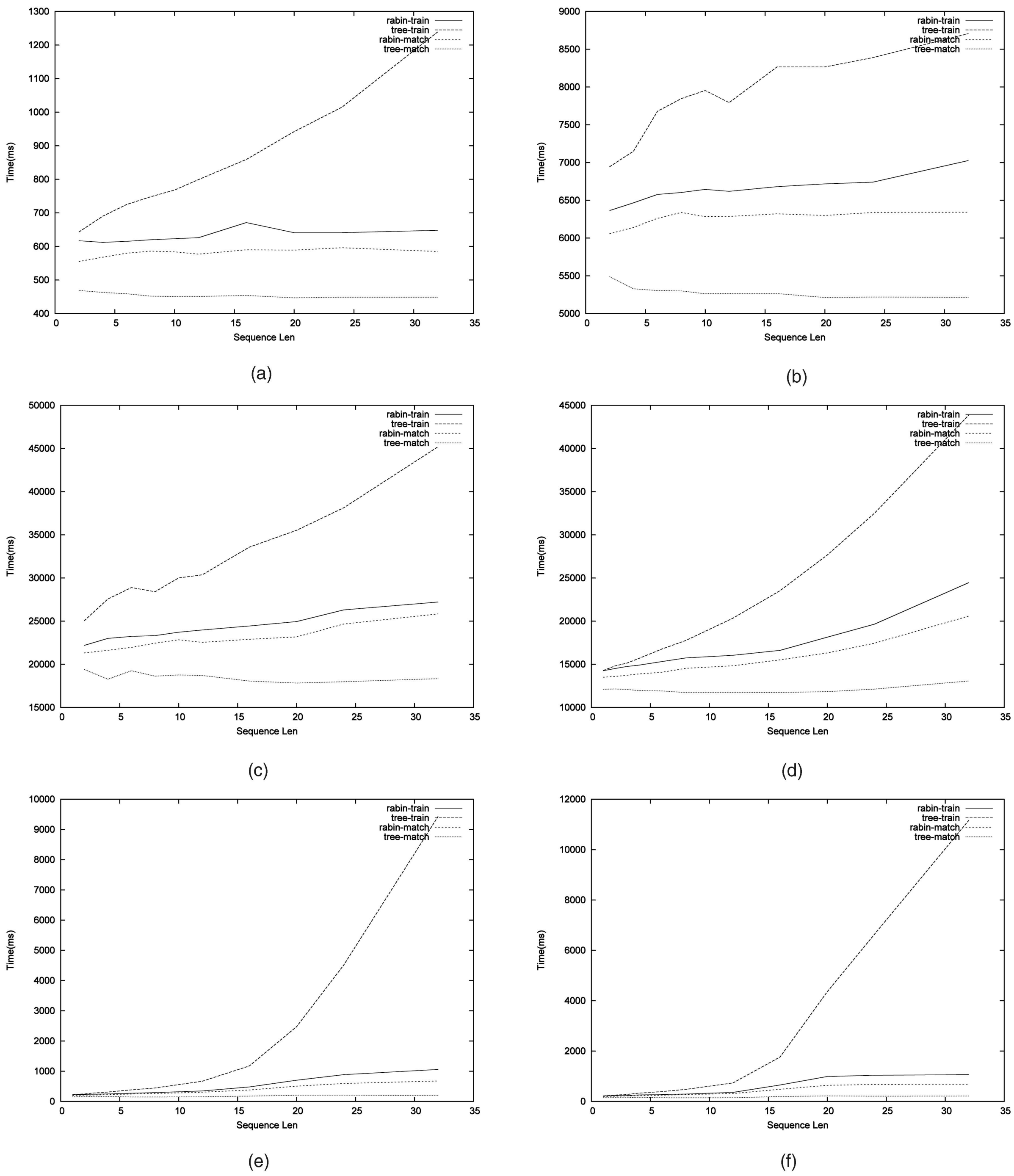


Fig. 12. Training and matching time for substring matching. (a) FTP data. (b) Sendmail.daemon data. (c) MIT data. (d) Markovian data. (e) Bernoulli data (p = 0.25). (f) Bernoulli data (p = 0.50).

matching time complexity of both algorithms is linear to the length of the query. The tree model takes considerably less time for matching than Rabin-Karp. This is because computation of rehash takes more time than traversing a link in the tree. Since matching is an online process, even a small time saving can be a significant advantage.

6 Conclusion

When applied to q-gram matching problems with huge text size, the time requirement of previous string matching algorithms become unacceptable. The Rabin-Karp algorithm is the only known algorithm which works well for q-gram matching. But, the Rabin-Karp algorithm can produce false matches because of the limitations of hashing. Also, for different values of q, the Rabin-Karp algorithm needs to create separate hash-tables. This may require a large amount of space.

In this paper, we presented a fast q-gram matching algorithm. The algorithm preprocesses the text and stores it in a tree structure that is efficient for storage and addition of new substrings. We also presented a pruning algorithm to reduce the size of the tree. Suffix links were added to the tree structure to facilitate a linear time substring matching algorithm. In addition, we have proven that we can perform an exact string match using a tree of sufficient length. Finally, we performed experiments on system call sequence data as well as Bernoulli and Markovian data to show the effect of pruning on runtime and space overheads.

Construction of tree $\mathcal{T}$ takes $O(q|T|)$ time. For the worst case, redundancy pruning and adding suffix links takes $O({N_T}^2)$ and $O(N_T q)$ time, respectively. The matching algorithm using tree $\mathcal{T}$ takes $O(q|Q|)$ time. q-gram matching with the pruned tree $\mathcal{T}_p$ has the worst case of $O(q|Q|)$ time. But as seen in Fig. 10, the expected time for matching is very small for system call data. When using tree with suffix links, $\mathcal{T}_s$ or $\mathcal{T}_{sp}$, the matching algorithm is linear to the size of query and does not depend on the size of text. For system call data, the space requirement of the unpruned tree with suffix links $\mathcal{T}_s$ is little more than that of the hash-table-based q-gram matching system. But, for the pruned tree with suffix links $\mathcal{T}_{sp}$, the space requirement is very modest and is much smaller than that required for a hash-table. Even for large sequence lengths, the space required to store a pruned tree is less than the size of the text. Only for Bernoulli data, the space requirement of the pruned tree is more than the size of the text. Our tree algorithm has better matching time than the Rabin-Karp algorithm. Also, our tree model has the additional advantage in its ability to perform multiple length q-gram matching using the same tree.

REFERENCES

[1] University of New Mexico System Call Data Set, http://cs.unm.edu/~immsec/systemcalls.htm, 2006.

[2] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM,* vol. 18, no. 6, pp. 333-340, June 1975.

[3] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *Comm. ACM,* 20, no. 10, p. 762, Oct. 1977.

[4] S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, and M. Vingron, "q-Gram Based Database Searching Using a Suffix Array (Quasar)," *Proc. Third Ann. Int'l Conf. Computational Molecular Biology,* pp. 77-83, 1999.

[5] W.I. Chang and T.G. Marr, "Approximate String Matching and Local Similarity," *Proc. Fifth Ann. Symp. Combinatorial Pattern Matching,* pp. 259-273, 1994.

[6] M.T. Chen and J. Seiferas, "Elegant and Efficient Subword Tree Construction," *Combinatorial Algorithms on Words,* pp. 97-107, 1985.

[7] J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *Proc. DARPA Information Survivability Conf. and Exposition (DISCEX II '02),* vol. 1, pp. 367-373, 2001.

[8] R. Cole and R. Hariharan, "Approximate String Matching: A Simpler Faster Algorithm," *SIAM J. Computing,* vol. 31, no. 6, pp. 1761-1782, 2002.

[9] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes," *Proc. 1996 IEEE Symp. Research in Security and Privacy,* pp. 120-128, 1996.

[10] Z. Galil and R. Giancarlo, "Improved String Matching with K-Mismatches," *SIGACT News,* vol. 17, no. 4, pp. 52-54, 1986.

[11] R.M. Karp and M.O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM J. Research and Developement,* pp. 249-260, 1987.

[12] D.E. Knuth, J.H. Morris, and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing,* vol. 6, no. 1, pp. 323-360, 1977.

[13] G.M. Landau and U. Vishkin, "Efficient String Matching with K-Mismatches," *Theoretical Computer Science,* pp. 239-249, 1986.

[14] E.M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM,* vol. 23, no. 2, pp. 262-272, 1976.

[15] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys,* vol. 33, no. 1, pp. 31-88, 2001.

[16] G. Navarro and R. Baeza-Yates, "Fast and Practical Approximate String Matching," *Information Processing Letters,* vol. 59, pp. 21-27, 1996.

[17] G. Navarro and R. Baeza-Yates, "Faster Approximate String Matching," *Algorihtmica,* vol. 23, no. 2, pp. 127-158, 1999.

[18] D.M. Sunday, "A Very Fast Substring Search Algorithm," *Comm. ACM,* vol. 33, no. 8, pp. 132-142, 1990.

[19] E. Sutinen and J. Tarhio, "On Using Q-Gram Locations in Approximate String Matching," *Proc. Third Ann. European Symp. Algorithms,* pp. 327-340, 1995.

[20] J. Tarhio and E. Ukkonen, "Boyer Moore Approach for Approximate String Matching," *Proc. Second Scandinavian Workshop Algorithm Theory,* pp. 348-359, 1990.

[21] E. Ukkonen, "Approximate String-Matching with Q-Grams and Maximal Matches," *Theoretical Computer Science,* vol. 92, no. 1, pp. 191-211, 1992.

[22] E. Ukkonen, "On-Line Construction of Suffix Trees," *Algorithmica,* vol. 14, pp. 249-260, 1995.

[23] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. IEEE Symp. Switching and Automata Theory,* pp. 1-11, 1973.

[24] S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *Comm. ACM,* vol. 35, pp. 83-91, 1992.

[25] S. Wu, U. Manber, and E.W. Myers, "A Subquadratic Algorithm for Approximate Limited Expression Matching," *Algorithmica,* vol. 15, no. 1, pp. 50-67, 1996.

Prahlad Fogla received the BSc degree in computer science from the Indian Institute of Technology in 2000 and the MSc degree in computer science from the Georgia Institute of Technology in 2003. He is a PhD candidate in computer science at the Georgia Institute of Technology. His research interests include computer security and worm detection.

Wenke Lee received the PhD degree in computer science from Columbia University in 1999. He is an associate professor in the College of Computing at the Georgia Institute of Technology. His research interests include systems and network security, network management, applied cryptography, and data mining. He received a Best Paper Award (in applied research category) at the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) in 1999 and a US National Science Foundation CAREER Award in 2002. He is a member of the IEEE Computer Society.