An Efficient Data Structure for Searching Strings

Marek Karpinski * Yakov Nekrich †

Abstract. We describe an efficient data structure for searching a set of strings of arbitrary length. Search and update operations for a string of length d can be performed in simultaneous time $O(\log n + d)$ and space $O(n)$ where n is the number of strings. The best up to now known data structures required a superlinear space for the same time bound. This data structure could be of particular interest for the case of sets containing long strings. We conclude with some experimental results and some benchmark comparisons with other search methods.

*Dept. of Computer Science, University of Bonn. Work partially supported by DFG grants , PROCOPE project and IST grant 14036 (RAND-APX). E-mail marek@cs.uni-bonn.de

†Dept. of Computer Science, University of Bonn. Work partially supported by IST grant 14036 (RAND-APX). E-mail yasha@cs.uni-bonn.de

1 Introduction

Management of a set of strings is a part of a large number of computer applications. Examples of such applications are e.g. the creation of text indices, management of text collections and data compression. With the growing number of electronic texts the importance of that problem obviously increases.

Data structures for searching in a dynamic or static set of strings were considered in a number of papers and a number of efficient approaches have been developed.

Tries [B59],[F60] provide a fast but space-intensive solution. Several papers (for instance, [M68]) describe approaches for the compaction of search tries, but the space requirements for dynamic tries still remain very high.

Ternary search trees (`TST`) [BS79],[V84],[BS96] give us another efficient solution to the string searching problem. A similar approach is also described in [M79]. In a TST every node n corresponds to some string prefix s and contains a character c and three pointers. Three descendants of n correspond to strings that are lexicographically less then, equal to or greater than sc. Ternary search trees require additional space proportional to the total number of characters in all strings in the worst case.

In [BS79],[V84],[BS96] the problem of searching in a set of strings of fixed length d is considered. Bentley and Saxe show in [BS79] that the search in a perfectly balanced ternary search tree requires $O(\log n + d)$ scalar comparisons. While search algorithm in a TST is extremely simple, rebalancing of a TST is a more complicated task. Vaishnavi [V84] presents an algorithm for multidimensional height-balanced trees. This algorithm can be applied to managing a set of strings of fixed length d. However for the set of strings of arbitrary length the worst case search or update time would be $O(\log n + d_{max})$, where d_{max} is the maximum string length.

The data structure described by Mehlhorn [M79] allows searching in time $O(\log n + |p|)$ for a string p but only at a cost of a non-linear storage in the total number of characters.

A related problem of searching in an array of strings for a string of length d is considered in [H80] and [AHHP00]. In [AHHP00] the authors prove an upper bound of $O((\frac{d \log\log n}{\log\log(4+(d \log\log n/\log n))} + \log n + d)$ and the optimality of this bound.

In this paper we describe an efficient method for searching strings that combines the advantages of both previously described approaches. Our data structure, further called a `differential tree`, achieves for the first time the optimal bound of $O(\log n + |p|)$, for a string p and simultaneously requires only additional linear space in the number of strings. Our data structure is

able to manage strings of *arbitrary* length.

Our differential trees can be combined with different types of height-balanced trees, and use the advantages for different situations. Our approach is similar to search in suffix arrays of Manber and Myers [MM93], but our data structure manages a *dynamic* set of strings. Low constants in the upper bound make this data structure particularly attractive for different applications. The simplicity of the data structure allows for an easy and efficient implementation. Experiments presented in the last section demonstrate practical advantages of the presented data structure in different settings.

2 Algorithm Description

Our data structure is a binary tree, in which every node contains a pointer to a string. In every node N a certain additional information about the preceding string and the strings in the subtree rooted in N is stored. Using this information we can determine for an arbitrary node N and an arbitrary number k , whether the longest common prefix of the string at the node N and the string at the left(right) child of N exceeds k. Below we show, how we can use that information to achieve the claimed time and space bounds and how we can store the information in such a way, that the data structure can be quickly updated.

In this paper the following notation will be used. We denote by $l[i..j]$ the substring of string l consisting of the characters $l[i], \ldots, l[j]$. We denote by the *least significant difference* of two strings l and r ($lsd(l,r)$) the smallest index k, such that $l[k] \neq r[k]$. We say that $l <_k r (l >_k r)$, if $l < r (l > r)$ and $lsd(l,r) = k$. If string l is a proper prefix of a substring r, than $l <_{|l|+1} r$. Notation $l =_k r$ means that $l[1..k] = r[1..k]$.

For the easiness of description we do not distinguish between a node N and the string which the node N points at. For every tree node N $set(N)$ denotes the set of all strings, pointed at by N and its descendants. The `difference index` of node N (`diffind`(N)) is the least significant difference of N and its predecessor in the data structure. If N is the leftmost node the difference index of N equals to $+\infty$. A `minimal difference index` of node N (`mdi`) is the minimal value of difference indices of elements from $set(N)$. `mdi`$(N) = min\{$`mdi`$(N') | N' \in set(N)\}$. For every node N we store its minimal difference index and its difference index. Thus only two additional values must be stored for every tree node.

Below we describe a search algorithm for a string p. First we sketch the general idea and then give a detailed description.

We call the trees **differential**, because of the general underlying principle

of storing information on the differences between neighboring strings. We compare the string at the current node s with the searched string p and find $k = lsd(p, s)$. If $p = s$ the search is completed. Otherwise, we assign T to the left(right) subtree of s if $s[k] < p[k](s[k] > p[k])$. Then we look for the node t in T such that $s[1..k-1] = t[1..k-1]$. We can find such a node by examining the values of `mdi` and `diffind` fields of nodes in T. Next $k' = lsd(t, p)$ is found. However, since we know that $p[1..k-1] = s[1..k-1] = t[1..k-1]$ we only have to compare the substrings $t[k..|t|]$and $p[k..|p|]$. We repeat this procedure until p is found or there is no more strings to search (a tree leaf is reached).

The pseudocode description of the search algorithm is shown on Figure 1.

```
s := root
k := 1
REPEAT STEP 1.

STEP 1:
1     while (s[k] = p[k])
2        k := k + 1
3        if (k − 1 = |p| AND |s| = |p|)
4           return s
5     if (s[k] < p[k])
6        s' := rchild(s)
7           while (diffind(s') < k OR
mdi(lchild(s')) < k )
8           s' := lchild(s')
9        s := s'
10    else
11       if (diffind(s) < k )
12          return NOT FOUND
13       s' := lchild(s)
14       while (mdi(rchild(s')) < k )
15          s' := rchild(s')
16       s := s'
17    if (s = NIL)
18       return NOT FOUND
```

Figure 1: Search algorithm

The variable k can be viewed as a measure of our knowledge about the strings in the data structure. At every algorithm step we know that our data structure contains a string t' such that $(k-1)$-prefix of t' is equal to the $(k-1)$-prefix of p. Besides that, during the execution of our algorithm only values of $s[k], \ldots, s[|s|]$ of strings s will be examined. Thus the key property of our method, is that after a string s' with $s'[1..r] = p[1..r]$ the r-prefixes of the following strings do not have to be examined. The value of k is never decreased. A variable s is used to traverse the tree. At the beginning k is initialized to 1 and s to the root of the tree.

The search procedure starts by comparing s with p. If $s = p$ the search is complete. Otherwise the smallest k such that $s[k] \neq p[k]$ is found.This is achieved by simply incrementing k as long as $s[k] = p[k]$. Suppose that $s[k] < p[k]$. Then the search continues in the right subtree of s. The next string s' we will examine is the right child of s. First, we check if the first $k-1$ characters of s' are equal to the first $k-1$ characters of p. Suppose one of the difference indices of descendants of $\texttt{lchild}(s')$, of $\texttt{lchild}(s')$ or of s' is smaller than k, Then there exists $n \leq k-1$, such that the value of the n-th character has increased. Therefore $s'[n] > s[n] = p[n]$ and the search must continue in the left child of s'. The search procedure chooses left node of s' as the next search node as long as the condition ''`difference index of` s' `or minimal difference index of the left child of` s' `is greater than` $k-1$'' is true, i.e. until the node s' with $s'[1..k-1] = p[1..k-1]$ is found. Then s is assigned to s' and the search continues in s, but from this time only characters $s[k], s[k+1], \ldots$ are considered. The almost symmetric case $s[k] > p[k]$ is described in lines 10-16 of the pseudocode.

Obviously,the algorithm works in $O(H + |p|)$ time, where H is the height of the tree, since after each comparison we either increase the value of k or descend one level down the tree. It can be seen from the algorithm description that the total number of comparisons does not exceed $4H + |p|$.

An example, illustrating the work of the search procedure is shown on Fig. 2. The values of difference indices and minimal difference indices are as follows:
$\texttt{diffind}(J) = +\infty$, $\texttt{diffind}(H) = 4$, $\texttt{diffind}(I) = 5$,
$\texttt{diffind}(A) = 4$, $\texttt{diffind}(F) = 4$, $\texttt{diffind}(E) = 5$,
$\texttt{diffind}(G) = 5$, $\texttt{diffind}(D) = 4$, $\texttt{diffind}(C) = 2$,
$\texttt{diffind}(C') = 5$, $\texttt{diffind}(B) = 3$ and
$\texttt{mdi}(A) = 2$, $\texttt{mdi}(B) = 2$, $\texttt{mdi}(C) = 2, \texttt{mdi}(C') = 5$,
$\texttt{mdi}(D) = 4, \texttt{mdi}(E) = 4, \texttt{mdi}(F) = 4, \texttt{mdi}(G) = 5$
$\texttt{mdi}(H) = 4, \texttt{mdi}(J) = +\infty, \texttt{mdi}(I) = 5$

For convenience we assume that every string s ends with a special `end-of-string` character such that `end-of-string` $< x$ for any other character x. The

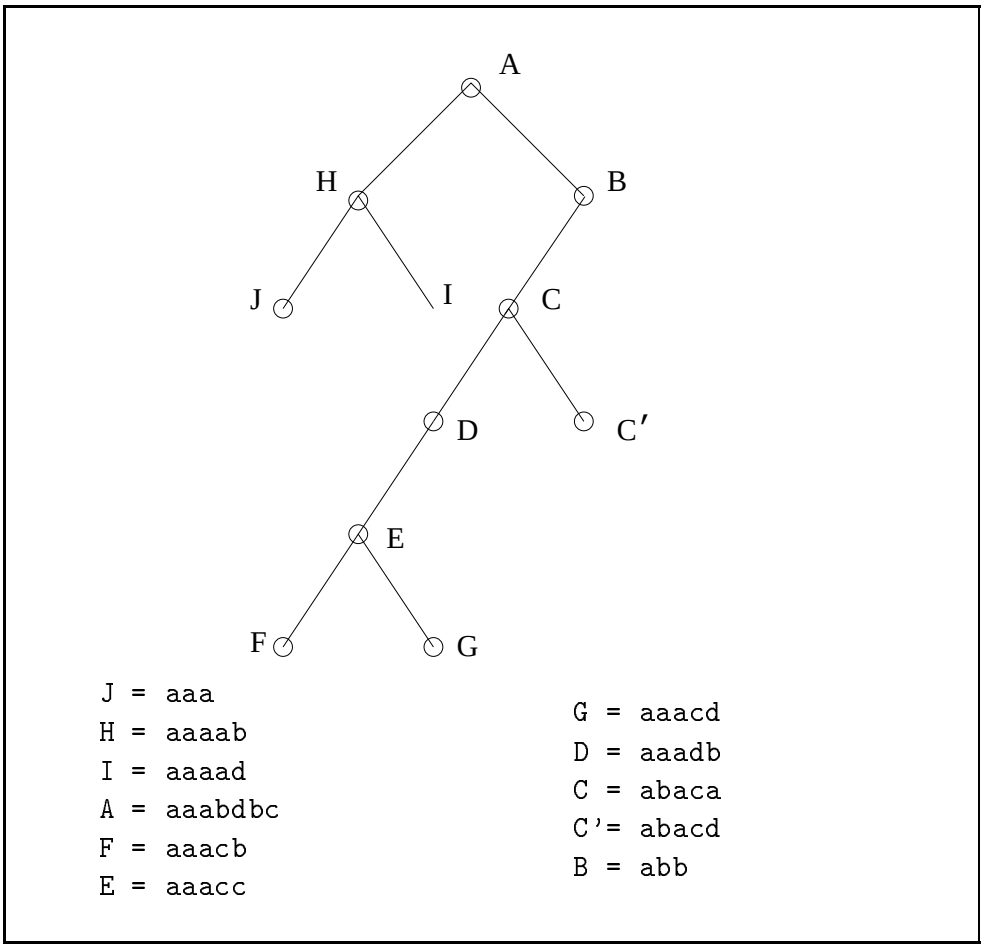

Figure 2: Example of the data structure

search procedure for a string $p = aaacb$ starts at $s = A$. Variable k will be incremented 3 times, since $lsd(A, p) = 4$. Then we visit nodes B and C, but we do not examine any characters in strings B and C because $\texttt{mdi}(C) = 2$ and $\texttt{diffind}(C) = 2$. Next we compare $D[4]$ with $p[4]$ and choose the left child of D (because $D[4] > p[4]$). Then, we increment k since $E[4] = p[4]$ and choose left child of E since $E[5] > p[5]$. Finally, we output F, because $F[5] = p[5]$ and $F[6] = p[6] =$ `end-of-string`. Thus the total number of comparisons is equal to 10.

Now suppose we are looking for the string $p' = aaa$. We examine the first three characters of node A and set k to 4 as in the previous example. We visit node H and then the left child of H, because $A[4] >$ `end-of-string` and $H[4] >$ `end-of-string`. Since $J[4] =$ `end-of-string` $= p[4]$, we output J.

Our search procedure can be easily modified to find the predecessor or the successor of a string p. For instance, to find a predecessor of p we maintain variable *cand* that points to a current candidate for a predecessor of p. Every time when a right node of the current node s becomes the next search candidate (lines `6` and `13` of the pseudocode on Fig. 2) *cand* is assigned to s.

In order to achieve the upper bound of $O(\log n + |p|)$ the height of the tree must be limited by $O(\log n)$, i.e. a balanced tree must be used. Many data structures, such as AVL trees [AVL62] and red-black trees (see, e.g. [GS78], [CLR94]), can be used as base data structures for differential trees.

After insertion the `diffind` values of the new inserted element and of its successor can be computed in $O(|p|)$. When a node y is deleted, the difference index of its successor z changes. The new difference index of the successor of y is simply the minimum of difference indices of z and y. Please observe that costs of deletion and insertion of a string p depend only on the length of p and the number of strings, and do not depend on the lengths of other strings.

Insertion and deletion operation in balanced trees are based on rotations. After a rotation operation values of `diffind` do not change, because the order of elements does not change and values of `mdi` nodes can be recomputed in a constant time. It is well known, that balanced trees, such as red-black and AVL trees can be re-balanced with at most a logarithmic number of rotations after insertion and deletion. From the above we formulate our result:

Theorem 1 *A differential tree structure for n strings requires space $O(n)$. A search for a string p in a balanced differential tree representing n strings of total length N takes time $O(\log n + |p|)$ with space $O(n)$. Insertion and deletion operation can be also performed in time $O(\log n + |p|)$.*

3 Implementation and Experimental Results

We have carried through several experiments with the differential trees We have used red-black trees as a base data structure in these experiments. In the first set of experiments we measured the time necessary for insert and search operations on a set of strings. We have used two files `dictcalls` and `dictwords` for this purpose. The file `dictwords` is a UNIX file `/usr/dict/words`. We compared the performance of our data structure with the standard C++ STL class `set`. We have chosen this method for comparisons, because its implementation is also based on red-black trees. Firstly, all the strings from a file were inserted into the data structure, then we searched

File	Differential Trees		STL implementation	
	Insertion in sec.	Search in sec.	Insertion in sec.	Search in sec.
dictcalls, dictionary	1,76	0,82	2,15	2,08
dictcalls, random	1,94	0,97	2,29	2,26
dictcalls, sorted	1,95	1,01	2,32	2,26
dictcalls, reverse,	1,96	1,00	2,30	2,24
dictwords, dictionary	0,27	0,08	0,35	0,21
dictwords, random	0,34	0,14	0,27	0,25
dictwords, sorted	0,36	0,15	0,29	0,25
dictwords, reverse	0,37	0,15	0,29	0,25

Table 1: Insertion and Search Times for `dictwords` and `dictcalls`

for each string from the file. The strings were inserted in the dictionary order, lexicographically sorted order, reverse sorted order and a random order. The dictionary order is different from the sorted order because, for instance, capital letters precede small letters. We performed the experiments on a SUN SPARC Workstation with 502 MHz Processor and 512 MB of main memory. The results in Table 1 showed clearly the advantages of our approach, despite of the highly tuned implementation of red-black trees used in the STL library. Our program shows faster insertion times in more than half of all experiments. However the main advantage are superior search times: a more than two-fold speed-up was achieved for all files.

For the second set of experiments we have artificially generated a set of strings. The strings are just hexadecimal representations of integer values. After each step integers were incremented by a step value of 256. Average string length in our set was 18.9. Our algorithms again showed better insertion and search times, as can be seen in Table 2

The table 3 shows the number of nodes and the average number of comparisons for the `dictwords` and `dictcalls` files. The average number of character comparisons equals to the average number of characters in a word,

Method	Differential Trees		STL implementation	
	Insertion in sec.	Search in sec.	Insertion in sec.	Search in sec.
random1	1,57	0,81	1,94	1,82
random2	1,70	0,93	2,02	1,90
sorted	1,72	0,93	2,01	1,90
reverse	1,71	0,93	2,01	1,92

Table 2: Insertion and Search Times for the generated set of strings

because each string character is compared to the equal string element only once. The last column, index comparisons, shows the number of comparisons between `mdi` and `diffind` fields and integers.

From the above experiments we can see that the presented data structure is highly competitive, compared to the best known search algorithms. Differential trees can also be used to find a predecessor or a successor of a string, or to report all strings from the specified range, stored in the data structure.

File	Nodes	Character Comparisons	Index Comparisons
dictcalls, dictionary	100188	22,53	14,72
dictcalls, random	100188	22,53	14,67
dictcalls, sorted	100188	22,53	14,68
dictcalls, reverse,	100188	22,53	14,68
dictwords, dictionary	25481	7,23	13,25
dictwords, random	25481	7,23	12,98
dictwords, sorted	25481	7,23	12,97
dictwords, reverse	25481	7,23	12,97

Table 3: Number of nodes and comparisons for `dictwords` and `dictcalls`

4 Conclusions and Future Work

Several modifications of the presented data structure are possible. For instance, instead of storing the difference index and the minimum difference index of a node N , we could store the `mdi` of the right child of N and $min(\texttt{mdi}(\texttt{lchild}(N)), \texttt{diffind}(N))$. This modification would probably lead to the better search times and slightly slower insertions.

It would also be interesting to shed some light on a lower space bound for a data structure that allows searches in time $O(\log n + |p|)$.

□

References

[AVL62] G. M. Adel'son-Velskij, Y.M. Landis "An Algorithm for the Organization of Information", Soviet Mathematics Doklady, vol. 3, pp. 1259-1263 (1962)

[AHHP00] A. Andersson, T. Hagerup, J. Håstad, O. Petersson, " Tight Bounds for Searching a Sorted Array of Strings", SIAM J. on Computing, 30(5), pp. 1552-1578 (2000)

[BS79] J. L. Bentley, J. B. Saxe, "Algorithms on Vector Sets" ,SIGACT News, 11, pp. 36-39 (1979)

[BS96] J. L. Bentley, R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings", Proc. SODA'96, pp. 360-369

[B59] R. de la Briandais, "File Searching Using Variable Length Keys", Proc. Western Joint Computer Conf. (1959)

[CLR94] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algortithms", MIT Press, 1994

[FG99] P. Ferragina, R. Grossi," The String B-tree: A New Data Structure for String Search in External Memory and Its Applications". J. of the ACM, 46(2), pp. 236-280 (1999)

[F60] E. Fredkin " Trie Memory", Communications of the ACM, pp. 490-499 (1960)

[GS78] L. J. Guibas, R. Sedgewick "A Dichromatic Framework for Balanced Trees", Proc. FOCS'78, pp.8-21.

[H80] D. S. Hirschberg, "On the Complexity of Searching a Set of Vectors", SIAM J. on Computing, 9, pp. 126-129, 1980

[MM93] U. Manber, G. Myers, "Suffix Arrays: A New Method fore Online String Searches", SIAM J. on Computing, 22, pp. 935-948 (1993)

[M79] K. Mehlhorn, "Dynamic Binary Search" SIAM. J. on Computing, 8, pp. 175-198 (1979)

[M68] D. R. Morrison, "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric", J. of the ACM, 15(4), pp. 514-534 (1968)

[ST85] D. D. Sleator, R. E. Tarjan, "Self-adjusting Binary Search Trees", J. of the ACM, 32(3), pp. 652-686 (1985)

[V84] V. K. Vaishnavi, "Multidimensional Height-Balanced Trees", IEEE Transactions on Computers, 33(4), pp. 334-343 (1984)