Suffix Trees for Very Large Genomic Sequences

Marina Barsky
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
mgbarsky@cs.uvic.ca

Ulrike Stege
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
stege@cs.uvic.ca

Alex Thomo
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
thomo@cs.uvic.ca

Chris Upton
Dept. of Biochemistry & Microbiology
University of Victoria
BC, V8W 3P6, Canada
cupton@uvic.ca

ABSTRACT

A suffix tree is a fundamental data structure for string searching algorithms. Unfortunately, when it comes to the use of suffix trees in real-life applications, the current methods for constructing suffix trees do not scale for large inputs. All the existing practical algorithms perform random access to the input string, thus requiring that the input be small enough to be kept in main memory.

We are the first to present an algorithm which is able to construct suffix trees for input sequences significantly larger than the size of the available main memory. As a proof of concept, we show that our method allows to build the suffix tree for 12GB of *real* DNA sequences in 26 hours on a single machine with 2GB of RAM. This input is *four times* the size of the Human Genome, and the construction of suffix trees for inputs of such magnitude was never reported before.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Algorithms, Design, Performance

Keywords

External memory algorithms, Suffix tree

1. INTRODUCTION

If we want to use the information from genetic databases to its full potential, we need to design more efficient techniques to support user-defined queries for this data. One of the promising directions is the preprocessing of entire collections of genomic data into indexing structures. In this paper we consider suffix trees, the full-text indexes which are *crucial* for many applications on genome data (cf. [4]).

As suffix trees are significantly larger than their input sequences and quickly outgrow the main memory, constructing suffix trees for large inputs calls for disk-based methods.

The scalability of all previously proposed methods does not go beyond inputs that fit into main memory.

The attempts to overcome this inpit string bottleneck were undertaken in [10, 8, 1], and they clearly indicate that the problem is far from being solved.

In [10] the authors proposed the *Suffix Tree Merge (ST-Merge)* algorithm, which was expected to have better locality of references in the access to the input string than their original *TDD* algorithm. However, the experimental evaluation reported in [10] has shown that the *ST-merge* algorithm runs an infeasible amount of time for moderate input sizes. For example, the construction of the suffix tree for an input of size 20MB using 6MB of main memory (allocated for the input string buffer) took about 8 hours. The performance for larger inputs was not reported. Since the improvement over the original *TDD* algorithm was insignificant,[1] in our comparative experiments we use *TDD* instead of *ST-merge*.

Interesting original ideas were proposed by the authors of the *Trellis* algorithm in [8] where they developed a new version of the algorithm – *Trellis with String Buffer* (*Trellis+SB*). During the merge, when the random access to the entire input is requred, some parts of the input string are kept in the main memory. *Trellis+SB* replaces, whenever possible, the substring positions labeling tree edges by positions in one small representative partition. This small representative part of the input is kept in memory during each merge and increases the buffer hit rate. Another technique used by *Trellis+SB* is the buffering of some initial characters for each leaf node. The combination of these techniques allowed in practice to reduce the number of accesses to the on-disk input string by 95%. Note that the remaining 5%, for example, for an input of 10GB correspond to 500 million of random disk I/Os. The authors report that they were able to build the suffix tree for 3GB of the Human genome, using 512MB of main memory, in 11 hours on an Apple Power

[1]The implementation of *ST-Merge* is not available [8].

CIKM'09, November 2–6, 2009, Hong Kong, China.

Mac G5 with a 2.7GHz processor and 4GB of total RAM. The performance for larger inputs was not reported.

Similar results in decreasing the number of random disk I/Os were obtained for the *DiGeST* algorithm [1], which merges suffix arrays built for input partitions, using a multi-way merge sort and organizing the output buffer in form of a suffix tree. In order to reduce the access to the input string in the merge phase, for each position in the suffix arrays of partitions a 32–character prefix was attached. This prefix served the comparison of suffixes of different partitions in the merge phase of the algorithm. The references to the input string were reduced by 98% for the DNA data used in the experiments in [1]. Even 2% of remaining random accesses significantly degraded the perfomance of *DiGeST* when the input string was kept on disk.

The main contribution of this paper is the first practical algorithm for constructing suffix trees for inputs larger than the size of main memory. We present B^2ST, an efficient external-memory suffix tree construction algorithm for very large inputs.[2] B^2ST minimizes random access to the input string and accesses the disk-based data structures sequentially. The algorithm scales to much larger inputs than the previous algorithms. It is able to build a disk-based suffix tree for virtually unlimited size of input strings, thus filling the ever growing gap between the increase of main memory in modern computers and the much faster increase in the size of genomic databases.

2. OUR ALGORITHM

Problem definition.

We consider a *string* X to be a sequence of N symbols. The first $N-1$ symbols are over a finite alphabet Σ, $x_i \in \Sigma$ $(0 \leq i < N-1)$. The last symbol x_{N-1} is unique and not in Σ (a so-called *sentinel*). By $S_i = X[i, N]$ we denote a *suffix* of X beginning at position i, $0 \leq i < N$. Note that we can uniquely identify each suffix by its starting position.

Prefix P_i is a substring $[0, i]$ of X. The *longest common prefix* LCP_{ij} of two suffixes S_i and S_j is a substring $X[i, i+k]$ such that $X[i, i+k] = X[j, j+k]$, and $X[i, i+k+1] \neq X[j, j+k+1]$.

If we sort all the suffixes in lexicographical order, and record this order into an array of integers, then we obtain the *suffix array* SA of X. SA holds all integers i in the range $[0, N]$, where i represents S_i. The suffix array can be augmented with the information about the longest common prefixes for each pair of suffixes represented as consecutive numbers in SA.

A *suffix tree* [6] is a digital tree of symbols for the suffixes of X, where edges are labeled with the start and end positions in X of the substrings they represent. Note also that each internal node in the suffix tree represents an end of the longest common prefix for some pair of suffixes. The tree's total space is linear in N in the case that each edge label can be stored in a constant space. Fortunately, this is the case for an implicit representation of substrings by their positions.

In this article, we discuss the problem of constructing suffix tree ST for string X of size N. Our challenge is that the main memory size, M, is smaller than the space needed to hold the entire input string; the *input-to-memory ratio* $r = N/M$ is at least 2. Therefore, neither X nor ST can be entirely loaded into the main memory and only some parts of them can be held in main memory buffers. The goal is to build ST minimizing *random* disk I/Os.

Our solution.

Our solution for the problem described above is based on the following ideas.

The suffix tree of X can be constructed given its suffix array SA augmented with an LCP information. As was proved in [2], the conversion of the suffix array into the suffix tree can be performed in a linear time. In practice, this process exhibits a good locality of references and therefore a good behavior in external memory settings. Both SA and ST can be kept on disk, and only their sequential parts can be loaded and manipulated in main memory.

Consequently, the first step in our algorithm for the suffix tree construction is obtaining a suffix array for string X, augmented with the LCP information. For this, we want to lexicographically sort all suffixes of X. Suffix sorting differs from conventional string sorting in that the elements to be sorted are N overlapping strings, and the length of each such string is $O(N)$. This implies that a comparison-based sorting algorithm, which requires $O(N \log N)$ comparisons, may take $O(N^2 \log N)$ time. Moreover, if we treat suffixes as if they were regular strings we have an even bigger problem: when comparing a pair of suffixes we need to scan the corresponding sequences of symbols in X starting at two positions along the string X. These positions are mostly non-consecutive. When X is on disk, this translates into a prohibitive number of random disk I/Os.

The second idea we use in the design of the new algorithm is the general paradigm of the external memory two-phase multi-way merge-sort *(2PMMS)* [3]. We partition X into substrings (partitions) and lexicographically sort the suffixes in each partition. We can do this in main memory by using any of the best algorithms for in-memory suffix sorting and writing the resulting suffix arrays to disk.

A problem arises when we want to merge these suffix arrays. In the simple case of merging sorted lists of keys, the relative order of elements from any two different lists is determined by comparing these elements. However, in our case, all we have are the starting positions of the suffixes from different partitions, since this is all the information we can store in suffix arrays. This does not help in determining the relative lexicographical order of suffixes, since our sorting keys are the substrings of X (and not their starting positions).

A naïve approach would compare two suffixes from different partitions by randomly accessing X, which is on disk. This would lead to $O(N)$ *random* disk I/Os. This takes a prohibitive amount of time even for small N. Be reminded that the size N of our input string is several times larger than the available main memory.

To avoid the access to the input string in the merge step, we do not compare the actual input string characters, but rather deduce the necessary information from the relative order and the LCP of any two suffixes stored in specific structures which we call *pairwise order arrays*.

B^2ST proceeds in three steps: input partitioning, sorting of suffixes for each pair of partitions and merging all suffixes into a disk-resident suffix tree.

[2] B^2ST stands for **B**ig string, **B**ig **S**uffix **T**ree

```
Algorithm pairwiseSorting
   input: k partitions of string X
1. for (u=0; u<k-1; i++)
2.    for (v=1; v<k; v++)
3.       concatenate X_u X_v and load into RAM
4.       build suffix array with LCP SA_uv
5.       during sequential scan of SA_uv
6.          if v==k-1  //last chunk
7.             output to disk SA_v
8.          output to disk SA_u
9.          output to disk R_uv          //order array
```

Figure 1: Algorithm for pairwise sorting of suffixes in all partition pairs.

Step 1: Input Partitioning.

Our algorithm first partitions the input string X of size N into k partitions, such that $k = 2r$ (recall that $r = N/M$ is the input to memory ratio). Note that the sequenced genomes are already partitioned into natural partitions – the chromosomes. In general, if one of the natural partitions is too large, it is partitioned into several artificial chunks, such that each chunk has length at most p. It is enough to append to the end of each such chunk, except the last one, a small "tail", namely the prefix of the next chunk. The tail of the chunk must never occur as a substring of this chunk. It serves as a sentinel for the suffixes of the chunk, and its positions are not included into the suffix array of the corresponding partition. Note that we require the combined size of any pair of partitions $2p$ to be less than M.

Step 2: Suffix sorting in partition pairs.

In this step we generate suffix arrays for each pair of partitions. The pseudocode for this step is shown in Figure 1. We concatenate every possible pair u, v of partitions ($0 \leq u < k-1$, $u+1 \leq v < k$, $u < v$) into string $X_u X_v$. We load this input into the main memory and build the suffix array SA_{uv} with attached LCP information for each suffix. From each SA_{uv}, we extract two structures: (1) the suffix array SA_u for partition X_u and (2) an "order array" R_{uv} of size $|X_u| + |X_v|$. The *order array* R_{uv} contains the LCP entries of SA_{uv} plus the partition ID information, which can be stored in 1 bit. Specifically, we use 0 for u and 1 for v ($u < v$).

At the end of this step we have on disk k suffix arrays for k partitions (of total size N), plus $k(k-1)/2$ order arrays for each possible pair of partitions (of total size kN). This is all the information we need to efficiently perform the merge. As a result of this merge we produce the suffix tree for the entire input string X. We are doing this without loading the entire input string into main memory. In fact, we never access X anymore.

Step 3: Merging.

In order to merge the suffix arrays of different partitions, we use the information from the order arrays. Notably, all these arrays are accessed sequentially. As in the classical *2PMMS*, we have k input buffers for each of the k disk-based suffix arrays created in Step 2. We denote the buffer for a suffix array SA_u by SA_BUF_u. In addition, we use $k(k-1)/2$ input buffers for order arrays. We denote the buffer for an order array R_{uv} by R_BUF_{uv}. Finally, we have an output buffer, ST_BUF, where we collect the nodes of the merged suffix tree before emptying it to disk. The total size of all the buffers matches the size of the available main memory.

```
Algorithm compareSuffix (S_i from partition u,
                         S_j from partition v)
1.  if (u = = v)
2.     return -1          //S_i <_lex S_j, since they are sorted
                          //in increasing order inside each partition
3.  if (u < v)
4.     if (partitionBit in R_buf_uv[current pointer] = = 0)
5.        return -1                  //S_i <_lex S_j
6.     else
7.        return 1                   //S_i >_lex S_j
8.  if (u > v)
9.     if (partitionBit in R_buf_vu[current pointer] = = 0)
10.       return 1                   //S_j <_lex S_i
11.    else
12.       return -1                  //S_j >_lex S_i
```

Figure 2: Algorithm for suffix comparison which uses the pairwise suffix information from the order arrays created during pairwise suffix sorting.

In essence, the merge corresponds to the merge phase of the multi-way merge sort. The only difference is that in order to compare suffixes from different partitions we use the order information recorded in order arrays, as shown in pseudocode for suffix comparison in Figure 2.

Note, that the disk-resident suffix arrays and the order arrays are read sequentially, which would not be the case if we were consulting the input string X to resolve a relative order for arbitrary suffix start positions of different partitions.

In the output buffer, ST_BUF, we incrementally build the suffix tree for lexicographically sorted suffixes.

Knowing the LCP of the current suffix with the suffix previously inserted into the tree, we simulate an Euler tour on the tree under construction [2], such adding one internal node and one leaf per each suffix.

Before flushing the output buffer to disk, as done in [1], we add the 32-character prefix of the suffix last inserted into the tree to a collection of *dividers*. Each divider contains the prefix of the lexicographically largest suffix in the corresponding suffix tree along with a pointer to the file where the tree is stored. Thus, at the end of the computation we have on disk a forest of suffix trees, each of which can quickly be located from a corresponding divider, and then loaded and queried in main memory. Notably, all these small trees are of equal length, which solves the problem of data skew for prefix-based partitioning of the on-disk suffix tree, as in [5, 9].

Finally, we note that the collection of dividers is small and can be kept in main memory during the query execution. For example, in order to locate a pattern in X, we first scan the collection of dividers to find the proper tree, then we load this tree into main memory and search inside it.

Analysis.

Since the suffix array construction and the LCP computation for each pair of partitions can be done in time linear in its length $2N/k$, and we have $k(k-1)/2$ different pair combinations, the running time of the sorting step is $O(kN)$, where $k = 2N/M$. In other words, the time is proportional to the total input size and the input-to-memory ratio, i.e.,

Program	Time, hours
TDD	125
Trellis+SB	11
B²ST	3

Figure 3: Running times of different suffix tree construction algorithms for approximately 3GB of DNA sequence (human genome) which is larger than the total allocated main memory.

how many times our input exceeds the available main memory. In the merge step, the suffix tree of size $O(N)$ is constructed in time linear in N. This requires, however, the complete scan of the intermediate order arrays of total size $O(kN)$. Thus, B^2ST runs in asymptotic time $O(kN)$, using the constant amount of the available main memory M and $O(kN)$ temporary *disk* space. The high efficiency of the algorithm is due to the sequential access of the disk data.

3. EXPERIMENTAL EVALUATION

We implemented B^2ST in C and compiled with a GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel Core Duo 2.66 Ghz CPU, 2GB RAM and 4MB L2 cache under Ubuntu 7.04, 32-bit Linux.

Our objective was to evaluate the main idea of B^2ST, assuming the required suffix arrays for each pair of partitions are given. For this purpose we used the modification of *DiGeST* [1] which outputs a suffix array with LCP values instead of suffix trees. Thus, *DiGeST* efficiently builds suffix arrays for 4GB of DNA using a main memory string buffer of 1GB assuming the DNA string is first compressed using 2 bits per character.

We first evaluated the performance of our algorithm in comparison with *TDD* [12] and *Trellis+SB* [13] implementations. The results are shown in Figure 3.

For B^2ST we divided the 3GB into partitions of 1GB each and built the suffix array for partition pairs of a total size of 2GB. We used for this 600 MB of main memory.

The sorting of suffixes in the 3 pairs of 3 partitions took 118 minutes, while the merge took only 13 minutes. This confirms that performing sequential scans as we do in B^2ST pays off compared to just focusing on reducing the number of random disk I/O's to the input string, as the other algorithms do.

Next we evaluated the scalability of our algorithm. Using only 1GB of main memory to hold input string we constructed suffix trees for approximately 6, 8, 10 and 12GB of genomic data. These datasets were generated from combinations of sequences of eucaryotic chromosomes (human, chimpanzee, zebra fish, cow, mouse, and chicken) obtained from [11].

The performance results are shown in Figure 4. The size of each partition is 2GB. For our largest input, 12GB, we had 6 partitions and 15 partition pairs. The time taken to build the suffix arrays of these 15 pairs was 25 hours and produced an intermediate on-disk output of size 234GB. Despite this, the merge phase completed in only 59 minutes, scanning all this on-disk data in sequential manner and produced 2514 suffix tree files of total 215GB.

This example shows that we need a large temporary disk space for scaling up the B^2ST algorithm. Specifically, we need $D = k^2p = kN$ disk space to store the order arrays for all partition pairs. Since the number of partitions is

Input size, GB	Number of partitions	Number of partition pairs	Pairwise sorting, min	Merge, min	Total time, min
6	3	3	441	27	468
8	4	6	730	34	764
10	5	10	1100	42	1142
12	6	15	1462	59	1521

Figure 4: Running time (min) of B^2ST for different sets (approximately 6, 8, 10 and 12GB) of genomic DNA using only 2GB of main memory.

$k = N/M$, from $D = N^2/M$ we can determine the size of the largest input that we can process with M bytes of internal memory and D bytes of disk space. If we substitute the common values for modern computers, $D = 10^{12}$ (1TB), and $M = 4 \times 10^9$ (4GB), then we can build suffix trees using such a machine for up to 60GB of input. Note, however, that the construction of suffix trees even for 10GB of input was never achieved and reported before.

Hence, by using our B^2ST algorithm, indexing a large amount of DNA data with suffix trees becomes a feasible routine task.

4. REFERENCES

[1] M. Barsky, U. Stege, A. Thomo, and C. Upton. A new method for indexing genomes using on-disk suffix trees. *Proceedings of CIKM 2008*: 649–658, 2008.

[2] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6): 987–1011, 2000.

[3] H. Garcia-Molina, J. D. Ullman, J. D. Widom. Database System Implementation. Prentice-Hall Inc., 1999.

[4] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[5] E. Hunt, M.P. Atkinson, R.W. Irving. A database index to large biological sequences. *The VLDB Jornal*, 7(3): 139–148, 2001.

[6] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2): 262–272, 1976.

[7] B. Phoophakdee and M. J. Zaki. Genome-scale Disk-based Suffix Tree Indexing. *ACM SIGMOD International Conference on Management of Data*, 2007.

[8] B. Phoophakdee and M. J. Zaki. Trellis+: An Effective Approach for Indexing Massive Sequence. *Pacific Symposium on Biocomputing*, 2008.

[9] S. Tata, R.A. Hankins, J.M. Patel. Practical suffix tree construction. *Proceedings of 30th VLDB conference*, 36–47, 2004

[10] Y. Tian, S. Tata, R. Hankins, J. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3) : 281–299, 2005.

[11] USCS Genome Browser: *hgdownload.cse.ucsc.edu/downloads.html*

[12] Source code for TDD: *www.eecs.umich.edutddownload.html*

[13] Source code for Trellis+SB: *www.cs.rpi.edu/∼zaki/software/trellis*