Chapter 44
Efficient Suffix Trees on Secondary Storage (extended abstract) *

David R. Clark J. Ian Munro

Abstract

We present a new representation for suffix trees, a data structure used in full text searching, that uses little more storage than the lg n bits per index point required to store the list of index points. We also show algorithms for maintaining the structure on secondary storage in such a way that we minimize the number of disk accesses while searching and we can efficiently handle changes to the text. Using this new representation, suffix trees are competitive in terms of space with, and actually offer significantly better performance than, existing methods for full text searching. While we present new algorithms and data structures in this paper, the emphasis is on practical searching methods that have been empirically verified against real data.

1 Introduction

The electronic storage and retrieval of information in large documents such as encyclopedias and other reference works requires the use of searching systems that are efficient both in time and storage requirements. In this paper, we present a practical data structure for full text searching in very large documents (billions of characters are quite possible). As well, we present methods for maintaining this structure under updates to the text. The new structure offers an order of magnitude improvement in performance over current methods and is appropriate for use on CD-ROM, a medium for which many competing structures are not well suited. The new structure is compact, requiring approximately the same storage as a suffix array and a fraction of the storage required by previous suffix tree representations.

In full text searching we want to preprocess a document so that given a query phrase all occurrences of the phrase in the original document can be located very quickly. One effective method to do this is to create a trie whose entries are the suffixes of the original text starting at each possible index point (i.e. starting at each word or character, as appropriate for the application). If the paths in the trie are truncated at the point where they represent a unique suffix and all the internal nodes at which no branching occurs are removed then we have a version of the *suffix tree*, see Weiner[23] or McCreight[16]. In such a tree, each leaf contains the offset of the appropriate suffix in the original document. Gonnet et al.[7] obtained the *Pat tree*[1] by storing the binary representation of each suffix in a variant of Morrison's PATRICIA structure[18]. The main advantage of suffix trees and Pat trees is their ability to locate a subtree of answers in time strictly proportional to the length of the query string. Once the subtree is located by tracing the query phrase down the trie, a single access to an arbitrary suffix in the subtree is required to determine if the entire subtree contains matches to the query or if there are no matches in the document.

Both Gonnet et al.[7] and Manber and Myers[15] report that, with careful implementation, suffix trees and Pat trees require approximately 17 bytes per index point when searching documents of up to 2^{32} characters. In practice, a 1 megabyte file will require 3 megabytes to store a word index (assuming typical English language text) and 17 megabytes for a character index. As a consequence, both groups propose dropping the trie structure altogether and simply perform a binary search on the array of references to the text. Searches on such a suffix array structure take time logarithmic in the text size but require approximately $\lg n$ random accesses into each of the suffix array and the text, where n is the size of the document being searched. Both groups report methods for obtaining some reduction in the number of random accesses by augmenting the structure, but, in general, at least lg n accesses are still required. These methods increase the size of the suffix array by about 25%. The $2 \lg n$ access cost for searching a document is quite acceptable provided the document is stored in primary storage. When using slower secondary storage,

*Department of Computer Science, University of Waterloo, Waterloo, ON, N2L-3G1, Canada
E-mail: {drclark,imunro}@uwaterloo.ca
This research was supported in part by the Natural Sciences and Engineering Research Council under grant A8237 and by the Information Technology Research Centre of Ontario.

[1]Not to be confused with the PAT^{TM} system.

such as magnetic or optical disk, it can be excessively large. The primary contribution of this paper is a representation for the Pat tree whose size is comparable to that of a suffix array. Furthermore, the data is organized to dramatically reduce the number of disk accesses: from 40 or 50 to 3 or 4 using reasonably sized pages on our large test document!

While Merrett and Shang[17][21] also attack this problem, our use of a more efficient encoding and optimal partitioning rules leads to a much more efficient set of structures. In particular, on a set of real world test documents containing 100 million index points each, Merrett and Shang report an average number of accesses between 5.2 and 7.1 with the maximum number of accesses varying between 11.0 and 46.0. For comparison, the structure reported here required at most 4 accesses on a slightly larger document when using the same page size. In addition the new structure is much less processor intensive and so can make effective use of larger page sizes to search the same document in 2 disk accesses (only appropriate for use on CD-ROM). Recently, Ferragina and Grossi developed the SB-Tree[5] and an efficient implementation of it[6] requiring about 12.3 bytes per index point with performance tradeoffs allowing reductions to 6.3 or fewer bytes per index point. While the worst case performance of our structure is significantly poorer (linear vs. logarithmic) than that of the SB-Tree, the worst case data is sufficiently unlikely that we expect it to perform better on real world data. However, the guaranteed worst case behaviour of the SB-Tree will be attractive in some applications. Due to issues of page fill ratios and time-space tradeoffs in the implementation of both structures, it is not clear which will be more compact in practice. Compact tries were also investigated by Darragh, Cleary and Witten[2] but their structure is not directly comparable due to its probabilistic nature. The updating of suffix trees and special structures for dynamic text have been considered by other authors[9][16][4] but these works only deal with the primary storage case. Finally, Barbosa et al. considered the physical attributes of magnetic disk to optimize the time taken to perform the $2\lg n$ accesses required by suffix arrays[1]. A survey of other text searching methods can be found in Faloutsos[3].

1.1 Suffix Trees and Related Structures Given a text string $S = s_1s_2s_3...s_n$ where each s_i is a member of an alphabet Σ, we want to preprocess S such that given a pattern $P = p_1p_2p_3...p_m, (p_i \in \Sigma)$, the set $\{i : s_i..s_{i+m-1} = P\}$ can be found as efficiently as possible. Suffix based search methods operate by searching the set of suffixes of the string S. In order to ensure that each suffix corresponds to a unique position in the text, a special

Offset	Suffix	Unique Prefix
1	*abccabca$*	*abcc*
2	*bccabca$*	*bcc*
3	*ccabca$*	*cc*
4	*cabca$*	*cab*
5	*abca$*	*abca*
6	*bca$*	*bca*
7	*ca$*	*ca$*
8	*a$*	*a$*

Table 1: Suffixes of *abccabca$*

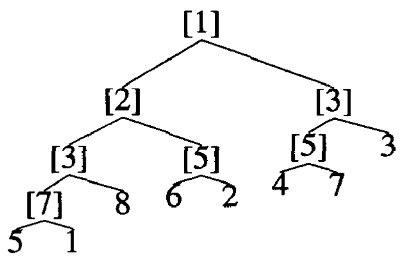

Figure 1: Pat tree, [n] indicates the bit to test

character "$," not in Σ, is appended to S. Given the string S=*abccabca$*, the suffixes of S are shown in Table 1. Each suffix has a minimal prefix that distinguishes it from the other suffixes. This unique identifier is very important to suffix based search methods and appears in the third column of Table 1. A suffix tree is a trie on the unique suffix identifiers that has had all degree one nodes removed to save space.

The suffix tree structure considered here is the Pat tree of Gonnet et al.[7]. Pat trees are a form of Digital Tree Search[12] resulting from a merging of suffix trees and the Patricia search method of Morrison[18]. Given a binary encoding of $\Sigma \cup \{\$\}$, the Pat tree is obtained by encoding the suffixes of S as bit strings and storing them in a Patricia tree for searching. Each internal node in a Pat tree is labelled with the first bit offset at which the suffixes in the sub-tree differ and has two children containing all the suffixes having a 0 at the differing bit in one subtree and those having a 1 in the other. By adopting the convention of 0 for the left child and 1 for the right, we can encode the suffix tree as a binary tree with each internal node labeled by the offset of the bit used to distinguish the suffixes under each node. Like a suffix tree, the leaves of the Pat tree contain the offsets of the appropriate suffixes. Using the encoding a=00, b=01, c=10 and \$=11, the Pat tree for the example string is given in Figure 1.

A Pat tree is searched by generating the binary encoding of the pattern and then traversing the tree. At each internal node, the bit offset is used to select a bit from the pattern. Based on the bit value, the traversal continues with either the left or the right child of the node. Because the search can skip bits in the pattern, the termination of the search is more complex than that of simple suffix trees. If the search terminates at a leaf node, then the pattern must be compared to the leaf suffix to see if it matches. If the end of the pattern is encountered before a leaf, then a representative suffix from the current subtree must be chosen and compared to the pattern. The representative matches the pattern if and only if all of the suffixes in the subtree match the pattern. In practice, the offset information stored in each node is a skip value one less than the difference between the offset value of the node and its parent (with an implicit parent offset of 0 for the root). The actual offset is accumulated as the tree is traversed. Provided care is taken to ensure that locating a sample for comparison can be performed efficiently, the search cost of Pat trees is the same as that of suffix trees, $O(m)$ where m is the size of the pattern.

1.2 Storage Requirements Computer representations of suffix based structures require the use of pointers and text offsets. For the purposes of comparison, it is useful and reasonable to assume that each of these require lg n bits. Manber and Myers performed an analysis of various possible representations of suffix trees and determined that approximately 17 bytes per index point were used in the most compact representations [15]. Under the assumption that their system was capable of handling documents of at most 2^{32} characters, 17 bytes equates to 4.25 $\lg n$ for the texts considered in their report. This result agrees with the "n to 5n words" reported by Gonnet et al.[7]. The use of Pat trees can reduce the storage requirements to 3 lg n bits if the obvious implementation of a node as two pointers and an integer skip is used. On our large test document (about 2^{29} characters), this results in a word index roughly twice the size of the text - a suffix tree would be three times the size. If all the characters in a document are indexed, the size each of these indices will be increased by a factor of approximately five.

2 Compact Pat Trees

The information stored in the Pat tree can be broken into three categories

- the tree structure,
- the skip values,
- the suffix offsets in the leaves.

By efficiently storing each class of information, our approach, Compact Pat Trees (CPTs), matches the storage efficiency of other suffix based search structures while retaining the functionality of Pat trees.

2.1 Static Text on Primary Storage In this section we use a compact tree encoding to represent the tree portion of the Pat tree and obtain an efficient data structure for searching static text in primary storage. This structure is the basis of later structures for searching on secondary storage. In order to implement the Pat tree search operations, the encoding of the tree structure must provide the following functionality:

- efficient selection of the left and right children of a node,
- support for the inclusion of constant size "fields" for each internal node, the skip, and another constant size field for each leaf, the suffix offsets. Given a node or leaf, we need to be able to efficiently determine the field values.
- given a node, efficiently retrieve the suffix offset field information from some leaf descended from the node.

In each case, we require that the operations be performed in a constant number of operations on $\lg n$ bit size objects. Finally, we want an encoding that is as compact as we can find. Given that there are $C_n = \frac{1}{n+1}\binom{2n}{n}$ binary trees on n nodes, a compact encoding of the tree structure for a Pat tree should require about $\lg C_{n-1}$ bits. Using Stirling's approximation to the logarithm of the factorial function, $\lg C_n$ can be shown to be approximately $2n$. The survey papers of Mäkinen[14] and Katajainen and Mäkinen[11] present many techniques for binary presentations of binary trees that attain the $2n$ bound, however none meet the criteria above. For this application, we use a slightly larger encoding developed by Jacobson[10] because it allows direct implementation of tree traversals on the encoded form of the tree. Most other representations require at least linear time to implement tree traversals[14]. While Jacobson also presents a $2n+o(n)$ representation, we chose the encoding below because it allows a more straightforward implementation of the Pat tree and includes the size of each subtree.

Jacobson's encoding represents each tree as a bit string of the form

Header	Left Tree Encoding	Right Tree Encoding

where the header contains two fields:

- a single bit indicating which of the two children is smaller with an arbitrary choice made in the case of a tie, and,

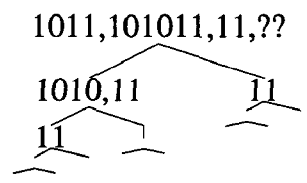

Figure 2: Tree with Encoding

- a prefix coded integer indicating the size of the smaller child. Jacobson forms his prefix code for an integer i by writing the binary representation of i interleaved with a unary encoding of $\lg i$. We use a similar code obtained by concatenating the unary encoding of $\lg i+1$ with the binary encoding of $i+1$ because it saves one bit per node. The size of the encoding is $2\lceil\lg(i+2)\rceil - 1$. More sophisticated encodings are possible but lead to only slightly smaller total sizes[10].

The critical point is that the encoding is padded so that the size of the encoding is independent of shape of the tree. This feature allows efficient implementation of the traversal operations.

The size of the encoding is $B(n) = 3n - 2\lfloor\lg(n+1)\rfloor - 2v_2(n+1) + 2 - [n \text{ is odd}]$ where v_2, which is defined in[8], is the number of ones in a numbers binary representation. From this equation, it is clear that $B(n) < 3n$ so the total storage requirement for the binary tree information is less than three bits per node. Using this representation, the tree structure in the Pat tree example is represented by the bit string 101110101111__, where the actual value of the last two bits is unimportant. The tree in Figure 2 shows this tree with each subtree labeled with its description.

The simple formula for $B(n)$ allows efficient implementation of the operations of fetching the left and right children of a node. The left child is found immediately following the prefix code of the integer giving the size of the smaller tree and the right child can be found immediately after the description of the left child whose size can be computed based on the number of nodes in the left subtree. This size can in turn be computed from the values of "smaller," "size of smaller," and the size of the overall tree. Because the tree traversal operation above returns, and requires, both the offset of the child's encoding and its size, we can also track the number of leaves to the left of the leftmost leaf in the current subtree. This is achieved by starting an index at zero and increasing it by the size of the left subtree plus one each time the right child is taken. Keeping this index allows us to store the suffix offsets in an array sorted lexicographically, a suffix array, and to determine the exact sub-array that contains the response to the query.

2.2 Storing the Skip

Compressing the skip information requires an understanding of the distribution of the skip values. For the purposes of analysing the skips, temporarily assume the suffixes are strings of independent uniformly sampled bits with 0 and 1 having equal probability. Consider an internal node with k leaves in its subtree, then the probability that the skip value of the node is greater than j is the same as the probability that k random bit strings match in their first $j+1$ bits. This value is easily seen to be $2^{-(j+1)(k-1)}$. From this formula, we see that the majority of the skip values are zero and that the likelihood of higher values decreases geometrically. We have verified this behaviour on real world documents. Shang also noted the rapid decline in the number of large skip values[21].

The low likelihood of large skip values leads to a simple method of compactly encoding the skip values. We reserve a small fixed number of bits to hold the skip value for each internal node and introduce a strategy to resolve problems caused by skip values that overflow this field. We handle overflows by inserting a new node (and a leaf) into the tree and distributing the skip bits from the original node across the skip fields of the new and the original node. The dummy leaf node must have some special key value that allows it to be easily recognized (typically all 0s or all 1s). Multiple overflow nodes and leaves can be inserted for extremely large skip values. When traversing the tree, simply checking for a single leaf with the dummy value is sufficient to determine if the skip should be checked or the bits concatenated to obtain the true skip value. The use of this overflow handling mechanism has one slight drawback in that the subtree size is no longer the exact size of the answer. However, the subtree size is still a good estimate of the size and also an upper bound on the size of the answer.

There are two approximations in the argument above that deserve serious consideration. The first is the assumption that the bit strings in the suffixes are independent. Clearly, being possibly overlapping substrings of a single string, this is not the case. However, this approximation does not seem inappropriate because of the, assumed large, size of the underlying string. The more serious problem above is the assumption that the binary string is generated by a uniform symmetric random process. This is *not* a good model of written text or other large documents. For example, English text coded in ASCII will have the high order bit of each byte set to zero and is unlikely to contain the codes 0..31. In order to resolve this weakness in our arguments (and the structures based on it), we convert all input strings to an approximation to a uniform symmetric binary string by running them through a data compression algorithm. Our current search engine uses fixed size codes and a simple uniform model of the text. While this model is the same as using a compact encoding for the character set, so far it has proven adequate. We are considering the use of a model including digrams for a future version of the search engine. The data structure will continue to operate without this encoding step but the loss of one or more bits per index point in storage efficiency should be expected.

THEOREM 2.1. *Under the assumptions above, the expected size of the Compact Pat Tree can be made less than* $3.5 + \lg n + \lg\lg n + O\left(\frac{\lg\lg\lg n}{\lg n}\right)$ *bits per node. We achieve this by setting the skip field size to* $\lg\lg\lg n$.

Proof. Under the assumptions and using the skip field size above, $\frac{n}{2\lg n} + \frac{n}{2^{(\lg\lg n)^2+1}} + \frac{n}{2^{(\lg\lg n)^3+1}} + \ldots$ is a significant overestimate of the number of overflow nodes obtained by assuming each internal node has exactly two suffixes below it. The total storage is then less than $\lg n + (3 + \lg n + \lg\lg\lg n)\left(n - 1 + \frac{n}{2\lg n} + \frac{n}{2^{(\lg\lg n)^2+1}} + \ldots\right)$. Multiplying through and dividing by n, we obtain the desired result.

In real text searching applications, the index is slightly larger because a larger skip field size is more effective. We typically use a skip field size of 5 or 6 depending on the document size. Even these larger sizes result in a very small index requiring about 10 bits per index point to represent the trie. For storing the skip values, we simply add a third constant size field to the tree header and modify $B(n)$ appropriately by adding n times the size of the skip field.

2.3 Suffix Offsets The suffix offsets take up the bulk of the storage used by the CPT and other suffix based structures. While the suffix offsets do not easily admit compression, they can be stored more compactly if we are willing to make some sacrifices on performance. To do this, we use a technique also used by Shang's PaTries[21]. If k low order bits in the suffix offsets are omitted from the CPT structure, nk bits are saved in the final index. This change incurs a 2^k cost in the searching time and a 2^k multiplicative factor on the conversion of a node to its list of leaf offsets because each offset value will require a search through 2^k characters to locate the exact occurrence of the pattern.

2.4 Empirical Results For presenting empirical results on text searching structures, we will use three documents as test cases:

- Holmes, an ASCII encoded extract from the works of Sir Arthur Conan Doyle,
- Bible, an SGML[20] encoded version of the King James Bible,
- OED, an SGML encoded version of the Oxford English Dictionary[19].

Name	#Characters	#Words
Holmes	238551	43745
Bible	5553621	1202504
OED	545578702	108687644

Table 2: Sample Documents

Text	Skip Field Size	#Overflow Nodes	Index Size (bytes)
Holmes	4	5486	153847
	5	1580	147307
	6	242	148457
Bible	4	264903	5502777
	5	127150	5152410
	6	48407	5003644
	7	19377	5040260

Table 3: Index Sizes for Sample Documents

The search engine performs case conversion and the mapping of special characters to blanks prior to indexing and searching. Various properties of these documents are shown in Table 2. Table 3 shows the number of overflow nodes and the resulting index sizes for two of the sample documents (the OED is too large for construction of an index in primary storage). From this table we see that the optimal skip sizes are 5 for Holmes and 6 for the Bible. In addition, the size of the final index for non-optimal values is still close to the optimal size.

3 Static Text on Secondary Storage

Searching methods for large text databases must be concerned with more than asymptotic time requirements; storage costs and the exact number of secondary storage access are also critical. If the index requires k bytes per character in the text, the index size will be k times the size of the document. For large documents the storage cost quickly becomes prohibitive as k gets large. Similarly, while the asymptotic operation count of the algorithm is important, the number of accesses to secondary storage is likely to have a far greater effect on the performance, and even the feasibility, of the index. The first half of the paper dealt with controlling the storage requirements of Pat trees. In the second half of the paper we concentrate on controlling the number of accesses to secondary storage during searching.

3.1 Partitioned Compact Pat Trees To control the accesses to secondary storage during searching we use the method suggested by Gonnet et al.[7]: decomposing the tree into disk block sized pieces (each called a partition). Each partition of the tree is stored using the CPT structure from the previous sections. The only change required to the CPT structure for storing the partitions is that the offset pointers in a block may now point to either a suffix in the text or a subtree (partition) so an extra bit is required to distinguish these two cases. We use a greedy bottom up partitioning algorithm and show that such a partitioning minimizes the maximum number of blocks accessed when traveling from the root to any leaf. The algorithm for building the index on secondary storage will be described in the full version of the paper.

The partitioning algorithm starts by assigning each leaf its own partition and a page depth of 1. Working upward, we apply the rules below at each node. A simple induction proof shows that the rules above produce a min-max optimal partitioning of the tree such that no other optimal partitioning has a smaller root block. The proof of optimality parallels the partitioning rules.

```
    If both children have the same page depth
(1)   if both children's partitions and the
              current node will fit in a block,
          merge the partitions of the children
              and add the current node
          set the page depth of the current
              node to that of the children
(2)   else
          close off the partitions of the
              children
          create a new partition for the
              current node
          set the page depth of the current
              node to one more than that of
              the children
  else
      close off the partition of the child
              with the lesser depth
(3)   if the current node plus the partition
              of the larger child will fit on
              a block
          add the current node to the child's
              partition
          set the page depth of the current
              node to match the child
(4)   else
          close off the partition of the
              remaining child
          create a new partition for the
              current node
          set the page depth of the current
              node to one more than that of
              the child
```

While the partitioning rules minimize the maximum number of secondary storage accesses, they can produce many small pages and poor fill ratios. There are several possible methods to alleviate this problem, including:

1. when a page is closed off, scan its children from smallest to largest to determine if they can be merged with the parent,
2. modify the rules to ensure a certain minimum fill ratio (e.g. all pages have to be 1/4 or 1/5 full),
3. pack multiple logical pages in each physical page,
4. ignore physical page boundaries when placing logical pages on disk.

Change one should be a part of any implementation of these rules. Change two will result in non-optimal partitioning in some cases but should be worthwhile in a practical system. The third technique should drastically minimize the storage requirements in practice but has a low guaranteed storage utilization and introduces some complications in the management of secondary storage. The last technique minimizes the storage requirements at a small cost for the potential transfer of an extra page of data on each access. In our current implementation for static text we use the first and fourth techniques.

We can bound the maximum number of pages traversed on any root leaf path in terms of the number of nodes in the Pat tree and the depth of the Pat tree.

THEOREM 3.1. *The page depth is less than* $1 + \left\lceil \frac{H}{\sqrt{p}} \right\rceil + \lceil 2 \log_p n \rceil$ *where H is the height of the Pat tree.* Szpankowski shows that under very reasonable conditions on the text, H is logarithmic in n with probability one[22]. Linking these two results we obtain an expected performance bound logarithmic in n.

3.2 Empirical Results When producing the empirical results for indices on secondary storage, the optimal skip field sizes from the primary storage case were used for Holmes and the Bible (see Table 3). For the OED, a skip field size of six was used based on the experience with the Bible.

In each case, the depth of the tree is equal to the number of accesses to secondary storage needed to perform a search if the root block is held in memory. In the static case, we do not enforce page alignment so an access will consist of a seek followed by the transfer of at most two pages worth of data. The results above are for full suffix pointers. Truncating the suffix offsets would

Text	Page Size	Depth	#Pages	Index Size
Holmes	1K	2	178	114220
	2K	2	85	141875
	4K	2	40	141733
	8K	2	19	141683
Bible	1K	3	10781	4875898
	2K	3	5459	4860069
	4K	3	2985	4853663
	8K	2	1397	4849707
OED	1K	5	1285521	583700474
	2K	4	698923	580757446
	4K	4	374414	579166308
	8K	3	195994	578344112
	100K	2	18409	577617776

Table 4: Results for Static Text on Secondary Storage

significantly reduce the size of the indices at a cost in computation time but without significantly increasing the number of disk seeks. For each seek, a small amount more data would be read. The bottom result in Table 4 is only of interest for indices on CD-ROM where the very high seek time makes the reading of large quantities of data worthwhile. The transfer time for the 100K page size is still significantly less than the expected seek time for this device. We note that in this case of a CD-ROM sized text and a similarly sized index we can perform searching using two accesses to the disk. While we do not cover the index building algorithm here, we will state that the indices on the OED were built in 5 to 6 hours on a fast machine using 32 Mb of memory. In our experience, this is competitive with other indexing algorithms.

4 Dynamic Text on Secondary Storage

The general approach to updating the static representation from the previous section is to search each suffix of the modified document in the CPT tree and then make appropriate changes to the structure based on the path searched. While updating the tree, it may become necessary to re-partition the tree in order to retain optimality. Because the partitioning algorithm used is based on information local to a subtree, the effects of repartitioning are limited to the root-change point path. In fact, we do not have to consider all nodes on that path but can limit ourselves to those nodes at the partition boundaries. This differs considerably from other partitioning strategies, for example the method of Lukes[13], where a local change to a tree can have global side-effects on the partition.

The updating of the CPT is best discussed in terms of the insertion or deletion of suffixes because other update models use suffix operations in their implementation. While constant (amortized) time solutions to the suffix insertion and deletion problem do exist (see McCreight[16]), they require the maintenance of extra data in each node. The solutions used here require time proportional to the tree depth but operate on the compact form of the tree.

To insert a new suffix into a CPT, the following steps are taken:

1. search the tree using the new suffix until we reach a leaf
2. determine the first bit position at which the new suffix and the suffix located at the leaf from step 1 differ
3. find the block containing the edge on the root-leaf path that skips over the bit position above
4. split the edge by inserting a new node having as children the new suffix and the old subtree reached by the edge
5. set the skip numbers of the new node and the old subtree appropriately

Deletion of a suffix is also easily handled:

1. locate the suffix's leaf in the CPT and fetch the block
2. remove the suffix and its parent by replacing the parent with a straight through edge
3. update the skip value in the suffix's ex-sibling

After these operations, we may have to adjust the partitioning so that it still matches the rules given earlier.

In order to update the CPT so that it reflects a change in the underlying character string, multiple tree operations may be required. For example, when changing the third character of the example string *abccabca$* from a *c* to a *b*, we must delete the suffixes *cc*, *bcc* and *abcc* and replace them with *bcab*, *bb*, and *abb*. The distance we must scan backward is bounded by the maximum offset in the tree which is in turn expected to be logarithmic in the text size[22]. A second, real world, model of updates allows the insertion, deletion and replacement of entire sub-documents in a database. Under this model of updates, the number of suffix operations is the same as the number of words in the documents being manipulated. We use the latter model in our tests. This model is equivalent to the External Dynamic Substring Search(EDSS) problem studied by Ferragina and Grossi[4]. An update to a block can cause two situations that require corrective action:

- on insert, a block may become too large,

Text	Depth	#Pages	Fill Ratio	Disk Writes per word
Holmes	2	197	43%	1.01
Bible	3	6914	38%	1.02

Table 5: Update Costs for Dynamic Text

- on delete, a block may become small enough to hold its root's parent and its sibling's block when the sibling has the same page depth

In each case, we re-apply the partitioning algorithm to a subset of the nodes on the root-change point path. Consider the case of a simple insertion without any overflow nodes (it is conceivable but unlikely that each overflow node could cause the consideration of new nodes). After locating the block containing the insertion point we need only check if the partition will still fit in a block after the node is inserted. If it does, we write the data back and the insertion is done. If not, the root is moved into either a new partition or its parent's partition, depending on the page depth of the parent block, and the process continues recursively. In such a case, the sibling will also have to be moved into its own partition if it is not already in one. Empirically, we observe that we expect only slightly more than one page write per node insertion. This behaviour likely results from the low page fill ratios and we expect a slight increase in the number of writes/word when we apply the modified partitioning rules or packing methods to this case. The case for deletion is very similar. Because the partitioning produced is dependent only on the current Pat tree, and hence the text, the quality of the partitioning does not degrade over time and periodic re-indexing is not necessary.

In order to test the CPT under the second model of updates, the two smaller test documents were broken down into sub-documents. We then tested the deletion and subsequent insertion of complete sub-documents. The number of blocks written for each word in the newly inserted document are shown in Figure 5. We have not as yet made any attempt to address the low fill ratios in the dynamic case. Several strategies including the alternate partitioning rules and some limited bin packing could be applied to this problem. The results above should be treated as preliminary because we are still investigating the dynamic CPT case.

5 Conclusions

In this paper we show several representations for Pat trees that use significantly less storage than previous methods. In particular, we present:

1. a new representation for static Pat trees in primary storage that allows efficient searching with an expect storage cost of only $3.5 + \lg n + \lg\lg\lg n$ bits per node for random documents. We also demonstrate the structure's practicality on realistic test documents.

2. a new representation for static Pat trees in secondary storage that is little larger than a suffix array and offers significantly better performance than that offered by suffix arrays.

3. methods for managing the structure mentioned in point 2 that allow us to efficiently handle updates to documents on secondary storage.

In addition to its uses in string processing, we believe the CPT structure for searching static text in primary storage will find uses in computational biochemistry where it will allow fast searching of even longer strings of genetic information. The structures presented here for searching on secondary storage are extremely practical solutions to the efficient phrase searching of large dynamic documents. It is yet to be seen if their added functionality is sufficient to make them a rival of inverted word lists in the large text database field. The new structures are clearly more suitable for many applications because of their ability to handle phrase and regular expression searching. While the CPT is more compact than traditional representations of suffix trees, it is still larger than an inverted word list which typically requires about 60-70% of the text size (as an experiment in data compression, Witten, Bell and Nevill reduced this to 30% of the text size[24]). Depending on the structure used to store the inverted word lists, the CPT structures may offer faster performance in addition to their more flexible searching.

5.1 Acknowledgments The authors thank the referees for pointers to some useful references and P. Ferragina and R. Grossi for supplying a preliminary version of their paper [6].

References

[1] E.F. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg, and N. Ziviani. Optimized binary search and text retrieval. In *Algorithms - ESA '95, Third Annual European Symposium*, pages 311–326, September 1995.

[2] J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: A compact representation of trees. *Software - Practice and Experience*, 23(3):277–291, March 1993.

[3] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, March 1985.

[4] P. Ferragina and R. Grossi. Fast incremental text editing. *ACM-SIAM Symposium on Dicrete Algorithms*, pages 531–540, 1995.

[5] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. *ACM Symposium on the Theory of Computing*, pages 693–701, 1995.

[6] P. Ferragina and R. Grossi. Fast string searching on secondary storage: Theoretical developments and experimental results. *ACM-SIAM Symposium on Dicrete Algorithms*, 1996.

[7] G. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographic indices for text: Inverted files vs. pat trees. Technical Report OED-91-01, Centre for the New OED., University of Waterloo, 1991.

[8] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, New York, 1989.

[9] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. In *Proc. of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 697–704, January 1994.

[10] G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Carnegie Mellon University, 1989.

[11] J. Katajainen and E. Makinen. Tree compression and optimization with applications. *Int. Journal Comput. Science*, 1(4):425–447, December 1990.

[12] D. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3.* Addison-Wesley, Reading Mass., 1973.

[13] J.A. Lukes. Efficient algorithm for partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[14] E. Makinen. A survey on binary tree codings. *The Computer Journal*, 34(5):438–443, 1991.

[15] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[16] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.

[17] T.H. Merrett and H. Shang. Trie methods for representing text. Technical Report SOCS-93.5, McGill University, 1993.

[18] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the Association for Computing Machinery*, 15(4):514–524, October 1968.

[19] *The Oxford English Dictionary, Second Edition.* Clarendon Press, Oxford, 1989.

[20] *(ISO 8879) Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML).* International Organization for Standardization (ISO), 1986.

[21] H. Shang. *Trie Methods for Text and Spatial Data Structures on Secondary Storage.* PhD thesis, McGill University, 1995.

[22] W. Szpankowski. Suffix trees revisited (un)expected asymptotic behavior. Technical Report CSD-TR-91-063, Purdue University, 1991.

[23] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, October 1973.

[24] I. H. Witten, T. C. Bell, and C. G. Nevill. Modeles for compression in full-text retrieval systems. In *Data Compression Conference*, pages 23–32, April 1991.