CPS-tree: A Compact Partitioned Suffix Tree for Disk-based Indexing on Large Genome Sequences

Swee-Seong Wong *

Wing-Kin Sung

Limsoon Wong

School of Computing
National University of Singapore, Singapore
{wongss, ksung, wongls}@comp.nus.edu.sg

Abstract

Suffix tree is an important data structure for indexing a long sequence (like a genome sequence) or a concatenation of sequences. It finds many applications in practice, especially in the domain of bioinformatics. Suffix tree allows for efficient pattern search with time independent of the sequence length. However, the performance of disk-based suffix tree is a concern as it is slowed down significantly by poor localized access resulting in high IO disk access.

The focus of this paper is to design an IO-efficient and Compact Partitioned Suffix tree representation (CPS-tree) on disk. We show that representing suffix tree using CPS-tree has several advantages. First, our representation allows us to visit any node in the suffix tree by accessing at most $\log n$ *pages of the tree where* n *is the length of the sequence. Second, our storage scheme improves the access pattern and reduces the number of page fault resulting in efficient search retrieval and efficient tree traversal operations. Third, by bit packing, our index is compact. Experimental results show that CPS-tree outperforms other indexes on disk. When fully loaded into the main memory, CPS-tree is still efficient. Hence, we expect CPS-tree to be a good disk-based representation of suffix tree, with potential use in practical applications.*

1. Introduction

Suffix tree is an important data structure for indexing text string since it can answer pattern searching query efficiently independent of the text string size. There are many practical applications that rely on suffix tree, especially for processing biological sequence data. As various genome sequencing projects are ongoing and more genome sequences are made known, the application of suffix tree on biological research is expected to increase.

Since genome size is in the order of gigabytes, maintaining suffix trees becomes an important issue. There are two immediate problems. The first problem is on constructing suffix tree efficiently. Fortunately, a suffix tree (or a suffix array) for human genome of 3 billion characters can now be constructed within 30 hours [7, 9]. Hence, the problem on suffix tree construction has largely been solved in practice.

The second problem is on accessing the suffix tree. As the genome database gets bigger, maintaining suffix tree in memory is no longer feasible. We need to have a disk-based representation of suffix tree that allows for efficient access. We have seen a number of disk-based representations of suffix tree [1, 4, 9] in the literature. However, these disk-based suffix trees either fail to support all the general suffix tree operations well or have high IO disk access for certain operations.

This paper focuses on having a practical and efficient suffix tree implementation on disk that supports various suffix tree operations efficiently. We proposed a Compact Partitioned Suffix tree representation (CPS-tree) for disk-based access. Our CPS-tree achieves good IO bound and time complexity, and is shown to be efficient on real datasets as well.

There are many applications on genome sequences that use suffix tree structure to search for patterns. These applications are memory based and hence only handle genomes that are small. For large genome that needs to reside on disk, the disk IO efficiency becomes an important issue. As such we study the disk IO efficiency of our proposed suffix tree, both in worst case performance and in practice, to answer exact match problem and also to handle general tree traversal operations. In Table 1, we present the worst case disk access performance of our CPS-tree and compare with other proposed suffix structures in the literature. Table 2 gives a list of notations used throughout this paper for easy refer-

*Supported by the Institute for Infocomm Research, a member of A*STAR, Singapore.

Suffix structure	Exact match query	Exact match count query	Edge label access	Child node access
SB-tree[4]	$\log_B n + \frac{m+occ}{B}$	$\log_B n + \frac{m}{B}$	$\log_B n + \frac{\ell}{B}$	$\log_B n + \frac{\ell}{B}$
CPT[1]	$\frac{H}{\sqrt{B}} + \log_B n + \frac{m+occ}{B}$	$\frac{H}{\sqrt{B}} + \log_B n + \frac{m+occ}{B}$	$\frac{H}{\sqrt{B}} + \log_B n + \frac{\ell}{B}$	$\log \lvert A \rvert$
WOTD-tree[5, 9]	$\min\{m, H\} + \frac{m+occ}{B}$	$\min\{m, H\} + \frac{m+occ}{B}$	$\frac{\ell}{B}$	1
CPS-tree	$\min\{m, \log n\} + \frac{m+occ}{B}$	$\min\{m, \log n\} + \frac{m}{B}$	$\frac{\ell}{B}$	1
suffix array[8]	$m \log n + \frac{occ}{B}$	$m \log n$	$\log n + \frac{\ell}{B}$	$\log n + \frac{\ell}{B}$

The WOTD-tree is generated using the TDD construction algorithm[9]. The SB-tree does not maintain the original suffix tree structure, so we derived the worst case complexity for the node and edge label access to recover the original suffix tree information. Note that H, the depth of the suffix tree, is bounded by $O(n)$.

Table 1. Worst case big-O IO bounds for operations on various proposed suffix data structures.

Notation	Description
n	Index size
N	Length of text string to be indexed
B	Memory page size (in bytes)
m	Query string length
occ	Number of matching occurrences of the query on the text
$\lvert A \rvert$	Alphabet size + 1
ℓ	Edge label length
H	Suffix tree depth

Table 2. Description of notations used.

ence. Exact match query on suffix tree structures is generally IO bounded by m, which can be in the order of n. When $\log_B n \leq m$, SB-tree gives the best worst case IO bound for exact match query running in $O(\log_B n+(m+occ)/B)$ (refer to Table 1). Here we present CPS-tree with an IO bound of $O(\min\{m, \log n\} + (m+occ)/B)$. Furthermore, if only the number of matches is needed, the exact match count query is IO bounded by $O(\min\{m, \log n\}+m/B)$ for CPS-tree, independent of occ, the number of occurrences.

On DNA sequences, we find that the average performance of CPS-tree gives 3 logical block accesses per exact match query on the suffix tree. This is on par with the reported average performances of SB-tree and CPT. Our experimental results also show that CPS-tree performs better both in memory and on disk for exact string match query when compared to other suffix data structures like WOTD-tree and suffix array.

Next, structures like SB-tree, CPT and suffix array are not well-suited for basic tree traversal operations like child node and edge label access. On the other hand, CPS-tree and WOTD-tree cater to these operations so that the bound on the IO access is independent of n. Tree traversal operations are essential to handle complex queries and to support various search techniques over a suffix tree.

As for disk space usage, we are able to keep the CPS-tree compact. Experiments show that for DNA sequence, we need 7N to 9N bytes to store our bit-packed suffix tree on disk, assuming that every position on the text is to be indexed and addressable using a 4 bytes word. Our scheme is comparable to the most space efficient suffix tree representations [1, 4, 6]. Retrieval of the matching occurrences on the text, given a search string, can be performed by traversing the suffix tree to access all the leaf nodes within a subtree. Alternatively, this can be handled more efficiently by using additional 4N bytes to store the suffix array on disk so that the occurrences which correspond to an index range in the array can be retrieved sequentially from disk directly. However, this increases the total index size on disk to range within 11N to 13N bytes. Most suffix tree implementations take 17N to 65N bytes [6] with more compact version in 12.5N bytes [5, 9], while a suffix array [8] uses 4N bytes.

In brief, we made improvements to the techniques used in CPS-tree to give the following results: (1) Fast searching and traversal of the suffix tree in terms of IO paging and computational time based on partitioning and buffering, (2) fast enumeration of the occurrences by storing the ranges on the suffix array, in the tree, together with the suffix array on disk, and lastly, (3) compact the suffix tree size using bit-packing representation and other space optimization.

2. Structures and Algorithms

Our CPS-tree representation is illustrated in Figure 1. We first partition the suffix tree into smaller trees, which we will address them as "local" trees, in a top-down fashion so that each local tree fits into a logical block (the bounding boxes in Figure 1). The end node in a local tree is either a leaf node (terminating circular node) or an external node (rectangular node with an outgoing dashed edge) pointing to the descending local tree in another logical block. Each local tree, rooted at a node v, is constructed by first adding the node v and its children, then the node with the heaviest subtree (largest number of leaf nodes), among all nodes at the local tree boundary, and its children are added. The process repeats until the local tree is full (that is, too big to fit into a logical block). This partitioning method guarantees a good IO disk access bound for both worst and average cases.

There are several tree partitioning methods in the literature. In the paper by Diwan *et. al.* [2], bottom-up, tree partitioning methods have been proposed, that find the optimal layout minimizing either the worst (maximum) or average block access when traversing from the root to any leaf in the tree. Another common approach is to build the partitions naively by grouping the nodes in the breadth-first order. This gives a good average performance in general. However, our partitioning scheme is good for both worst and average case. The average and worst case block accesses are $O(H/B)$ and $O(H/B + \log n)$ respectively (the proof is skipped). We did a comparison of the breadth-first order partitioning and our approach for CPS-tree, and the results show that our proposed approach achieves fewer page faults in both average and worst cases in exact string matching.

To compact the local tree, for each edge, we store the first character together with its label length (only for non-leaf edges for compactness). For each external node u in a local tree, we find in the subtree rooted at u (considering the whole suffix tree), a leaf node whose text position is to be replicated in u. The leaf node is choosen such that every node on the path from u to the selected leaf node, has the heaviest subtree (largest number of leaf nodes), among its siblings. Also, at the root v of each local tree, we store the SA range of the subtree rooted at v (see Figure 1 for an example).

To facilitate searching of nodes further down the tree, we maintain extra link, denoted as "forward link", at the block-level (in addition to the CPS-tree structure presented in Figure 1). We can then access any node from the root, by traversing through, in the worst case $O(\log n)$ logical blocks. This property is useful for applications that demand worst case guarantee in query time. However, due to the limit of space, we will leave the discussion on the "forward link" to the full paper.

Based on our observations, the top few levels of nodes in the suffix tree are most frequently visited in answering queries. As such, CPS-tree is written to disk in a top-down order. The order to be written is illustrated in Figure 1 as the label on the top left corner of each bounding box. Memory buffer is implemented to handle the accessing of the suffix tree where the memory buffer can be initialized very quickly through sequential read of the first few pages of the suffix tree from disk. Using an optimized bit-packing scheme to encode the individual tree structure, CPS-tree can further achieve good space utilization and IO efficiency in answering string matching query.

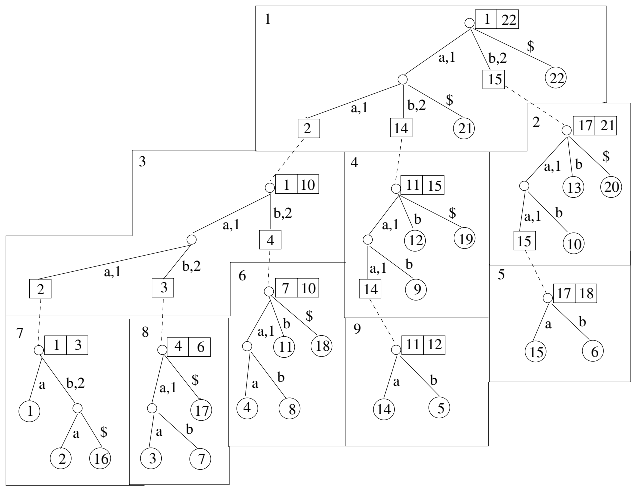


Figure 1. CPS-tree representation for text = "aaaaabaaabaababaaaaba$".

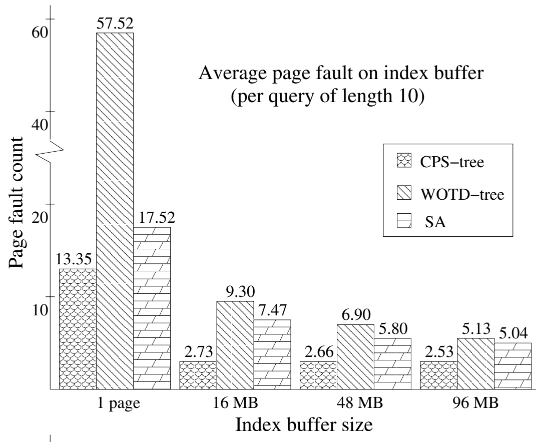
3. Performance Studies

We study the performance of reporting on the exact match locations in the text sequence, given a query string. CPS-tree is compared with WOTD-tree and SA for both on-disk and in-memory settings. The WOTD-tree is constructed in a top-down approach using the TDD package [9] available. We also perform string searching over the suffix array using binary search technique.

3.1. Experimental settings

The dataset used are the Fruit Fly genome of 118.3 million bases (http://www.fruitfly.org/sequence, Release 4) and the E. Coli K12 genome of 4.6 million bases (http://www.ncbi.nih.gov, GI: 49175990). These are DNA sequences consisting of characters 'A', 'C', 'G' and 'T'. The data and index are buffered separately. The index file size for CPS-tree on the Fruit Fly genome is $\approx 850M$ bytes $(7.2N)$ while WOTD-tree takes $\approx 1475M$ bytes $(12.5N)$.

The buffers are first initialized fully with the first few blocks read from the text sequence and index files respectively. Initialization of the buffers can be performed very quickly with sequential read from the files. If a block to access is not in the buffer, a page fault occurs and a new page of $8K$ bytes containing the required block is fetched into the buffer, using the First-In-First-Out buffer replacement policy.



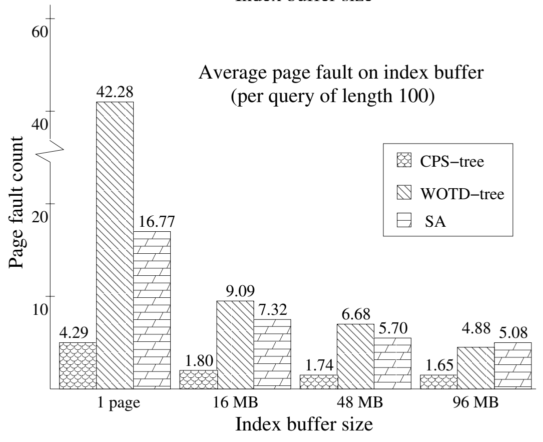


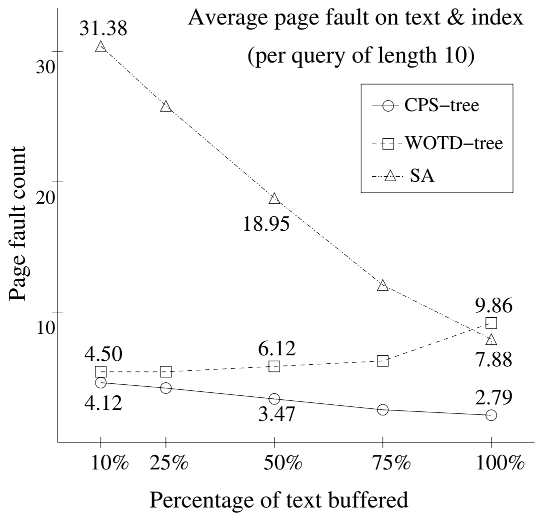
Figure 2. Result 1 - Page faults on index buffer for Fruit Fly genome.

Queries are generated from random positions on the genome itself so that it is guaranteed to return a match in the indexing structure. The average performance is measured from running consecutively, 1000 different random queries of the same length.

The experiments are carried out on an Intel P4 2.4GHz machine with 512KB cache and 1GB of RAM, running Linux, with codes written in C++. We implemented the search algorithms for CPS-tree, WOTD-tree and SA, so that they all share the same access routines to the buffers.

3.2. Performance results

Result 1. First, we examine the IO efficiency in traversing CPS-tree index structure. We use the Fruit Fly genome in this comparison with the main portion of the index structure residing on disk. The page faults are generated from traversing the indexes and also from reporting on the occurrences. We observed that CPS-tree consistently outperforms the other indexes for different index buffer size. The average *occ* per query found are 388.6 and 1.6 for query length 10 and 100 respectively. Query of length 10 generates more page faults than those of length 100, mainly from reporting on the occurrences. Our finding shows that a careful organization of the nodes into blocks does significantly improve upon the search performance.



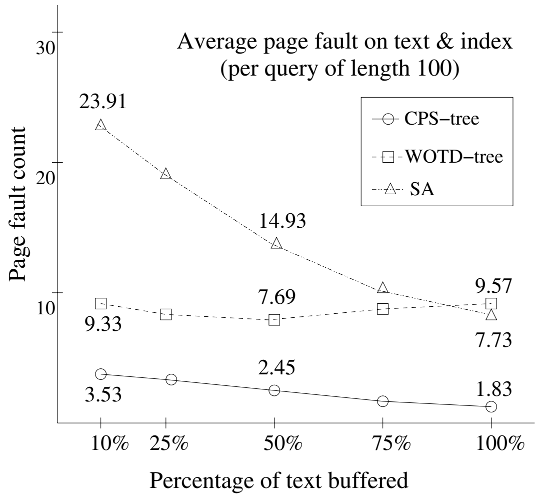


Figure 3. Result 2 - Page faults on text and index buffers for Fruit Fly genome.

Result 2. Next, we look at the buffer size allocation between the text and the index. Given a total of 128MB for buffering, we varied the text buffer size as a percentage of the text size, and use the remaining space available to buffer the index. Results in Figure 3 show that CPS-tree has the best IO performance, generating significant less page faults when compared to the other 2 indexes. Also CPS-tree and

SA work best with full text buffering (in memory) while WOTD-tree gives mixed results depending on the query length.

Result 3. We now give the in-memory (for both index and text) running time of the different indexes in exact match as shown in Table 3. This is performed on the E. Coli genome (4.6M). When compared to WOTD-tree, CPS-tree is much faster, showing that CPS-tree has a better representation scheme for suffix tree. CPS-tree is equally fast when compared to SA for short queries with CPS-tree gaining faster performance as the query gets longer.

Query	Per query	Query time (μsec) per query		
length	*occ* count	CPS-tree	WOTD-tree	SA
10	9.960	21	52	22
100	1.078	16	39	18

Table 3. Result 3 - In-memory (exact match) query timing on E. Coli genome.

Query	k = 1 (per query)		k = 2 (per query)	
length	*occ* count	paging	*occ* count	paging
10	7369	58	79346	684
100	1.68	37	1.72	460

Table 4. Result 4 - k-mismatch query on Fruit Fly genome.

Result 4. We run k-mismatch query on CPS-tree to show its capability in handling complex query. The results are shown in Table 4. The k-mismatch query finds all occurrences of the query, located on the text with Hamming distance $\leq k$. The number of occurrences increases sharply for larger k, especially for short queries. Using 200MB for buffering, the running time is around 0.1 to 0.2 sec per query for $k = 1$ and 0.4 to 2 sec when extended to 2 mismatches. For $k = 1$, a total of 31 and 3001 substituted query patterns are searched, for query length 10 and 100 respectively. That gives an average page fault of 1.87 and 0.01 per substituted query pattern. This is faster than searching the individual patterns directly as there is saving in the page access through our search approach. For long query of length 100, not many substituted patterns can find a match and hence resulting in early termination of the search and faster running time. It is to note that edit distance can be handled, by adopting the standard dynamic programming technique over the suffix tree.

4. Conclusion

CPS-tree is optimized to support tree traversal and enumeration of occurrences efficiently. Our experiments show that CPS-tree outperforms other compact representation of suffix tree in both in-memory and on-disk scenarios. CPS-tree performs better than suffix array on disk and is equally fast (or slightly faster) when runs fully in memory.

References

[1] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *ACM-SIAM SODA*, pages 383–391, 1996.

[2] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of VLDB*, pages 343–353, 1996.

[3] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *ACM-SIAM SODA*, pages 373–382, 1996.

[4] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[5] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, pages 30–42, 1999.

[6] S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 13:1149–1171, 1999.

[7] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proceedings of the International Computing and Combinatics Conference*, pages 401–410, 2002.

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

[9] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *Proceedings of VLDB*, pages 36–47, 2004. http://www.eecs.umich.edu/tdd/index.html.