Suffix Cactus: A Cross between Suffix Tree and Suffix Array*

Juha Kärkkäinen

Department of Computer Science, P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
E-mail: Juha.Karkkainen@cs.Helsinki.FI
URL: http://www.cs.helsinki.fi/~tpkarkka

Abstract. The suffix cactus is a new alternative to the suffix tree and the suffix array as an index of large static texts. Its size and its performance in searches lies between those of the suffix tree and the suffix array. Structurally, the suffix cactus can be seen either as a compact variation of the suffix tree or as an augmented suffix array.

1 Introduction

The *suffix tree* is one of the most important data structures in stringology. The suffix tree is an index-like structure formed from a string that allows many kinds of fast queries about the string. What makes the suffix tree attractive is that its size and its construction time are linear in the length of the text [19, 14, 17]. Suffix trees have a wide variety of applications. Apostolico [4] cites over forty references on suffix trees, and Manber and Myers [13] mention several newer ones.

The application, that we are mostly interested in in this paper, is the use of a suffix tree as an index of a large static text to allow fast searches. The basic search type is string matching, i.e. searching for the occurrences of a pattern string in the text. Other useful forms of queries include regular expression matching and approximate string matching. Examples of very large texts requiring fast searching are electronic dictionaries [8], and biological sequence databases [16].

To work efficiently, the whole suffix tree must fit in the main memory. Thus the space requirement of the suffix tree is an important issue. Gonnet, Baeza-Yates and Snider [8] have studied the use of suffix trees with only a small part at a time in the main memory, but many applications slow down unacceptably. The exact size of the suffix tree depends on the implementation and the type of the text. A typical size for a tight implementation on english text is about 15 bytes per text symbol.

The *suffix array* [13, 8] is a data structure which, like the suffix tree, allows fast searches on a text. The size of an efficient implementation of a suffix array, including the text itself, is only 6 bytes per text symbol. In string matching the

* Work supported by the Academy of Finland.

performance of suffix arrays is comparable to suffix trees, but other types of searches, such as regular expression matching, are slower on suffix arrays.

In this paper we present a new suffix-tree-like data structure called the *suffix cactus*. The size of a suffix cactus, 10 bytes per text symbol, lies between the sizes of suffix trees and suffix arrays. The same holds for the performance in many applications, such as regular expression matching.

The suffix cactus offers an interesting new point of view to the family of suffix structures. The structure of the suffix cactus has similarities with both the suffix tree and the suffix array. The suffix cactus could be described either as a compact version of the suffix tree or as a suffix array augmented with some extra information. The suffix cactus can therefore be called a cross between the suffix tree and the suffix array.

Recently, Anderson and Nilsson [2, 3], and Irving [9] have introduced new alternative data structures. The level compressed trie of Andersson and Nilsson takes about 12 bytes per text symbol and has matching properties comparable to the suffix cactus. The suffix binary search tree of Irving takes 14 bytes per text symbol and is similar to the suffix array in matching problems.

1.1 Basic Definitions

Let $T = t_1t_2 \dots t_n$ be a string over alphabet Σ. A *substring* of T is a string $T_i^j = t_it_{i+1} \dots t_j$ for some $1 \le i \le j \le n$. The string $T_i = T_i^n = t_i \dots t_n$ is a *suffix* of string T and the string $T^j = T_1^j = t_1 \dots t_j$ is a *prefix* of string T. Let S and T be two strings and let j be the largest number for which $S^j = T^j$. Then the string $S^j = T^j$ is called the *longest common prefix* of S and T and its length j is denoted $\textsc{lcp}(S, T)$.

A *trie* (see e.g. Knuth [11]) is a rooted tree with the following properties.

1. Each node, except the root, *contains* a symbol of the alphabet.
2. No two children of the same node contain the same symbol.

A node v *represents* the string which is formed by catenating the symbols contained by the nodes on the path from the root to v, inclusive. Due to the second property, no two nodes may represent the same string. Note that, if a node v represents string S, then the ancestors of v represent the prefixes of S. The *depth* of a node v, denoted by $\textsc{depth}(v)$, is the length of the path from the root to v, i.e., the length of the string that v represents.

The *suffix trie* $STr(T)$ of text T is a trie whose leaves represent the suffixes of T. The nodes of suffix trie $STr(T)$ represent exactly the set of substrings of T, because every substring of the text is a prefix of some suffix, i.e. $T_i^j = (T_i)^j$. An example suffix trie, for the string `cabacca$`, is shown in Fig. 1.

The size of the suffix trie for a text of length n is $O(n^2)$ which makes it impractical for large texts. However, the suffix tree and the suffix cactus are basicly more compact (linear size) versions of the suffix trie. In Section 2 we will define the suffix cactus using the above description of the suffix trie.

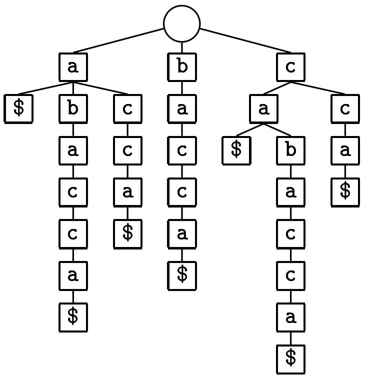

Fig. 1. The suffix trie of the string `cabacca$`. The symbol `$` is an extra symbol used for making all suffixes end in a leaf. The suffix `$` is omitted from the trie.

1.2 Matching

Let a string T of length n be the *text* and a string P of length m the *pattern*. The problem of *string matching* is to find the occurrences of string P as a subtring of T. It can be solved in linear time by scanning text T using, e.g., the Knuth-Morris-Pratt algorithm [12]. For a large static text, a faster solution can be achieved by preprocessing the text. Suffix trees, suffix arrays and suffix cactuses are suitable preprocessing structures.

In *regular expression matching* the goal is to find all substrings of text T that match a given regular expression. A similar problem is *approximate string matching* where, given a string P and an integer k, one wants to find the substrings T_i of text T such that the edit distance between P and T_i is at most k. Both of these problems can be solved by scanning the text. Regular expression matching takes $O(n)$ time (excluding the preprocessing of the regular expression) [1] and approximate string matching $O(kn)$ time [7, 18].

Baeza-Yates and Gonnet have described methods to use the suffix tree to do both regular expression matching [5] and approximate string matching [6]. The latter idea was also independently mentioned in [10, Remark 2]. Both of these methods are based on scanning one suffix of T at a time to find whether it has a matching prefix. The methods take advantage of the fact that, if a set of suffixes has a common prefix of length d, then the state of the scan after the first d characters is the same for all of the suffixes. Therefore that part of the scan needs to be done only once. The suffix tree provides the information about common prefixes. It can be replaced by another suffix structure.

The above method for approximate string matching is more efficient than the basic text scan method only with short patterns and small values of k. However, Myers [15] has developed a method to do efficient approximate string matching even with long patterns and large k. The method divides the pattern into smaller parts whose approximate occurrences with small edit distance limit are searched separately. The results are then combined and used to restrict the area of the text that needs to be scanned. The matching of the parts can be done with the method of Baeza-Yates and Gonnet; Myers uses a slightly different method.

1.3 Suffix Tree and Suffix Array

The *suffix tree* discovered by Weiner [19] is a compact version of the suffix trie. It is formed by catenating each unary node (a node with exactly one child) with its child. An example is shown in Fig. 2(a). The strings in the catenations are substrings of the text and can thus be represented by two pointers into the text. The suffix tree has one leaf for each suffix and the number of other nodes is less than the number of leafs, because all the other nodes have at least two children. Thus the size of the suffix tree is linear in the length of the text.

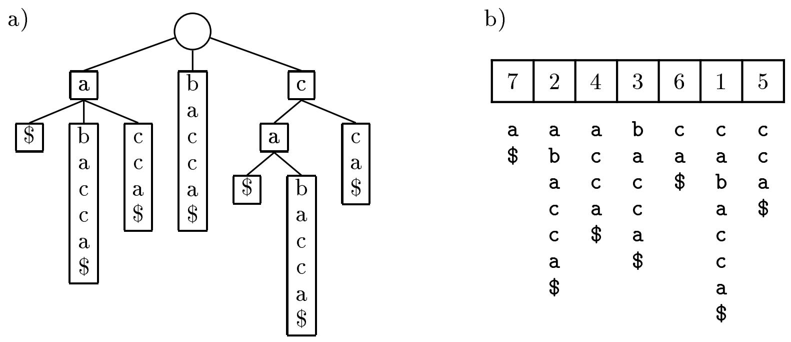

Fig. 2. a) Suffix tree and b) suffix array for string `cabacca$`.

If the alphabet size $|\Sigma|$ is considered constant, the suffix tree can be constructed in time $O(n)$ [19, 14, 17] and string matching takes time $O(m)$. The dependency on $|\Sigma|$ may be linear, logarithmic or constant depending on the implementation of branching. The most compact alternative uses linked lists and has linear dependency on $|\Sigma|$. In regular expression matching and approximate string matching the linked list implementation is as good as or better than other implementations.

In its basic form, the *suffix array* is just a lexicographically ordered array of the suffixes of the text. The suffixes are represented by their starting positions as illustraded in Fig. 2(b). The suffix array was discovered by Manber and Myers [13], and independently by Gonnet, Baeza-Yates and Snider [8].

String matching in suffix arrays can be done in $O(m \log n)$ time by a binary search. Manber and Myers [13] improved the string matching time to $O(m+\log n)$ by providing additional information about the lengths of the longest common prefixes (LCPs) between the suffixes. The LCPs are provided for each parent-child pair in an implicit tree structure called the interval tree. The interval tree is defined by the binary search order. The root of the interval tree is the middle suffix of the array, i.e. the first suffix processed in the binary search. The left child of the root is the middle suffix of the first half of the array and the right child is the middle suffix of the second half of the array. The next level of nodes is formed by the middle suffixes of the quarters of the array, and so on.

The above described LCP information is essential for efficient regular expression matching and approximate string matching in suffix arrays. The suffix array is still slower than the suffix tree in these tasks, in the worst case by a factor $O(\log n)$. In practice the difference is smaller, though.

The advantage of the suffix array over the suffix tree is its smaller size. Even with the LCP information the suffix array can be implemented using only 6 bytes per text symbol including the text itself.

The suffix array can be constructed in linear time by constructing first the suffix tree and then listing the suffixes in lexicographic order from the tree. Manber and Myers [13] have also described a construction algorithm that works by sorting the suffixes. It takes $O(n \log n)$ time in the worst case and $O(n \log\log n)$ time on average for random texts with even and independent distribution of characters. The advantage of this construction over the construction via the suffix tree is its smaller space requirement, 10 bytes per text symbol.

2 Suffix Cactus

The new data structure, *suffix cactus*, can, like the suffix tree, be viewed as a compact suffix trie. The suffix tree was formed by catenating the unary nodes with their children. To get a suffix cactus, every internal node is catenated with one of its children. The catenations are called the *branches* of suffix cactus.

Definition 1. Let v be a node of suffix trie $STr(T)$ of text T such that either v is the root or v is not the first child of its parent w. Then suffix cactus $SC(T)$ of T has a branch s that contains exactly the nodes on the path from v to the first leaf u under v.

Clearly, each node of $STr(T)$ is contained by exactly one branch of $SC(T)$. The branch containing the root of $STr(T)$ is called the *root branch*. The node v is called the root of branch s, u is called the leaf of s, and the parent w is called the parent node of s. The *branching depth* of s, denoted by $\mathrm{DEPTH}(s)$, is the depth of the parent node w. The branching depth of the root branch is 0.

Branch s *contains* the string formed by catenating the characters in the nodes contained by s. Branch s *represents* the same string as the leaf u. The leafs of $STr(T)$ represent the suffixes of T and there is thus a one-to-one correspondence between the suffixes of T and the branches of $SC(T)$. The starting point of the suffix represented by branch s will be denoted by $\mathrm{SUFFIX}(s)$. The string contained by s is now $T_{\mathrm{SUFFIX}(s)+\mathrm{DEPTH}(s)}$.

The term 'first' in Definition 1 implies the existence of an ordering among the children of a node. Any ordering can be used, which allows many alternative forms for the cactus. Two variations for string `cabacca$` are shown in Fig. 3. The left-hand side variation uses alphabetical ordering and is the one used by the implementation described in this paper.

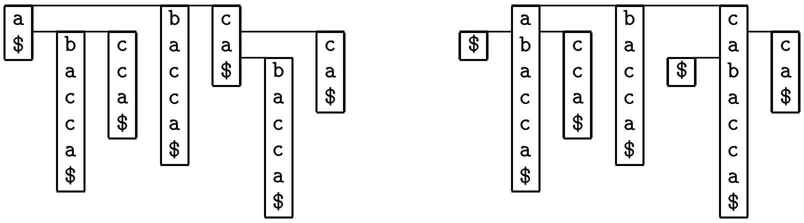

Fig. 3. Two variations of suffix cactus for the string `cabacca$`. Turn the figure upside down to see an explanation for the name 'cactus'.

The most obvious way to define the tree structure of a suffix cactus is the following.

Definition of parent (alternative 1). Let s be a branch of $SC(T)$ and let v be its root. The parent (branch) of s is the branch containing the parent node of v.

However, for the implementation that is described in the next section, the following is a more natural definition.

Definition of parent (alternative 2). Let s be a branch of $SC(T)$ and let v be its root. The parent (branch) of s is the branch containing the preceding sibling of v. The preceding sibling is defined by the same ordering as the one used in Definition 1.

With both of the alternative definitions all branches, except the root branch, have a parent.

As an example, let us consider the third branch from left in the cactus on the left in Fig. 3. By the first definition its parent is the first branch, but by the second definition the parent is the second branch.

3 An Implementation

The name 'cactus' comes from the way the branches start in the middle of other branches. Whichever of the alternative definitions of the tree structure is used, this kind of branching needs to be implemented differently from the traditional tree branching. The implementation affects the exact space requirement of the suffix cactus and the time complexity of the different matching problems. In this paper we describe in detail an implementation that is space efficient and has, in all of the above described matching problems, the same time complexity as the linked list implementation of the suffix tree.

This implementation is based on alphabetical ordering of the children of a node and the second alternative definition of the parent branch. The children of each branch are in a linked list from the highest branching one to the lowest branching one. A key property of the second alternative definition is that a branch can have at most one child at each branching depth. Therefore, following a child list to find a specific child takes no more time than following the string contained by the branch to the point of branching of that child. The child list structure can be formalized by the operations FIRSTCHILD and NEXTSIBLING in the obvious way. Their implementation is described a little later.

The SUFFIX and DEPTH values are kept in two tables. The tables are in the lexicographic order of the suffixes. The SUFFIX table is, in fact, the basic suffix array. To simplify notation, we use the rank of a branch in the above order as the name of the branch. That is, the suffix $T_{\text{SUFFIX}(s)}$ represented by branch s is the sth suffix of T in the lexicographic order. Branch 1 is the root branch.

The following three lemmas show how the branching structure of the suffix cactus of text T can be derived straight from the text.

Lemma 2. *The branching depth* $\text{DEPTH}(s)$ *of a branch* $s > 1$ *is* $\text{LCP}(T_{\text{SUFFIX}(s-1)}, T_{\text{SUFFIX}(s)})$.

Proof. Let v be the root, u the leaf, and w the parent node of branch s. Let v' be the alphabetically preceding sibling of v and let u' be the leaf of branch $s-1$. Then v' must be an ancestor of u'. The paths from root to u and u' go together until node w where they get separated. Thus $\text{LCP}(T_{\text{SUFFIX}(s-1)}, T_{\text{SUFFIX}(s)}) = \text{DEPTH}(w) = \text{DEPTH}(s)$. □

Lemma 3. *The parent branch of branch* $r > 1$ *is the latest branch* $s < r$ *such that* $\text{DEPTH}(s) \leq \text{DEPTH}(r)$.

Proof. Let v be the root and w the parent node of r. Let v' be the alphabetically preceding sibling of v. If s is the parent of r, then s contains v'. The parent node of s is w or an ancestor of w. Therefore the depth of s is at most $\text{DEPTH}(w) = \text{DEPTH}(r)$. Suffix $T_{\text{SUFFIX}(s)}$ precedes $T_{\text{SUFFIX}(r)}$ lexicographically and thus $s < r$. It remains to show that s is the latest branch satisfying these conditions.

Let t be a branch such that $s < t < r$. Let u'' be the leaf of t. Node v' must be an ancestor of u''. Because v' is contained by s, the root of t must be below v' on the path from v' to u''. Thus it holds $\text{DEPTH}(t) \geq \text{DEPTH}(v') > \text{DEPTH}(w) = \text{DEPTH}(r)$. □

Lemma 4. *A branch* s *has child branches only if branch* $s+1$ *is a child of* s. *Let* s *be such a branch and let* $r_1, r_2, \ldots, r_k$ *be the children of* s *from the highest branching to the lowest branching. Then* $s+1 = r_k < \cdots < r_1$.

Proof. By Lemma 3 r is a child of s if and only if

1. $s < r$,
2. $\text{DEPTH}(s) \le \text{DEPTH}(r)$ and
3. there is no branch $t > s$ such that the first two conditions would hold if s was replaced with t.

For $r = s+1$ the first and last condition always hold. Therefore, if $s+1$ is not a child of s, then $\text{DEPTH}(s) > \text{DEPTH}(s+1)$. In such a case, if any node r satisfies the first two conditions, then $t = s+1$ violates the third condition. Thus s can have no children, if $s+1$ is not a child of s.

The second claim of the lemma is clearly true if $k = 1$. Otherwise, let r_i and r_{i+1}, $1 \le i < k$, be two of the children of s. Then it holds that $\text{DEPTH}(r_i) < \text{DEPTH}(r_{i+1})$. If now $r_i < r_{i+1}$, then $t = r_i$ would violate the third childhood condition of r_{i+1}. Therefore we must have $r_{i+1} < r_i$. □

The last lemma enables us to describe the implementation of the branching operations FIRSTCHILD and NEXTSIBLING. The implementation consists of a single table called SIBLING. Using the notations of Lemma 4 this table can be defined by

$$\text{SIBLING}(r_i) = \begin{cases} r_1, & \text{if } i = k \\ r_{i+1}, & \text{if } i < k \end{cases}$$

or alternatively by

$$\text{SIBLING}(s) = \begin{cases} \text{FIRSTCHILD}(s-1), & \text{if } s-1 \text{ has children} \\ \text{NEXTSIBLING}(s), & \text{if } s \text{ has a next sibling} \end{cases}$$

In other words, the children of each branch form a cyclical list. In addition we define $\text{SIBLING}(1) = 1$. The FIRSTCHILD and NEXTSIBLING can now be defined as follows.

$$\text{FIRSTCHILD}(s) = \begin{cases} \text{SIBLING}(s+1), & \text{if } \text{SIBLING}(s+1) \ge s+1 \\ \text{none}, & \text{if } \text{SIBLING}(s+1) < s+1 \end{cases}$$

$$\text{NEXTSIBLING}(s) = \begin{cases} \text{SIBLING}(s), & \text{if } \text{SIBLING}(s) < s \\ \text{none}, & \text{if } \text{SIBLING}(s) \ge s \end{cases}$$

Fig. 4 shows an example of this implementation.

s	1	2	3	4	5	6	7
SUFFIX(s)	7	2	4	3	6	1	5
DEPTH(s)	0	1	1	0	0	2	1
SIBLING(s)	1	4	3	2	5	7	6

Fig. 4. The implementation of the left-hand side suffix cactus in Fig. 3.

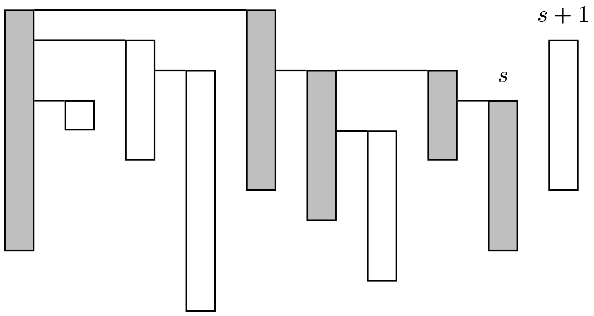

Fig. 5. The state of a suffix cactus before the processing of branch $s+1$. The active branches are grayed.

4 Construction

In this section we will describe two construction algorithms for the above implementation of the suffix cactus. The algorithms work in two phases, the second of which is common to both. The first phases of the algorithms construct the SUFFIX and DEPTH tables. One algorithm uses the suffix tree to do this and the other uses the suffix array. The common second phase then constructs the SIBLING table from the DEPTH table. We start by describing the second phase.

At the start of the second phase the DEPTH table tells the branching depths of each branch. By Lemma 3 the parent branch of branch r is the latest branch s preceding r such that $\mathrm{DEPTH}(s) \leq \mathrm{DEPTH}(r)$. Therefore the DEPTH table fully defines the branching structure of the cactus and the SIBLING table can be calculated from it.

The SIBLING table is constructed in one first branch to last branch pass. Let us look at the situation when a branch s has just been processed and the processing of branch $s+1$ is about to start (Fig. 5). Let $s_1, s_2, \ldots, s_k$ be the path from branch 1 (the root) to branch s with $s_1 = 1$ and $s_k = s$. The branches on the path are called the *active branches*. The first (highest branching) children of each active branch may still be among the unprocessed branches. The first children of the other processed branches and the next siblings of all processed branches have, on the other hand, all been processed. Therefore, we can assume that the SIBLING table is finished up to the entry s, excluding the entries $s_0+1, s_1+1, \ldots, s_{k-1}+1$.

The parent of branch $s+1$ must be one of the active branches. To be able to find the parent quickly, the active branches are on a list from the last to the first. The parent of $s+1$ is the first branch s_i on the list such that $\mathrm{DEPTH}(s_i) \leq \mathrm{DEPTH}(s+1)$. The list is implemented using the so far unfinished SIBLING table entries, i.e. $\mathrm{SIBLING}(s_i+1) = s_{i-1}$ for $i = 1, \ldots, k-1$.

Let us now see what happens when branch $s+1$ is processed. If the parent of $s+1$ is s, we make $s+1$ active by adding it to the beginning of the list of active branches and we are done. Assume then that active branch s_i, $i < k$, is the parent of $s+1$. Now we do the following.

1. Find s_i by following the list of active branches.
2. Remove the branches $s_{i+1}, \ldots, s_k$, that are passed during the search, from the list of active branches and finalize their first children by setting $\textsc{sibling}(s_j + 1) = s_{j+1}$ for $j = i+1, \ldots, k-1$.
3. Make s_{i+1} the next sibling of $s+1$ by setting $\textsc{sibling}(s+1) = s_{i+1}$.
4. Add $s+1$ to the beginning of the list of active branches.

When all branches have been processed, we travel the list of active branches once more to set the first children of the remaining active branches. The algorithm is presented in detail in Fig. 6.

```
SIBLING(1) = 1
s_{k-1} = 0
for s = 1 to n - 1 do
    if DEPTH(s) ≤ DEPTH(s + 1) then          % Is s parent of s + 1?
        SIBLING(s + 1) = s_{k-1}
        s_{k-1} = s
    else
        s_{i+1} = s
        s_i = s_{k-1}
        while DEPTH(s_i) > DEPTH(s + 1) do    % Travel the list of active branches
                                              % until the parent of s + 1 is found.
            r = SIBLING(s_i + 1)
            SIBLING(s_i + 1) = s_{i+1}        % Remove passed branches from the list
                                              % and finalize their first children.
            s_{i+1} = s_i
            s_i = r
        end
        SIBLING(s + 1) = s_{i+1}
        s_{k-1} = s_i
    end
end
s_{i+1} = n
s_i = s_{k-1}
while s_i > 0 do                              % Finalize the first children
    r = SIBLING(s_i + 1)                      % of the last active branches.
    SIBLING(s_i + 1) = s_{i+1}
    s_{i+1} = s_i
    s_i = r
end
```

Fig. 6. The construction of SIBLING table from the DEPTH table. The variables s_{k-1}, s_i and s_{i+1} are so named to help the comparison between the algorithm and the description in the text.

Excluding the **while** loops, the algorithm clearly works in linear time. Each round of the **while** loops walks one step in the list of active branches and removes one branch from the list. Once removed, a branch cannot return to the list. Thus, at most one round of the **while** loops is executed for each branch. This gives us the following theorem.

Theorem 5. *The* SIBLING *table can be constructed from the* DEPTH *table in linear time and constant additional space.*

The remaining problem with the construction of the suffix cactus is to get the SUFFIX and DEPTH tables somehow. One way is to use the suffix tree. A lexicographically ordered depth-first traversal of the tree can be used to recover the necessary information from the tree in linear time. As mentioned in Section 1.3, the suffix tree itself can be build in linear time, so the whole construction works in linear time. The construction takes at least as much space as the suffix tree construction and may take a little more depending on the details of implementation.

The SUFFIX and DEPTH tables can also be constructed from the suffix array with LCP information. The basic suffix array forms the SUFFIX table as such. As mentioned in Lemma 2, the values in DEPTH table are LCPs of lexicographically adjacent suffixes. These values can be recovered from the LCP information of the suffix array by a traversal of the interval tree in linear time. If the suffix array is build using the $O(n \log n)$ sorting method, it dominates the time complexity of the whole cactus construction. The advantage of this construction is that all stages work in the space of the final suffix cactus.

5 Experimentation

To see how the suffix cactus behaves in practice, we implemented the described variation of the suffix cactus together with the linked list version of the suffix tree and the version of the suffix array with LCP information. The tests were run on a 90 MHz Pentium PC with 16 Mbytes of memory running Linux operating system.

We implemented the standard suffix tree construction [14, 17], the suffix array construction by sorting [13], and both of the suffix cactus construction algorithms described in the previous section. Table 1 gives the execution times and the space requirements of these construction algorithms. The space requirements include the text.

The space requirement of a finished structure is 6 bytes per text symbol for the suffix array and 10 bytes per text symbol for the suffix cactus, regardless of the construction method. In principle, the space requirement of a finished suffix tree could be reduced a little from the construction time space requirement by releasing the suffix links. In our implementation this is not done because of the complications in memory management caused by not knowing the number of nodes in the suffix tree in advance.

In the implementations most numbers and pointers take 4 bytes. The exceptions are the LCPs of the suffix array and the DEPTHs of the suffix cactus, both of

Table 1. Space requirements and execution times of the construction.

text			space (bytes/n)				time (s)			
type	$\|\Sigma\|$	n	tree	array	cactus via tree	cactus via array	tree	array	cactus via tree	cactus via array
english	71	3000	13.48	10	14.48	10	0.08	0.21	0.09	0.23
english	74	30000	14.77	10	14.77	10	0.67	2.85	0.84	2.99
english	77	300000	15.17	10	16.17	10	6.60	36.4	8.63	37.7
random	77	300000	9.72	10	10.72	10	21.2	27.0	22.7	28.4
DNA	4	300000	17.70	10	18.70	10	5.62	41.4	7.78	42.6
random	4	300000	17.43	10	18.43	10	5.66	33.8	7.84	35.1
random	16	300000	11.80	10	12.80	10	8.10	31.2	9.91	32.5
random	64	300000	10.95	10	11.95	10	19.4	26.8	21.0	28.1

Table 2. String matching and regular expression matching times. The string matching times are total times of matching 10000 patterns.

text			string matching					regular expression matching			
				matches	time (s)				time (ms)		
type	$\|\Sigma\|$	n	m	/pattern	tree	array	cactus	matches	tree	array	cactus
english	71	3000	8	3.87	0.82	0.38	0.79	1	1.13	2.50	1.43
english	74	30000	8	1.67	0.97	0.46	1.13	2	5.20	9.53	5.86
english	77	300000	8	4.86	1.63	0.67	1.86	33	19.5	33.1	20.8
random	77	300000	8	1.00	1.35	0.62	2.19	0	9.61	19.2	12.0
DNA	4	300000	8	8.17	0.96	0.71	0.61	19206	201	123	91.9
random	4	300000	8	5.58	0.79	0.69	0.58	18708	195	119	88.8
random	4	300000	12	1.02	0.57	0.64	0.58	"	"	"	"
random	16	300000	6	1.02	0.66	0.63	0.90	4670	740	800	730
random	64	300000	4	1.02	1.26	0.62	1.94	4	13.6	24.2	16.3

which take only one byte. The rare case that a longest common prefix between two suffixes is more than 255 is recognized and handled separately when necessary. This might affect the pattern matching time, but only when the length of the pattern exceeds 255.

To test matching performance we implemented string matching and regular expression matching algorithms for all three data structures. The results of our tests are given in Table 2. The execution times include going through the set of matches.

The string matching tests used 10000 patterns selected randomly from the text. The regular expression $\mathtt{a}S^*\mathtt{c}S^*\mathtt{c}$, where $S = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\} \setminus \{\mathtt{d}, \mathtt{t}\}$, was used in the regular expression tests. All the test texts contain letters `a`, `c`, and at least one of `d` and `t`. The matching times do not include the conversion of the regular expression into an automaton.

6 Concluding Remarks

We have described one variation of the suffix cactus in this paper. There are other interesting variations, notably one which implements the branching using hashing and another that uses a kind of binary tree structure. The main advantage of these variations would be better performance in string matching for large alphabets. Due to the nature of the suffix cactus these other variations need implementation stuctures and construction algorithms that are totally different from the ones described in this paper. There remains work to be done in developing these versions.

Acknowledgements

I would like to thank Esko Ukkonen who suggested the name 'cactus'.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, chapter 9, pages 318–361. Addison-Wesley, 1974.
2. A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Inf. Process. Lett.*, 46(6):295–300, July 1993.
3. A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software—Practice and Experience*, 25(2):129–141, Feb. 1995.
4. A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 85–95. Springer-Verlag, 1985.
5. R. A. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. In *Proc. 16th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 46–62, 1989.
6. R. A. Baeza-Yates and G. H. Gonnet. All-against-all sequence matching. Technical report, Department of Computer Science, University of Chile, 1990.
7. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. Comput.*, 19(6):989–999, Dec. 1990.
8. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.
9. R. W. Irving. Suffix binary search trees. Technical report TR-1995-7, Computing Science Department, University of Glasgow, Apr. 1995.
10. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 240–248, Sept. 1991.
11. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, chapter 6.3, pages 481–505. Addison-Wesley, 1973.
12. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, June 1977.

13. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
14. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.
15. E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct./Nov. 1994.
16. *Nucleic Acids Research*, 20(Sequences Supplement):2009–2210, May 1992.
17. E. Ukkonen. Constructing suffix trees on-line in linear time. In J. van Leeuwen, editor, *Algorithms, Software, Architecture. Information Processing 92*, volume 1, pages 484–492, 1992. Full version is to appear in *Algorithmica*.
18. E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, Nov. 1993.
19. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.