Efficient Algorithms for Similarity Search

S. Rajasekaran[1], Y. Hu[1], J. Luo[1], H. Nick[2],

P.M. Pardalos[3], S. Sahni[1], and G. Shaw[2]

Univ. of Florida

Abstract: The problem of our interest takes as input a database of m sequences from an alphabet Σ and an integer k. The goal is to report all the pairs of sequences that have a matching subsequence of length at least k. We employ two algorithms to solve this problem. The first algorithm is based on sorting and the second is based on generalized suffix trees. We provide experimental data comparing the performances of these algorithms. The generalized suffix tree based algorithm performs better than the sorting based algorithm.

Key words: Generalized Suffix Tree (GST), Color Set Size (CSS), Quick sort, Radix sort.

1 Introduction

We consider the following sequence analysis problem: Given a database of m sequences and an integer k find all pairs of sequences that share a common subsequence of length at least k. This problem was communicated to us by Dr. David Lipman (a senior scientist at NCBI) and has numerous applications in biology. Tools for identifying similarities among biological sequences are used by biologists on a regular basis. One example is the BLAST program housed at NIH.

In this paper we provide two different solutions to this problem. The first solution is based on sorting and the second is based on generalized suffix trees (GSTs). Our experimental data indicate that the GST based algorithm is faster. The preprocessing time is more for the GST based algorithm. Since this is done only when the trees are built, this is not a drawback.

[1]Department of Computer and Information Science and Engineering

[2]Department of Neuroscience

[3]Department of Industrial Systems Engineering

In Section 2 we provide a summary of the sorting based algorithm. Section 3 is devoted to a discussion of the GST based algorithm. In Section 4 we compare the two algorithms empirically. Section 5 provides some concluding remarks.

2 Sorting Based Algorithm (SBA)

The idea of SBA is to identify substrings of length k from each sequence; sort all these substrings in alphabetic order (so that identical subsequences come closer); scan through the sorted list to identify identical subsequences and hence output the relevant pairs.

A detailed description of the algorithm follows. Input are m sequences $s_1, s_2, \ldots, s_m$ from some alphabet Σ and an integer k.

1. Let $sum = 0$. Employ an array $len[1 : m]$. Initializing each element of $len[\ ]$ to zero;
2. For every i, $1 \leq i \leq m$, read the input string s_i and let $sum = sum+$ the number of substrings of length k in s_i.
3. Use an array $A[1 : sum]$.
4. Put all subsequences of length k from the input sequences into A.
5. Use radix sort algorithm to sort all the substrings in A according to alphabetical order. When sorting, the subsequences are divided into several subsections. After finishing a subsection's sorting, the total substrings are examined to discard the substrings which are surely not matched by other substrings. Continue with the sorting of remaining subsequences.
6. Scan through A to find all the subsequences which have a match with the neighbouring subsequences.

3 GST Based Algorithm (GSTBA)

Suffix trees have been employed in the past to solve string matching problems and approximation string matching problems in sequence and in parallel.

A suffix tree is a trie-like data structure representing all suffixes of a string. Its linear time construction algorithm is described in [5].

Each edge in the suffix tree is labeled with a symbol from the alphabet Σ. The sequence of labels in any path from the root to a leaf corresponds to a suffix of the string. We can also associate a string with every node (not necessarily leaves) in the tree. We can say that the node represents the corresponding string. Note that this string need not be a suffix.

A suffix tree for the sequence *agcata* is shown in Figure 1.

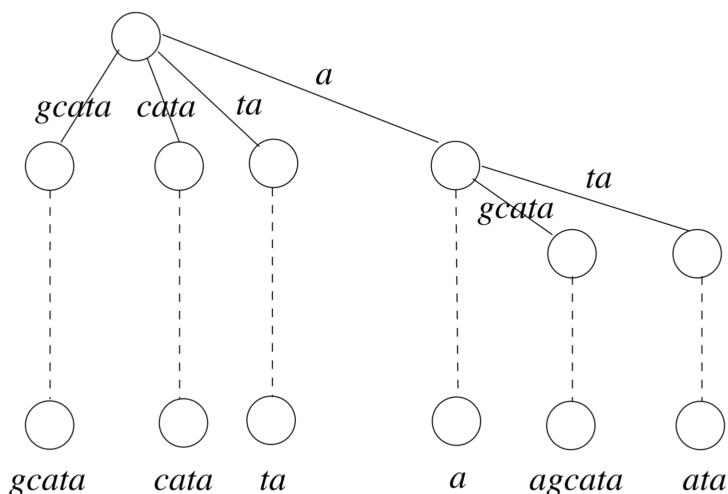

Figure 1: A suffix tree for the string *agcata*

Suffix trees have the following properties: Let s be the sequence under consideration. 1) Every suffix of s is represented by a leaf in the tree. 2) Let x and y be two nodes in the tree. If x is an ancestor of y, then the string that x represents is a prefix of the string that y represents. 3) If the nodes $x_1, x_2, \ldots, x_k$ represent the strings $X_1, X_2, \ldots, X_k$, then the least common ancestor of $x_1, x_2, \ldots, x_k$ represent the longest common prefix of $X_1, X_2, \ldots, X_k$.

We can also conceive of a suffix tree corresponding to multiple sequences and we end up with a GST. In a GST, every suffix of every sequence is represented as a leaf. The properties mentioned for suffix trees hold for GSTs as well. If multiple sequences have the same suffix, then there will be a leaf corresponding to each such sequence. A GST

can be constructed in time linear in the total length of all input strings. Figure 2 shows a GST for the strings *gatc*, *atc*, and *catag*.

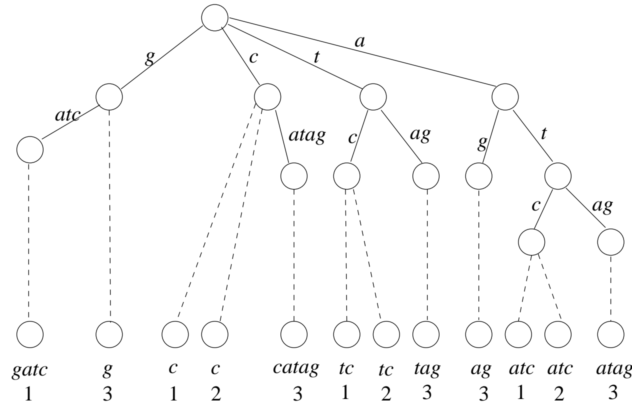

Figure 2: GST for *gatc*, *atc*, *catag*

We will employ the GST data structure to solve the sequence analysis problem of our interest. One of the problems that arises is the *color set size (CSS)* problem introduced by [1].

3.1 Color Set Size (CSS) Problem

Let T be any tree in which each leaf has been colored with a color from the set $\{1, 2, \ldots, q\}$. The *color set size* of any node v in T, denoted as $css(v)$ is the number of different colors that can be found in the subtree rooted at v. The *color set size* problem is to determine the color set size of each internal node in T.

For example, in Figure 3, the color set size of the nodes p, m, n are $3, 2, 3$, respectively.

An optimal $O(n)$ time algorithm for the CSS problem has been given in [1]. Here n is the number of nodes in the tree. This algorithm can be summarized as follows. Let $LeafList$ refer to the list of leaves in T ordered according to a post-order traversal of the tree. The $LeafList$ of the tree in Figure 3 is a, b, c, d, e, f, g, h. Also, for any leaf x of color c, let $LastLeaf(x)$ stand for the last leaf that precedes x in $LeafList$ which has the same color c. In Figure 3, $LastLeaf(c) = a$; $LastLeaf(f) = d$; $LastLeaf(e) = Nil$; and so on.

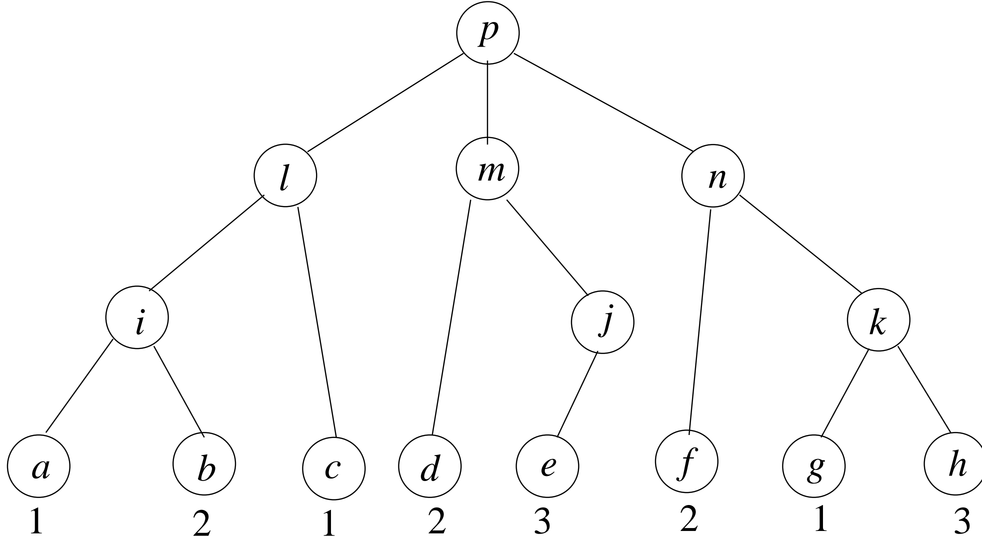

Figure 3: A tree with colored leaves

For any vertices u and v in T, let $LCA(u, v)$ stand for the least common ancestor of u and v. For example, $LCA(a, c) = l$; $LCA(d, e) = m$; $LCA(d, h) = p$; and so on in Figure 3. For any node u in T, $Subtree(u)$ stands for the subtree rooted at u and $LeafCount(u)$ stands for the number of leaves in $Subtree(u)$. An internal vertex x is said to be a $ColorPairLCA$ (of color c) if there exist leaves u and v of color c with $LastLeaf(v) = u$ and $x = LCA(u, v)$. In the tree of Figure 3, the node l is a $ColorPairLCA$ of color 1 (due to the pair a, c). The nodes m and n are not $ColorPairLCA$s for any color.

For any internal node x of T let $CPL\text{-}Count(x) = k$ if among all colors there are k leaf pairs for which x is their $ColorPairLCA$. For example, in Figure 3, $CPL\text{-}Count(p) = 5$ (due to the pairs $(a, c), (g, c), (d, b), (f, d)$, and (h, e). $CPL\text{-}Count(m) = 0$. Also let $Duplicate(x) = \sum_{v \in Subtree(x)} CPL - Count(v)$.

It can be shown that for any node x in T, $css(x) = LeafCount(x) - Duplicate(x)$. The linear time algorithm for CSS proceeds to compute $LeafList$, $LeafCount()$, $LastLeaf()$, $CPL\text{-}Count()$, and $Duplicate()$, in this order. Finally, it computes $css()$. Computation of $LeafList$, $LeafCount$, and $Duplicate$ are straight forward and can be done using a post-order traversal of the tree. $LastLeaf()$ can be computed for every node as follows. Traverse $LeafList$ from left to right. Keep an array $last()$ such that

$Last(c)$ is the last seen node of color c. whenever a leaf x is encountered of color c, set $LastLeaf(x) = Last(c)$ and update $Last(c)$.

To compute *CPL-Count*(), we make use of the constant time algorithm for computing least common ancestors [4]. For every leaf x compute $u = LCA(x, LastLeaf(x))$ and increment *CPL-Count*(u) by 1. Once we have all the above values $css()$ can be computed easily.

3.2 The GSTBA algorithm

The sequence analysis problem of our interest can be solved now as follows. First construct a GST corresponding to all the database sequences. A simple algorithm for constructing a suffix tree for a single string can be found in [3] which is based on the algorithm of [5]. We have generalized this algorithm in a nontrivial way. Because in GST each internal node contains a pointer to the next prefix of current prefix node and each leaf node may represent the same suffix that occurs in multiple strings, the implementation of GST will be different from that of the single string suffix tree. Note that, when dealing with the last character of a string, the construction procedure should not stop at the endpoint until it reaches root node so that each leaf node gets the right color size. The color given to any leaf (i.e., suffix) is nothing but the sequence number that the suffix comes from.

After constructing a GST, we solve the CSS problem for this GST. In order to solve the sequence analysis problem of our interest, we do the following. We perform a breadth-first search in the GST. At each node x, if its distance from the root is $\geq k$, x is an internal node, and $css(x) > 1$, we report all the relevant pairs. This is done using two pointers *start_point* and *end_point* that index to the first and last leaves of the subtree rooted at x. Clearly this algorithm runs in time $O(n+q)$ where n is the number of nodes in the GST and q is the size of the output. Thus we arrive at the following Theorem.

Theorem 3.1 *The sequence analysis of our interest can be solved in time $O(n+q)$, where n is the size of the GST and q is the size of the output. The preprocessing time is $O(n)$ and the memory needed is also $O(n)$.*

4 An Experimental Comparison of SBA and GSTBA

In this section we report experimental data obtained in the comparison of the two algorithms SBA and GSTBA. These results were collected on the Sun workstation Ultra Enterprise 4000. This machine has a memory of 2 GB and its operating system is SunOS 5.6. Both GSTBA and SBA were tested on randomly generated DNA sequences.

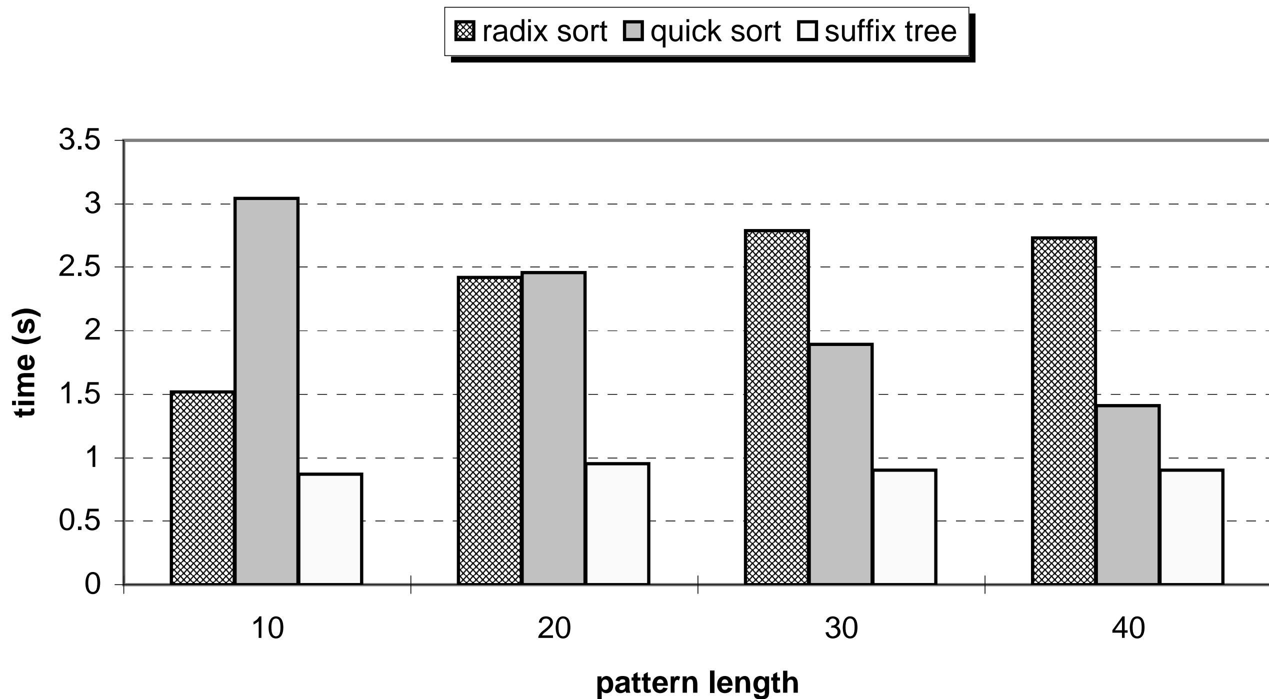

Figure 4: First set

Figures 4, 5, 6, and 7 show the results of our experiments. These graphs compare GSTBA with SBA. Two different versions of SBA were built, one based on radix sort and the other based on quicksort. For a description of radix and quick sort algorithms please see [2].

Figure 4 corresponds to the case of 2000 strings. The maximum string length was 100. The pattern length ranged from 10 to 40 in increments of 10.

Figure 5 describes the situation when the total number of strings is 2000, the maximum string length is 500, and the pattern length varies from 10 to 40.

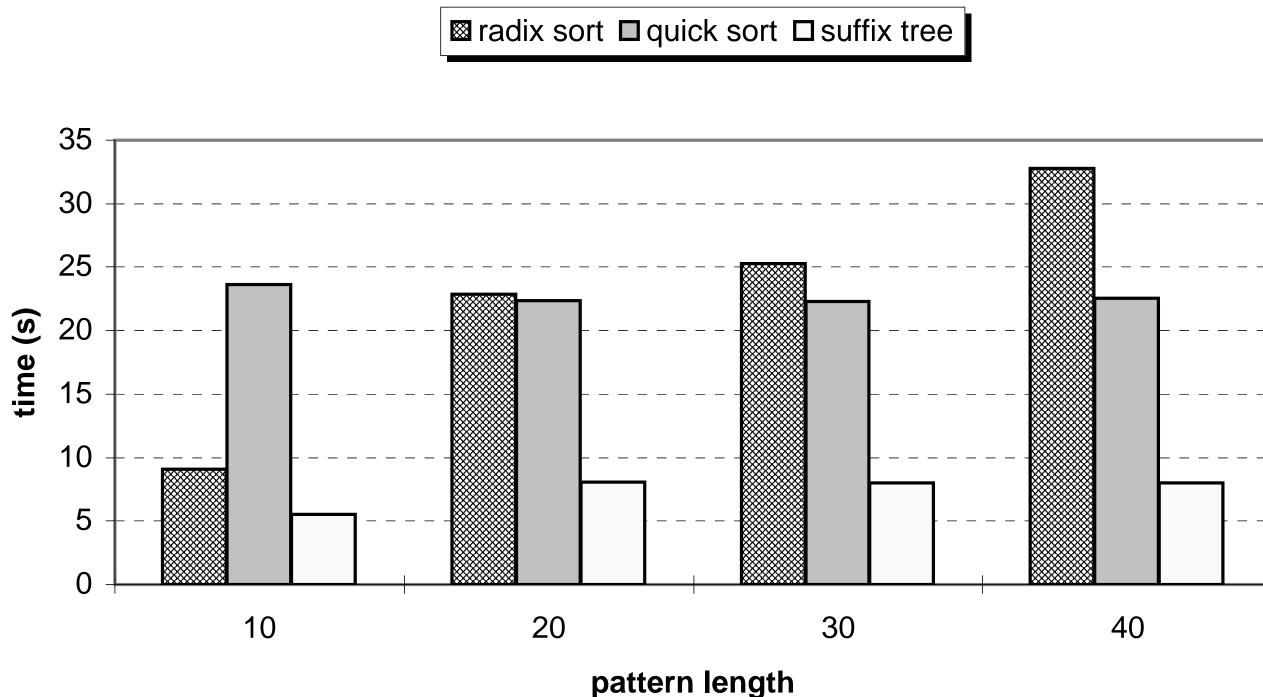

Figure 5: Second set

The third (Figure 6) and fourth (Figure 7) sets of data also had 2000 strings each and the maximum string lengths were 1000 and 2000, respectively. In these cases the pattern length was varied from 10 to 40 in increments of 10.

As is seen from these graphs, GSTBA is always faster than the SBA (both versions). Also note that as the database size increases, the difference between GSTBA and SBA increases with the GSTBA being the clear winner. Biological data nowadays tend to be voluminous and databases with gigabytes of data are commonplace. When GSTBA and SBA are applied to such large databases, we can expect GSTBA to be two orders of magnitude faster than the SBA. We should point out here that the preprocessing time for GSTBA was larger than that needed for SBA. But this is acceptable since preprocessing is done only once when the GST is constructed.

It is conceivable that the database changes on a regular basis with more and more sequences being added. In such cases, the GST also will have to be updated. But fortunately, the update time is only proportional to the amount of new data that is being added.

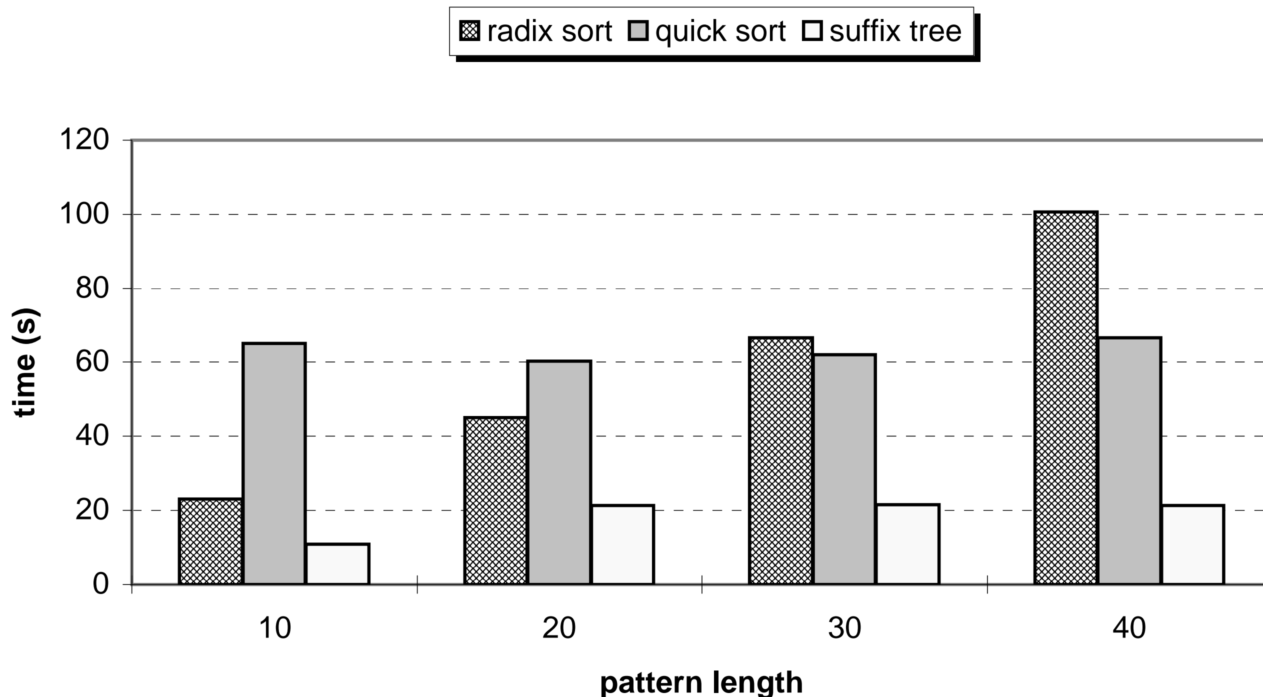

Figure 6: Third set

5 Conclusions

In this paper we have considered two different algorithms for the problem of identifying similar pairs of sequences in a given database. One was based on sorting (SBA) and the other was based on generalized suffix trees (GSTBA). Experimental data were obtained for comparing the performances of these algorithms. We conclude, from these data, that the GSTBA is faster than SBA. GST has the added advantages of linear space and linear update time. An interesting open problem is to extend the above algorithms for cases where the matches we are looking for can have gaps.

Acknowledgement

We are grateful to Dr. David Lipman for communicating to us the problem considered in this paper. We are also thankful to him for many fruitful discussions and encouragement. SBA is based on a discussion that the first author had with Dr. Lipman.

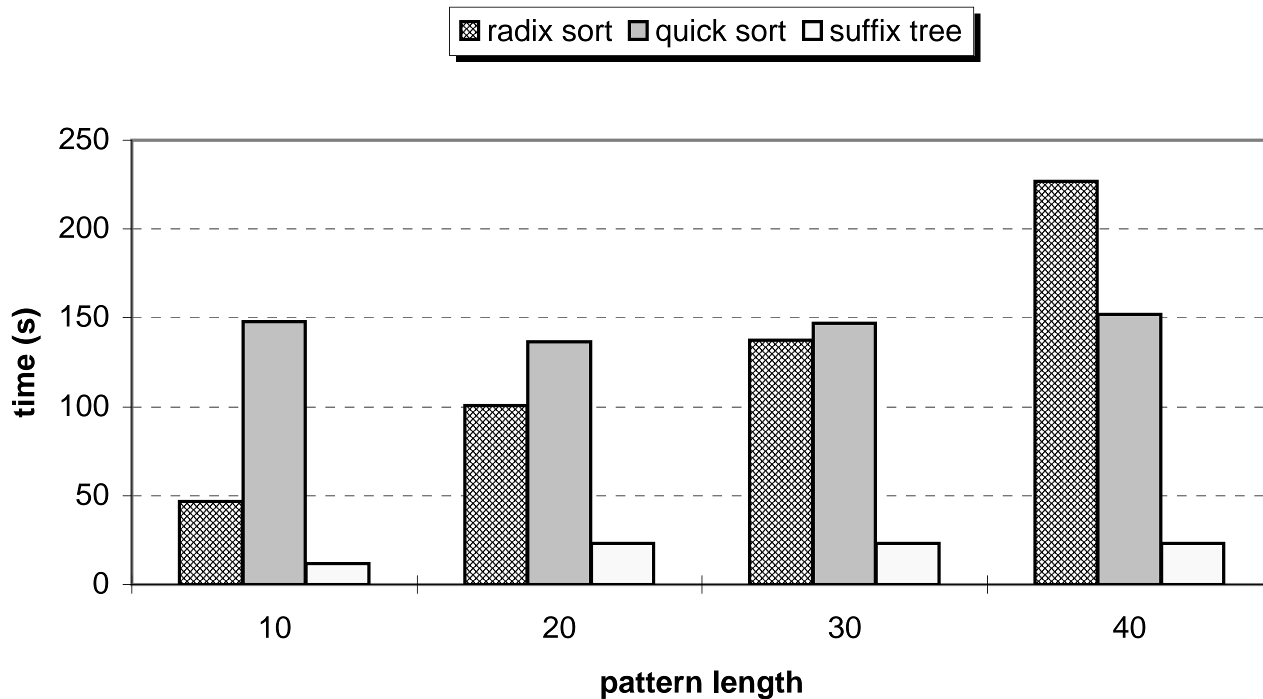

Figure 7: Fourth set

References

[1] L. Chi and K. Hui, Color Set Size Problem with Applications to String Matching, Technical Report, Dept. of Computer Science, University of California, Davis, 1992.

[2] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, W.H. Freeman Press, 1998.

[3] M. Nelson, Fast String Searching With Suffix Trees, *Dr. Dobb's Journal*, August 1996.

[4] B. Schieber and U. Vishkin, On finding lowest common ancestors:simplification and parallelization, *SIAM J. Comput.* 17(6), December 1988, pp. 1253-1262.

[5] E. Ukkonen, On-Line Construction of Suffix Trees, *Algorithmica*, 14(3), 1995, pp. 249-260.