

A Space Efficient Persistent Implementation of an Index for DNA Sequences

Gabriele Witterstein ¹
Technical University of Munich

Abstract:

Due to newly developed high-throughput technologies for DNA sequencing, the number of fully sequenced species increases rapidly. String databases holding these sequences are very large. On the field of molecular biology the handling of large string data which cannot be broken in words is a great challenge. Hereby the most important string operation is the approximate substring match. This type of match is essential for many applications in biology.

The suffix tree has been established as the most predestined in-memory data structure supporting approximate substring matches on DNA sequences. They are belonging to the wider class of suffix structures.

The central issue of the paper is the theoretically evaluation of the suffix structures with the aim to reveal the most suitable structure for a persistent implementation on the disk. I will show that this is a variant of a suffix tree. Further, the paper addresses the question in which way this data structure can be stored on disk, and how fast access can be achieved. In this connection I want to introduce a space efficient implementation by using a tree coding scheme which optimizes disk saving and therefore preventing unneeded disk access. Here the most important design issue lies in the representation of different variants of nodes of the tree.

I will present an implementation in C programming language and show that the implementation in a first evaluation step compares favourably to other implementations.

Keywords: DNA index structure, comparative genomics, suffix tree, databases

I. Introduction

In the field of genome research very large strings appear. For example, the human genome contains roughly 3 GB consisting of 22 words, each in size of about 220 MB. As a result of newly developed high-throughput technologies for DNA sequencing, the number of fully sequenced species increase. The statistic shows that the size of databases holding these sequences, like GenBank, has doubled every 15 months. At present, GenBank has the

¹Department of Applied Mathematics, Technical University of Munich, Boltzmannstr. 3, 85747, Garching b. Munich, Germany, gw@appl-math.tu-muenchen.de, (+49)89 289 16832

size of approximately 18 GB. Here, an index structure supporting fast processing of very large strings, is needed. With progressive time, this demand will be more urgent, because the size of databases will dramatically grow. In bioinformatics, the suffix tree has been established as data structure allowing many different kinds of search operations needed in this application field (see [11], [4]). Hereby, the most important search is the approximate substring search, used at different opportunities in the analyses of biologists.

I want to shortly demonstrate this by an introductory example from functional genomics. In a typical functional genomic analysis, one may start with an expressed sequence tag (EST) with the aim to find out the presumably function of the corresponding gene. In a first step a request for a specialized genome database has to be formulated to retrieve the corresponding gene. Here the nucleotide sequence is used to trigger a BLASTX query into a protein database getting homologous protein sequences. For this, the proteins are exported from the database.

Generally this means, in functional genomics one tries to reveal similar pieces of the DNA sequence from different species. Very popular tools such BLAST [1, 2], FASTA and newer BLAT [7] are mostly used. Such tools run outside of the database engine where the sequences are stored. And all these widely used methods base on heuristic filter algorithms and work without indexes, but flat files. Their disadvantages lie in the unexact computation. And today's algorithms have not the necessary performance for all biological tasks, such as for all-versus-all queries. If one would send to the BLAST server 90 queries, then the processing time would be more than 70 hours. Another point is, that one always has to export the sequences, because the algorithms run outside the database engine. This approach is undesirable in general. Algorithms should be executed near the data.

In order to performe the approximate substring match very fast, one can use filter algorithms. At the core, filter algorithms are based on exact searches. Hereby, the great advantage of a suffix tree against other suffix structures lies in the following: With a suffix tree the exact search of substrings depends in complexity only on the length of the query string and its appearances. Definitly, it doesn't depend on the length of the whole coded text as it is the case, for example, for the suffix array.

The managing of string data appears in various application fields. String data is ubiquitous. There are two kinds of string data holding in databases. Firstly, string data are be held in tables as values of attributes. They are very short. One task could be the approximate joining of columns. Secondly, string data could be very long, for example literature texts or web documents. In databases they are stored as data type CLOB. The predestinated approximate match in most cases is a substring match. There are two classes of such texts, texts which can be broken in words, and which cannot. The choice of the most suitable index structure depends of such criterias.

Searching in unstructured text, like DNA, adds a special difficulty to indexing. The set of candidate keys is much larger than for structured texts, because each suffix is a candidate key. The suffix tree belongs to the wide class of suffix structures which is the most suitable data structure on the top of a text which cannot be broken in words. This special case appears in a DNA sequence databases. This situation also appears in video databases or in large texts for spell-checking, regarding human typing errors. DNA sequences are large and regarded as static. The construction of a persistent suffix tree is needed and a storage on disk in a manner that makes fast access possible.

Aim of the Paper

This paper deals with the question, in which way one can store the tree on disk without loosing fast access. By this, I describe the logical data structure of a variant of a suffix tree, most suitable for the persistent implementation.

For the physical implementation, I introduce a coding scheme of the tree that's disk saving and enabling fast access by preventing unneeded disk access.

Paper Overview

The paper is organized as follows. Section II summarizes previous work on the field. There is an introduction in suffix structures and which of those have been implemented persistently at present. Section III.A introduces my variant of a suffix tree and section III.B presents how this variant can efficiently be stored on disk, supporting exact matching. In section IV, I discuss how this data structure can be stored and present first results of a performance study.

II. Related Work

II.A. Related Work - Logical Suffix Structures

The predestinated persistent data structure for string searching is the suffix array. A suffix array is just a lexicographically ordered array of all suffixes of the text. These suffixes are represented by their starting positions. That means, a suffix array stores only the pointers of each substring from the text within an array. Therefore, the total space requirement is $4 * n$, if n is the length of the text [8].

In this array, string location is performed by making a binary search. However, the worst case search time is very high. The search complexity of a binary search is large and depends logarithmically on the length of the indexed text. If the indexed text is very large, this fact is a main drawback. For the most kinds of texts, practical investigations have shown that the average search time is comparable with the search time of a suffix tree. But this is not the case by DNA sequences.

Based on this, great efforts have been made in order to further reduce the theoretical search complexity. Here, the one-dimensional structure of the suffix array has been augmented. This approach is called augmented suffix arrays. For augmented suffix arrays, there are many variants, for example suffix cactus [6], suffix vector [9] or the augmented suffix array from Manber and Myers [8]. All these kinds of suffix arrays deals with the introduction of additional array dimensions standing for lcp (longest common prefix) values or skip factors. (These numbers indicate how many characters at the beginning of each string can be skipped during comparisons.) These additional numbers should further reduce the complexity of the search algorithm. Thereby, it's possible to reach a better complexity for the worst case search than for a plain suffix array.

Whereas, a suffix tree is a compressed digital trie. It is built by joining each non-branching node with its child. Here, each edge is labeled by substrings of the text. Thus, it can be represented by two references into the text.

That is to say, the most detailed logical data structure is the suffix tree. The most sporadic the suffix array. Each suffix tree can be mapped to a augmented suffix array. That is, one can augment a suffix array as far as a suffix tree arises, or the developed data structure equals a suffix tree. Because of the unbalanced and adaptive tree structure, this implementation includes many blank positions.

All endeavors alike, the primary text must be accessed by each search operation. For texts stored on secondary memory, this is very time expensive. In all cases, DNA strings with a size of about 280 MB per chromosome (this can be regarded as word) or circa 3,3 GB per genome have to be kept on the disk. Here, suffix structures which work for coding of the suffix strings only with references are inappropriate. Regarding the search performance in a persistent index it should be exposed as better, to code the letters of the suffix in the index itself. Also, data structures working with backtracking are unsuitable in the observed case.

That is to say, one have to find an optimal data structure in relation to both, its theoretical search complexity, and the possibilities of its practical implementation. In other words, one can describe a data structure by its logical structure and its physical representation. But a closer look shows, that these two aspects intertwine.

II.B. Related Work - The Possibilities of the Persistent Implementation

There is a large literature about transient suffix trees. But even, the most space efficient implementation uses 13 bytes per indexed letter. Therefore, as in [5] is pointed out, a transient suffix tree for very large strings cannot be created in-memory and cannot be held there. A construction algorithm of a persistent suffix tree is needed as well as storing an efficient mechanisms.

A partitioning algorithm has been introduced in [5], where a large tree is divided in partitions, each of them created in-memory and then written on disk. Hereby PJama is used, which makes the in-memory structure automatically persistent. In [5] it has been mentioned that a space efficient implementation is needed.

In [12] this partitioning algorithm has been improved by introducing variable length of the prefixes determining the partitions, and therewith preventing the failure of the algorithm in [5]. This approach is adapted to the blocksize of the disk, and avoids the creation of many low occupied partitions. Among other things these small partitions support the big size of the index on the disk.

III. My Approach

III.A. The Logical Tree Structure

Clearly, if one has the aim to use an index structure lying on disk, one has to minimize the disk access. Therefore a structure like a suffix array, which depends in its search complexity on the length of the indexed string, is not appropriat. The same holds for structures working with backtracking such as Patricia tries, unless one is able to controll the backtracking behaviour. Therefore, a data structure supporting the most straight forward search is needed. For example for a suffix tree, this is guaranted. This tree permits string search like

1	2	3	4	5	6	7
A	G	A	A	G	G	\$

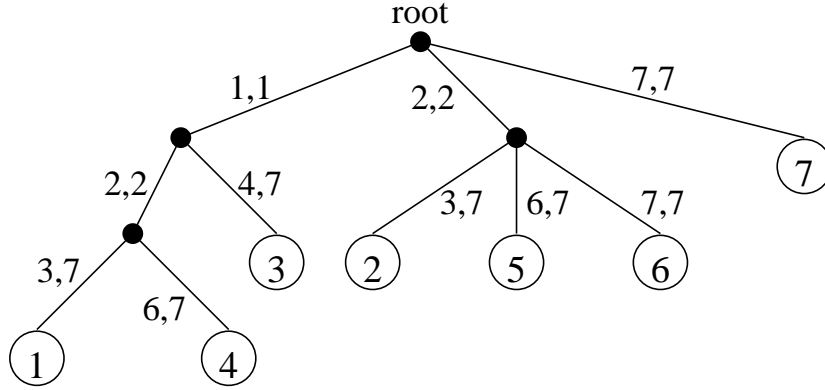


Figure 1: Normal suffix tree

1	2	3	4	5	6	7
A	G	A	A	G	G	\$

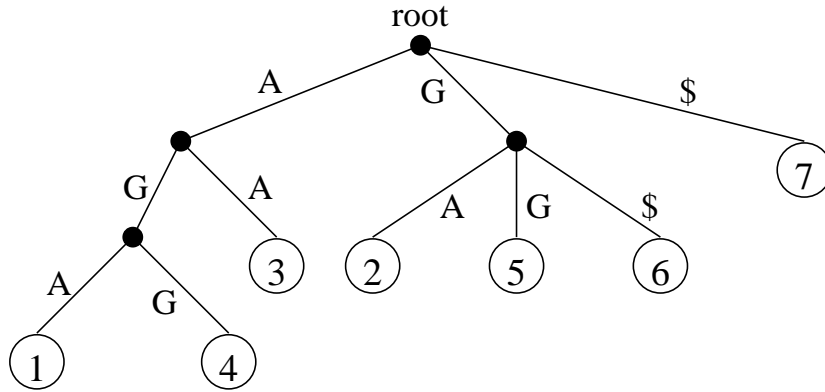


Figure 2: My variant of a suffix tree

"If s a substring of the text T ?" to be answered in $O(|s|)$ time.

Therefore, on this field, a variant of suffix tree is the most applicative for a persistent index.

A *normal suffix tree* is a compressed digital trie. It is built by joining each non-branching node with its child. An example is shown in figure 1. Here, each edge is labeled by substrings of the text. Thus, it can be represented by two references into the text. Therefore the space requirement for each edge is constant. And the size of the suffix tree depends linearly on the length of the text.

In order to find the proper child node during a search, one has to jump to the DNA file, to seek the position and to match the characters. Usually the DNA letters are not coded on the edges of the tree. This causes a bottleneck which lies in the random access to the text being indexed. To avoid this its useful to code the letters of the suffixes in the index itself. Algorithms, which traverse repeatedly the tree from bottom to top and reverse, profit from

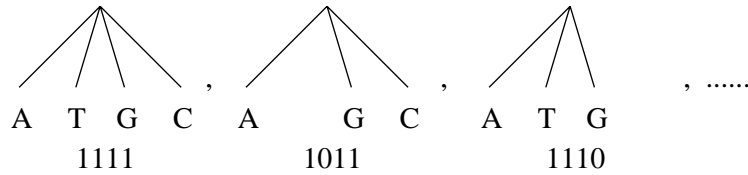


Figure 3: Treecode

this advantage.

But for an exact match only that prefix of the suffix is needed which characterizes the position of the suffix uniquely. Therefore, I store only such parts of the suffix, which are defined by its starting position and continue to the right as far as necessary to make the string unique. In order not to claim needless disk space, it is traced back, in the case of uniqueness, to the primary sequence file. Therefore, the tree I introduced is more a compacted suffix trie ending with leaves making the suffix unique. Theoretically the transition from a trie to a tree reduces the storage requirements from $O(n^2)$ to $O(n)$. But in my case, since the alphabet size is only 4, one can code each letter with 2 bits. Therefore in practice, my variant shouldn't constitute a considerably loss of storage space. In Figure 2 one can see my variant of a suffix tree.

III.B. The Physical Representation - The Tree Coding Structure

Although, since one stores the partitions of the tree on disk, in the long term a space efficient implementation is needed. Hereby, data structure engineering is necessary. As example, one wants to index two complete genomes, the human genome (3 GB) and the mouse genome (3 GB).

The small size of alphabet enables an efficient coding both the tree topology and the letters of the DNA. In order to code the local tree topology one can use a lookup table. Since one knows, that the size of the used alphabet is 4, every node of the tree can have at most 4 child nodes. First one should fix a preferred order of the letter, for example the lexicographical order or anything else. Then one can code each branching configuration of the tree in a manner demonstrated in figure 3.

Because one has specified an exclusive order of the letters, one can reconstruct the letters from the treecode. Note that in this way only maximal 4 bits for each node is necessary in order to code the branching node as well as the branching letter. Now the treecode is stored linearly in an array traversing the tree from left to right. In this way, the array represents the entire tree. The leaves can be expressed by 0000. In that way one has stored the local tree topology.

In my application, it is necessary to be able to traverse the tree selectively, i.e. to extract a subtree of the suffix tree. Since the edges are arranged in one large array, one needs to know the number of edges which are skipped in order to find the right branching node. One can solve this problem by additional storage of relative branch pointers. For each node in the tree these pointers are exactly the number of nodes in the subtree below. Thereby the expensive storage of absolute pointers is avoided, which usually requires more memory than the data itself. But also, some of the relative pointers can get quite large, i.e. the topmost node will contain number maximal 4^n . But, for the most nodes the numbers are small, e.g.

Table 1: Data type of variable sized record structure

Node Mode	Treecode	Pointer Number	Pointer Size	...	Pointer Size	
2 bits	4 bits	2 bits	2 bits	...	2 bits	
		Skip Pointer	...		Skip Pointer	
		variable sized	...		variable sized	

on finest level they are 2 and on the second finest level they are at most 4.

Therefore it is useful to allocate variable sized memory for the pointers and for the number of pointers. I use a bit code for the allocated size. Here one can, for example, use 2 bits for the size, i.e. 00 for 1 byte, 01 for 2 bytes, 10 for 4 bytes and 11 for 8 bytes. Of course, since we use variable sizes, the branch pointer is now not the number of nodes in the subtree, but the number of bytes allocated for all the nodes in the subtree. It can be computed in the top to down traversal of the tree. This way, the average storage requirement for a branch pointer is less than two bytes independent of the depth of the suffix tree.

Since each node has at most 4 subtrees it leads to a data type, shown in table 1.

By this way, in average for each node about 3 bytes are necessary. This scheme allows a very fast and efficient tree traversal, since only a few bit operations are necessary to find subsequent data values.

Taking the various structure of different branching nodes into account, one could introduce different node sorts, e.g., 00 specifies compacted records with only one child pointer.

In order to accommodate further the different structure of a suffix tree, one can introduce different types of records for different suffix lengths.

The node type indicates whether it concerns a record with sister nodes or a record, where several nodes with only one child can be summarized to one single node.

IV. Implementation and Experimental results

IV.A. Implementation

In principle, there are 4 different ways to make the index persistent.

One can use a relational DBMS, an object-oriented DBMS or persistent programming languages, such as persistent C++ or PJama. Another way is to create an own formatted flat file holding the data types of the index. That would be an own system adapted to use a minimum of disk space.

These various systems have their own advantages and disadvantages themselves.

Regarding the first approach, one can try to map the suffix tree structure into a relational database. Then, the index would be implemented on the top of the DBMS lying on the same level as the data itself. But, the relational database concept doesn't consider recursive

data structures. One would have to break down this tree structure on flat database tables and then could index these tables with an B+-tree index typical for relational databases. But this way doesn't reach a performance gain, since there is no proper support of that data structure. But, a relational DBMS has a sophisticated caching behaviour. Today's commercial relational database systems are in practice strongly optimized concerning their secondary storage management. This optimization could by far compensate the performance loss by mapping on flat tables. In addition, relational data bases are commonly used and so widely spread, that maintenance of the index for many users could be simpler than an own data format.

A commercial object-oriented DBMS comes to meet the tree structure more strongly, however has a poor caching behaviour. Since the object-header of the persistent objects would be substantially larger than it's content, it is further not clever to make from each node an object. The same is true for persistent object-oriented programming languages as C++ or PJama.

As described above, I have developed my own data format and have these written in index files on operating system level. All the code is written in the C programming language. I think, it isn't useful to program the index as main memory structure together with an automatical persistence mechanism. The index would be further enlarged.

For the whole procedure, there is no problem as in the main memory model one has small resources. But for a fast search, a minimal number of disk accesses are important. This can be achieved only by a space efficient representation of the suffix tree on the disk.

IV.B. Practical Results

All experiments were processed on a 1200 MHz AMD Athlon PC with 32 MB of memory and 256 MB RAM, running under the Linux operating system.. For my experiments I have used 33,4 MB DNA sequence. There are pieces of chromosome 22 of the human DNA. The construction time for this DNA text is about 4,5 hours. The really use storage space is nearly 838.000.000 B.

At present, I have examined tests about the exact match. These tests are reported in table 2 and 3. The tests show that this implementation is a magnitude faster than other results such as reported in [5].

V. Conclusion

This paper shows, how explicit persistent memory management can be performed for large suffix tree's. Altogether the question was answered, which suffix structure is the most applicable for persistent indexing. It would be taken runtimes for exact string matches, which compares favorably with other implementations.

Further works: The runtime of the construction algorithm have to be improved. We have the aim to index a whole genome of a specie. And a real evaluation of the application of the index in approximate string match algorithms must be undertaken.

Table 2: Cold Store Query Behaviour

size of the batch	query length in byte	average response time per query in ms	total hits
100	12	40	86.690
1.000	12	19	195.840
10.000	12	18,8	989.802
50.000	12	16,42	4.103.959
100	17	30	26.113
1.000	17	16	46.636
10.000	17	15,3	244.059
50.000	17	15,58	1.112.030
100	30	20	885
1.000	30	15	2.765
10.000	30	14,9	17.897
50.000	30	15,24	100.094
100	50	20	118
1.000	50	15	771
10.000	50	14,3	6.874
50.000	50	15,3	34.607
100	70	10	64
1.000	70	15	657
10.000	70	15,5	6.494
50.000	70	15,3	32.315

Table 3: Warm Store Query Behaviour

size of the batch	query length in byte	average response time per query in ms	total hits
100	12	0	86.690
1.000	12	0,5	195.840
5.000	12	0,4	543.521
7.000	12	0,29	671.647
100	17	0	26.113
1.000	17	0	46.636
5.000	17	0,1	146.656
7.000	17	0,14	1.112.030
100	30	0	885
1.000	30	0	2.765
5.000	30	0,1	11.800
7.000	30	0,14	14.247

References

- [1] S.F. Altschul et al, *Basic local alignment search tool.* J.Mol.Biology, 215:403-10,1990.
- [2] S.F. Altschul, T.L. Madden, A.A. Schaeffer, J. Zhang, W. Miller, D.J. Lipman, *Gapped BLAST and PSI-BLAST: a new generation of protein search programs.* Nucleic Acid Research, 25:3389-3402, 1997.
- [3] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, *Using q-grams in a DBMS for Approximate String Processing.* IEEE Data Engineering Bulletin, vol. 24, no. 4, December 2002.
- [4] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge: University Press, 1997.
- [5] E. Hunt, R.W. Irving, M.P. Atkinson, *A database index to large biological sequences.* Proceedings of the 27th International Conference on Very Large Databases, pp. 139-148, 2001.
- [6] J. Kärkkäinen, *Suffix cactus: A cross between suffix tree and suffix array.* In Proc. Sixth Symposium on Combinatorial Pattern Matching (CPM '95), (eds. Z. Galil and E. Ukkonen), LNCS 937, Springer, pp. 191-204, 1995.
- [7] J. Kent, *BLAT - The BLAST-Like Alignment Tool.* Genome Res. 12: 656-664, 2002.

- [8] U. Manber, G. Myers, *Suffix Arrays: A new method for on-line string searches.* SIAM J. Computation, 22(5): 935-948, Oct. 1993.
- [9] K. Monostori, A.B. Zaslavsky, I. Vajk, *Suffix Vector: A Space-Efficient Suffix Tree Representation.* ISAAC 2001: 707-718, 2001.
- [10] G. Navarro, *A Guided Tour to Approximate String Matching.* 2001.
- [11] L. Marsan, M.-F Sagot, *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification.* RECOMB 2000, pp. 210-219, 2000.
- [12] K.-B. Schürmann, B. Stoye, *Suffix Tree Construction for Large Strings.* In: Proceedings 14. Workshop of Fundamentals of Databases. Rostock, Germany, May 2002.
- [13] G. Witterstein, *A Space Efficient Implementation of a Persistent Suffix Tree.* RECOMB 2003 Poster Abstract, 2003.