Efficient Implementation of Suffix Trees*

Arne Andersson   Stefan Nilsson
Department of Computer Science, Lund University,
Box 118, S-221 00 Lund, Sweden

Abstract

We study the problem of string searching using the traditional approach of
storing all unique substrings of the text in a suffix tree. The methods of path
compression, level compression, and data compression are combined to build
a simple, compact, and efficient implementation of a suffix tree. Based on
a comparative discussion and extensive experiments, we argue that our new
data structure is superior to previous methods in many practical situations.

Keywords: LC-trie, path compression, level compression, data compression,
suffix tree, suffix array.

Introduction

Locating substrings in text documents is a commonly occurring task. For large
documents, such as books, dictionaries, encyclopedias, and DNA sequences, the
choice of data structure and algorithms will have noticeable effect on computer
capacity requirements and response times. If the document is small or if speed
is of no concern, we can answer a query by scanning the entire text. However, if
the text is long and we expect to perform many searches it will be worthwhile to
perform a preprocessing step where we build a data structure to speed up searches.
A comprehensive survey of string searching in preprocessed text can be found in
Section 7.2 of "Handbook of Algorithms and Data Structures" [1].

A common method to accelerate string searching is to store references to all
sistrings of the text in a *trie*. A sistring (semi-infinite string) is a substring of the
text, defined by its starting position and continuing to the right as far as necessary
to make the string unique. The resulting trie is known as a *suffix tree*. This data
structure has a wide range of applications, including approximate string matching,
compression schemes, and genetic sequences. The papers by Apostolico [2] and
Manber and Myers [3] contain extensive lists of references. In this article we discuss
how the suffix tree can be used for string searching.

We examine an unconventional representation of a trie, which uses adaptive
branching; the number of descendants of a node depends on how the elements are
distributed. This data structure, the *level-compressed trie* or *LC-trie* [4], constitutes
an efficient representation of a binary trie. We show that this new approach, when
combined with other standard compression methods, gives a fast and compact data
structure that is easy to implement.

The Level-Compressed Patricia Tree

In this section we discuss a new implementation of a *level-compressed Patricia tree*, which can be used to accelerate string searching in large texts. In essence, this structure is a compact version of a *trie* [1], a tree structure that uses the characters of the key to guide the branching. In the standard implementation the internal nodes of the trie contain a set of pointers to nodes. A search is performed in the following way: The root uses the first character of the search string to select the subtree in which to continue the search, the direct descendants of the root use the second character, and so on. An example of a trie is shown in Figure 1(a). For a detailed explanation of the figure we refer the reader to the example at the end of this section. We will focus our attention on binary tries, since they are particularly easy to implement. An example of a binary trie is given in Figure 1(b).

Path Compression

A well known method to decrease the search cost in binary tries is to use *path compression*. At each internal node an index is used to indicate the character used for branching at this node. With this additional information available at the nodes, we can remove all internal nodes with an empty subtree. The path-compressed binary trie is often called a *Patricia tree* [1]. The Patricia tree corresponding to the binary trie in Figure 1(b) is shown in Figure 1(c). We also observe that the size of the Patricia tree does not depend on the length of the strings, but only on the total number of strings. A Patricia tree storing n strings has exactly $2n - 1$ nodes.

A Patricia tree can be represented very space efficiently by storing the nodes in an array. Each node is represented by two numbers, one that indicates the number of bits that can be skipped and one that is a pointer to the left child of the node. A pointer to the right child is not needed if the siblings are stored at consecutive positions in the array. Each leaf represents a unique sistring and may contain a pointer to some other data structure.

Level Compression

Level compression [4] can be used to reduce the size of the Patricia tree. To make the description easier to follow we start by considering a plain binary trie (without Patricia compression). If the i highest levels of the trie are complete, but level $(i + 1)$ is not, we replace the i highest levels by a single node of degree 2^i. This replacement is repeated top-down and we get a structure that adapts nicely to the distribution of the input. We will refer to this data structure as a *level-compressed trie* or an *LC-trie*. For random independent data the expected average depth of an LC-trie is much smaller than that of a trie: $\Theta(\log^* n)$[1] for an LC-trie but $\Theta(\log n)$ for a trie.

It should be clear that path compression and level compression can be combined. Each internal node of degree two that has an empty subtree is removed, and at each internal node we use an index that indicates the number of bits that have been skipped. The level-compressed version of the Patricia tree in Figure 1(c) is shown in Figure 1(d).

The LC-trie is very easy to implement. In fact, we can use essentially the same implementation as above. We store the nodes in an array and siblings are stored in consecutive positions. Each node is represented by three numbers, two of which indicate the number of bits to be skipped and the position of the leftmost child. The third number indicates the number of children. This number will always be a

---

[1]The function $\log^* n$ is the iterated logarithm function: $\log^* 1 = 1$; for $n > 1$, $\log^* n = 1 + \log^*(\lceil \log n \rceil)$.
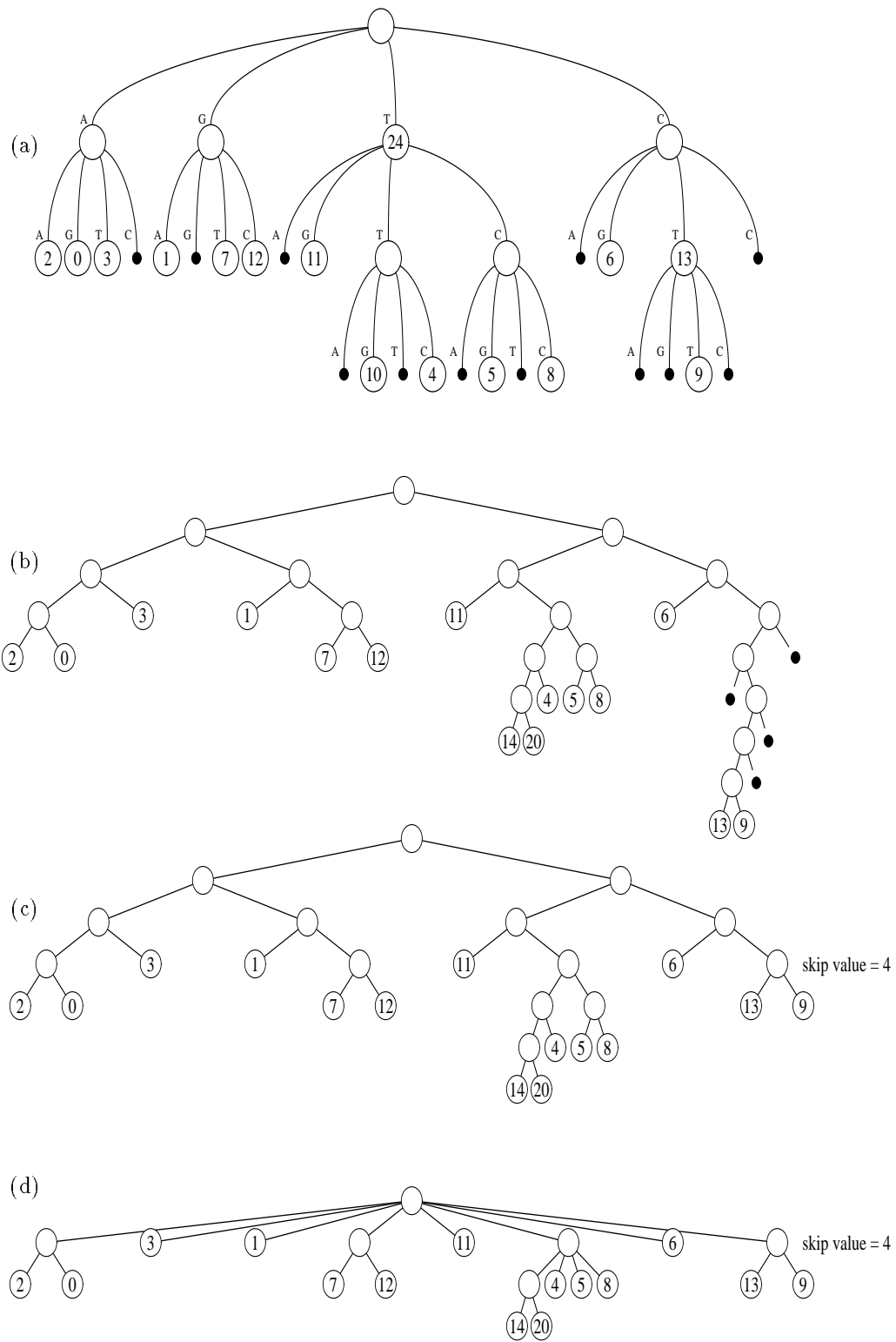
Figure 1: (a) Trie   (b) Binary Trie   (c) Patricia tree   (d) LC-trie.

	branch	skip	pointer
0	3	0	1
1	1	0	9
2	0	0	3
3	0	0	1
4	1	0	11
5	0	0	11
6	2	0	13
7	0	0	6
8	1	4	19
9	0	0	2
10	0	0	0
11	0	0	7
12	0	0	12
13	1	0	17
14	0	0	4
15	0	0	5
16	0	0	8
17	0	0	14
18	0	0	10
19	0	0	13
20	0	0	9

Table 1: Array representation of the LC-trie in Figure 1d.

power of 2 and hence it can be represented using a small number of bits. In fact, if the tree has n leaves, $\lceil \log \log(n) \rceil$ bits will suffice.

Example: DNA sequences are strings made up from the four nucleotide bases A, G, T, and C (A = adenine, G = guanine, C = cytosine, T = thymine). In this example we consider the first 15 nucleotide bases of the Epstein-Barr virus:

`AGAATTCGTCTTGCT`

In Figure 1 we show how the sistrings of this string can be represented by different trie structures. The numbers inside the nodes are pointers into the string.

(a) A traditional trie, internal nodes have degree 4.

(b) The corresponding binary trie. We have coded A, G, T, and C, as 00, 01, 10, and 11, respectively. The string 100000... has been appended to the end of the sequence to make sure that all sistrings are unique.

(c) The corresponding Patricia tree. The long path has been replaced by a skip value.

(d) The corresponding LC-trie.

Table 1 shows the array representation of the LC-trie, each entry represents a node. The number in the "branch" column indicates the number of bits used for branching at each node. A value $k \geq 1$ indicates that the node has 2^k children. The value $k = 0$ indicates that the node is a leaf. The number in the "skip" column is the Patricia skip value, i.e. the number of bits that can be skipped during a search operation. The value in the "pointer" column has two different interpretations. For

4

an internal node, it is used as a pointer to the leftmost child; for a leaf it is used as
a pointer to the position of the string in the text.

As an example we search for the string `TCG`. The binary code for this string is
101101. We start at the root, node number 0. We see that "branch" is 3 and "skip"
is 0 and therefore we extract the first three bits from the search string. These 3 bits
have the value 5 which is added to "pointer", leading to position 6. At this node
"branch" is 2 and "skip" is 0 and therefore we extract the next two bits. They have
the value 2. Adding 2 to "pointer" we arrive at position 15. This node has "bits"
equal to 0 which implies that it is a leaf. The pointer value 5 gives the position of
the string in the text. Observe that it is necessary to check whether this constitutes
a true hit. For example, the search for `TCG` is exactly mirrored by a search for `TCA`;
the former string is present in the text, whereas the latter is not.   □

Further Compression

In our experiments we have used a simple strategy to reduce the size of the LC-
trie. Internal nodes are restricted to have either a positive Patricia skip value or a
branching factor larger than two. We replace the two fields "branch" and "skip"
by one integer, one bit of which is used to distinguish between the different types
of values. This will reduce the size of the nodes and the fact that we may end up
using somewhat fewer skip values turns out to have very little effect in practice.
Compared to the original LC-trie, the number of nodes will only increase by a small
fraction and hence we get an overall size reduction.

A New Data Structure for String Searching

In this section we discuss how the LC-trie can be used for string location. In
particular, we discuss how the binary encoding of the text affects the behavior of
the data structure and how texts that are to large to fit in main memory can be
managed.

Suffix Tree

A document of length n may be viewed as consisting of n sistrings, one sistring
starting at each position of the text. The traditional method is to store these
sistrings (or a suitable subset of them) in a path-compressed trie; the resulting
structure is called a *suffix tree* [1]. As will be shown by our experiments this data
structure can be substantially improved by level compression.

To keep the data structure simple it is common practice to make sure that no
sistring in the text is a proper prefix of another sistring. This can be achieved by
appending a unique string at the end of the text. A frequently used method is to
augment the alphabet with a new character solely used for this purpose. The new
character is appended to the end of the text. Another solution that is especially
attractive when handling binary strings, is to append the string 100000 . . . to the
end of the text. [2]

Huffman Coding

Often, a text is represented using ASCII-code, where each character is represented
by an 8 bit integer. However, it is unlikely that all 256 possible characters will
actually be used. In fact, a normal text often uses less than a hundred characters.
The superfluous bits in the ASCII-code will increase the search time in the binary

---

[2] In fact, any non-periodic string may be used.

trie, even when path compression is used. Therefore we suggest the use of *Huffman coding* [5], the easiest and most well known data compression method. Huffman coding has the advantage of generating more evenly distributed binary strings and hence we will get a well balanced LC-trie with high branching factors. Observe that it is possible to use the Huffman coded version of the characters within the trie even if the text is stored in a different format.

Construction

A suffix tree implemented as a level-compressed Patricia tree can be constructed in $O(n \log k)$ time and $O(n)$ space, where n is the size of the text and k is the size of the alphabet. A brute force method that achieves this starts by constructing an ordinary suffix tree, i.e. a path-compressed k-ary trie, containing all sistrings of the text. There are several algorithms that do this in $O(n \log k)$ time and $O(n)$ space [6, 7]. We also assume that the text ends with a special end marker and we assume that the binary encodings of the characters are of equal length. The LC-trie can now be constructed in two steps.

First, each node of the suffix tree is replaced by a binary Patricia tree and the skip values are updated. This takes $O(n)$ time since the total number of nodes is $O(n)$ both in the original and in the modified tree. (Observe that a skip value can be updated in constant time only if we assume that the binary encodings of the characters are of equal length.)

Then, the level compression is performed during a top-down traversal of this modified tree. The topmost i complete levels of the tree are replaced by a single node of degree 2^i and this replacement is performed recursively on each subtree. Once again, this step can be performed in $O(n)$ time, since the total number of nodes in the tree is $O(n)$.

In practice an even simpler method can often be used. First, construct an array of pointers to all sistrings of the text and sort this array using a standard sorting algorithm, such as the $n \log n$ time suffix sort algorithm by Manber and Myers [3] or a fast string sorting algorithm such as Forward Radix Sort [8]. Given this sorted array, it is easy to make a top-down construction of the LC-trie. The algorithm is quite easy to implement, but it has a quadratic worst case running time. However, we found this approach to be quite efficient for most of our input data. Only one text, a DNA sequence containing long repeated substrings, took noticeably more time to construct.

Secondary Memory

Many applications need to handle large texts that do not fit in main memory. To reduce the number of time-consuming accesses to secondary memory a partial data structure that fits in main memory could be employed. We suggest a partial LC-trie stored in main memory used as an index into a *suffix array* [3] (*PAT array* [1]) stored in secondary memory.

The suffix array consists of an array of pointers. Each pointer points to a sistring in the text and the pointers are sorted according to the sistrings that they point to. The partial LC-trie is constructed in the following way: As soon as we reach a node that covers less than k strings, where k is a constant, we stop the tree building algorithm and add a pointer into the suffix array. The number k is called the *cutoff value*.

Other Implementations

In this section we describe some of the most common data structures used to speed up string searching.

Trie Implementations

The implementation of a suffix tree most commonly referred to in the literature is a k-ary trie, where k is the size of the alphabet. Such a trie may be implemented in several ways. Since this data structure has been extensively studied [1] we just briefly mention the three most common implementations.

- Each internal node consists of an array of pointers to the descendants. The length of the array equals the size of the alphabet.

- The non-null outgoing pointers are stored in a linked list.

- The non-null outgoing pointers are stored in a (balanced) binary search tree.

However, all of these implementations give rise to a large space overhead, reducing the usefulness in practical applications.

Another possibility is to use a Patricia tree. In this case, we ignore the size of the alphabet and use a binary encoding of the strings. The space requirement for the Patricia tree will be much smaller than for the corresponding alphabetic trie. On the other hand, the binary strings are longer than their alphabetic counterparts and hence the search time will increase.

Array Implementations

Another very natural method, suggested by Manber and Myers [3], is to store references to all sistrings in a suffix array. The suffix array consists of an array of pointers. Each pointer points to a sistring in the text and the pointers are sorted according to the sistrings that they point to. String location is performed by making a binary search in the array. One possible drawback of this simple structure is that the worst case time for searching is high. Also, the text has to be accessed at each step of the search, which might be very costly if the text is stored on secondary memory.

In order to achieve a low worst case cost, Manber and Myers also introduced an augmented suffix array, where two numbers were added to each entry. These numbers indicate how many characters at the beginning of each string that can be skipped during comparisons. In this way, the worst time search cost was reduced. But also in this case the text has to be accessed at each step of the search.

Since the augmented suffix array has a large space overhead, another method that combines a suffix array with a *bucket array* was suggested for practical use. Strings from a k-ary alphabet are treated as k-ary numbers and the universe is split into a number of buckets. Each bucket contains a pointer into the suffix array. Thus, a string location is reduced to a binary search within a small part of the suffix array. Choosing $n/4$ buckets the total space requirement increases by only one fourth compared to the suffix array alone. Their comparison with traditional implementations of suffix trees demonstrated that the suffix array/bucket array combination required roughly the same search time as a suffix tree, while the space requirements were considerably smaller.

Secondary Memory

The combination of a bucket array and a suffix array is also well suited for very large texts. The bucket array may be stored in main memory and the suffix array in

secondary memory. In this way the number of accesses to secondary memory during the binary search will be small. However, this method does not adapt gracefully to data that deviates markedly from the uniform distribution. Some buckets may contain many elements while others are empty.

Experiment 1: Main Memory

In order to study the real world behavior of the various trie structures discussed above, we have carried out a number of experiments. In order to get reproducible results, we have chosen not to make experiments on a specific architecture using a specific programming language. Instead, we have performed simulations, measuring critical operations.

Method

To estimate space requirements, we let characters and branching values occupy 1 byte each, small integers, such as skip values, occupy 2 bytes each, and pointers occupy 4 bytes. In order to measure search costs, we have counted the average number of steps required for a successful search. As steps we count the number of traversed nodes in a trie (i.e. the average depth of a leaf), and the number of string comparisons made in a suffix array.

Experiments were performed on the following data:

Random text: A text consisting of zeroes and ones, independently chosen with equal probability.

DNA: The longest available complete DNA sequence, the Epstein-Barr virus.

FAQ, ASCII: An English text in ASCII format, the Frequently Asked Questions list for the newsgroup comp.windows.x (2 Sep 91).

FAQ, Huffman: The same text as above, but Huffman coded.

In each case, we have performed our experiments on 1%, 10%, and 100% of the input text.

The following data structures were examined:

Augmented suffix array: The worst-case efficient data structure presented by Manber and Myers. Each entry in the array contains one address (4 bytes) and two skip values (2 bytes each). Hence, an augmented suffix array of length n occupy $8n$ bytes.

Trie, linked list implementation: An alphabetic trie were at each internal node the outgoing pointers are stored in a linked list. Path compression is used. An internal node in the trie of degree d is represented by d list nodes, each one contains one character, one skip value, and two pointers, a total of 11 bytes per list node. External nodes are marked by a specially designated skip value and contain one pointer into the document, requiring 6 bytes per node. For n leaves and I internal nodes, the total space is $11I + 6n$.

Trie, array implementation: An alphabetic trie with an array of length k equal to the size of the alphabet at each internal node. Path compression is used. Each internal node uses $4k$ bytes for pointers and 2 bytes for its skip value. External nodes are represented as above. For n external nodes and I internal nodes, the total space is $(4k + 2)I + 6n$.

Patricia tree: A binary trie implemented as a Patricia tree. The trie is stored in an array and each node is represented by a skip value and an address, requiring 6 bytes. The total number of nodes in a Patricia trie is $2n - 1$ and hence the space required is $6(2n - 1)$.

LC-trie: We implement the trie in the space efficient way described above, where each node is represented by two integers, one short integer that either holds a skip or a branch value and one long integer that holds a pointer, a total of 6 bytes per node. An LC-trie with n leaves and I internal nodes requires $6(n + I)$ bytes.

Not all combinations of input texts and data structures make sense. In an alphabetic trie it makes no difference how the characters have been encoded. Also, the alphabetic trie and the binary trie coincide for a binary string.

We also performed experiments on a trie where each internal node was represented as a binary search tree. This structure achieved roughly the same average depth as a Patricia trie, but at the cost of using considerably more space. These results are not included here.

Discussion

The simulation results are presented in Table 2. Starting with the space requirements, we observe that among the various trie structures the LC-trie requires the smallest space for all kinds of input. The only data structure that uses less space than the LC-trie is the suffix array.

The DNA-sequence has a very smooth character distribution and hence the results for this input are very similar to the results for random data. In both cases, the average depth in an LC-trie is considerably smaller than in any of the other tries. As expected, it is also smaller than the number of binary search steps required in a suffix array; the average depth of an LC-trie is $O(\log^* n)$ for independent random data [4].

For English text the LC-trie has a markedly better performance than an ordinary trie, except for the array implementation, which achieves smaller average depth at the price of a prohibitively large space overhead. We also observe that ASCII-coding is not the best alternative when building an LC-trie; if we use Huffman coded text the search cost is significantly reduced. Using Huffman coding on a binary trie also gives a slight improvement. However, it is the combination of Huffman coding and level compression that makes the main difference. Huffman coding smoothes the text so that level compression becomes more effective.

One might argue that the branching operation made in an LC-trie is more expensive than in a Patricia trie, instead of using just one bit for branching at each node we use a varying number of bits. However, the following argument shows that the total search cost in an LC-trie is in fact smaller. First, we note that during a search in an LC-trie we examine exactly the same bits in the query string as during a search in a Patricia trie. The difference is that in the LC-trie consecutive bits are sometimes examined in groups, while they are examined one at a time in a Patricia trie. For example, if a node in an LC-trie uses four bits for branching, we extract these four bits from the query string and treat them as a 4-digit binary number. Depending on the value of this number we determine in which subtree the search is to be continued. In the corresponding Patricia trie, we extract the same four bits one at a time, traversing four nodes in the tree.

In summary, with respect to space and average depth, the LC-trie is a clear winner among the various trie structures. The only real advantage with a suffix array is the fact that it uses slightly less memory than the trie structure. However,

Input		Suffix Array		Trie (list)		Trie (array)		Patricia Tree		LC-trie	
File	Size	Steps	Size	Depth	Size	Depth	Size	Depth	Size	Depth	Size
Random text	2	10.0	16					12.4	24	5.0	20
	20	13.3	160					15.6	240	4.6	202
	200	16.6	1600					18.9	2400	4.7	2018
DNA	1.7	9.8	14	15.8	41	6.3	20	12.4	21	5.1	17
	17	13	138	19.1	423	8.2	213	15.9	207	5.6	180
	172	16.4	1378	24.5	4279	11.0	2209	20.3	2067	6.8	1824
FAQ, ASCII	1.9	9.9	15	60.2	42	3.9	328	15.5	23	11.2	22
	19	13.2	155	80.7	431	5.7	3593	21.6	232	15.9	222
	193	16.6	1545	115.8	4313	8.2	36122	30.2	2318	21.6	2207
FAQ, Huffman	1.9	9.9	15					13.3	23	7.2	21
	19	13.2	155					18.3	232	9.9	219
	193	16.6	1545					24.5	2318	13.1	2196

Table 2: Empirical results for suffix array and suffix trees. Sizes are measured in kbytes.

recall that a suffix array, as opposed to a trie, requires access to the document itself at each step of the binary search.

Experiment 2: Secondary Memory

In most applications the text data base is too large to fit into main memory. Therefore, in this case, we have conducted a more extensive experiment, using several different kinds of texts.

Method

We have studied the two data structures for large texts described above.

Suffix array/bucket array: In all experiments, we use approximately $n/4$ buckets, as suggested by Manber and Myers. We have chosen the number of buckets in such a way that the proper bucket can be computed by truncating a number of bits. Thus, in most cases (the exception being the k-digit coding) we have chosen the number of buckets to be a power of 2. Each bucket is represented by a pointer, the total space for b buckets is $4b$.

Suffix array/LC-trie: The cutoff value (the maximal number of elements stored in a leaf in the LC-trie) has been chosen in such a way that both the time and space measurements of the two data structures can be easily compared, i.e. both measurements will be better for one structure than for the other.

We have ignored the time spent in the internal structure and only counted the number of accesses to secondary memory required during the binary search, since these are typically much more expensive. Also, we have only counted space required by the internal structures.

Another natural way to try to improve the behavior of the bucketing method is to use a better encoding. For the sake of completeness we have tried both the arithmetic coding suggested by Manber and Myers and a combination of bucketing and Huffman coding. Both methods enhance the performance slightly but do not come close to the performance of the LC-trie.

In addition to the texts from the experiment above we have performed experiments on a selection of the standard texts found in the Calgary/Canterbury text compression corpus [9]. Pictures, geophysical data, object code and texts that were too large to be easily handled by our simulation program have been excluded. The following additional texts were used:

bib: Bibliographic files (refer format).

paper1: Witten, Neal and Cleary: Arithmetic coding for data compression.

paper2: Witten: Computer (in)security.

progc: C source code: compress version 4.0.

progl: Lisp source code: system software.

progp: Pascal source code: prediction by partial matching evaluation program.

trans: Transcript of a session on a terminal.

Input		Bucket Array			LC-trie		
File	Size	Accesses		Size	Accesses		Size
		Aver.	Worst		Aver.	Worst	
Random text	200	2.5	5	131	2.9	5	140
DNA	172	2.9	7	131	3.1	6	137
FAQ, ASCII	193	8.2	14	131	4.8	7	64
FAQ, 96-digit		6.8	14	193			
FAQ, Huffman		5.2	14	131	4.7	7	56
bib, ASCII	111	7.7	13	131	4.9	7	34
bib, 96-digit		6.8	13	111			
bib, Huffman		5.1	11	131	4.9	7	30
paper1, ASCII	53	7.0	11	66	4.0	6	31
paper1, 96-digit		5.9	11	53			
paper1, Huffman		4.1	9	66	3.9	6	27
paper2, ASCII	82	7.7	13	131	4.0	6	50
paper2, 96-digit		6.8	13	82			
paper2, Huffman		4.2	10	131	3.9	6	42
progc, ASCII	40	6.9	11	33	4.1	6	22
progc, 96-digit		5.6	11	40			
progc, Huffman		4.1	8	33	4.0	6	20
progl, ASCII	72	7.9	12	66	4.1	6	41
progl, 96-digit		6.9	12	72			
progl, Huffman		5.1	12	66	4.0	6	39
progp, ASCII	49	7.6	13	33	4.1	6	28
progp, 96-digit		6.4	13	49			
progp, Huffman		5.2	12	33	4.0	6	27
trans, ASCII	94	7.1	12	131	4.0	6	61
trans, 96-digit		6.4	13	94			
trans, Huffman		5.1	11	131	4.0	6	57

Table 3: Empirical results for partial data structures. Sizes are measured in kbytes.

Discussion

Looking at Table 3 we observe that for almost all texts the LC-trie has both a lower average number of accesses to secondary memory and a much smaller size. The only exception is the random string and the DNA sequence, where the bucket array is slightly better. This is to be expected since bucketing is very efficient for uniformly distributed data.

Our experimental results accentuate some of the problems of using a plain bucketing scheme. The major drawback is that for nonuniform distributions many elements may end up in a few buckets while many buckets are empty. This will not only be space inefficient, but also the accumulation of elements will increase the number of accesses to secondary memory. The LC-trie/suffix array implementation has the advantage that the number of elements represented by a leaf is bounded. In our experiments no cutoff value is greater than 100. However, when using the bucket array/suffix array on the text "FAQ" the largest bucket contained no less than 9198 elements and hence the maximum search cost is much higher for the bucket array.

Conclusions

The experimental results presented in this paper strongly indicate that the combination of path compression, data compression, and level compression gives a time and space-efficient data structure for string searching that is is easy to implement and hence should work well in real world applications. In particular, we believe that the LC-trie/suffix array combination will prove to be efficient for very large texts, since it minimizes the number of accesses to slow secondary memory while at the same time using only a small amount of main memory.

Acknowledgments

References

[1] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1991.

[2] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial algorithms on words*, volume 12 of *NATO ASI Series F: Computer and System Sciences*, pages 85–96. Springer-Verlag, 1985.

[3] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. 1st ACM-SIAM SODA*, pages 319–327, 1990.

[4] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46:295–300, 1993.

[5] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, 1952.

[6] E. M. McCreight. A space-economical suffix tree construction algoritm. *Journal of the ACM*, 23(2):262–272, 1976.

[7] E. Ukkonen. On-line construction of suffix trees. Technical report, Department of Computer Science, University of Helsinki, 1993. Report A-1993-1.

[8] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th Annual IEEE Sympos. FOCS*, 1994. To appear.

[9] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.