A Space Efficient Persistent Implementation of an Index for DNA Sequences

Gabriele Witterstein

Technische Universität München
Boltzmannstr. 3
D-85747 Garching b. München
Germany
gw@appl-math.tu-muenchen.de

Abstract

In the field of molecular biology the handling of large string data is a great challenge. Hereby the most important string operation is the approximate substring match. Suffix trees have been established as the most successful in-memory data structure supporting approximate substring matches on DNA sequences. They belong to the wider class of suffix structures.
The central issue of this paper is the theoretical study of suffix structures with the aim to reveal the most suitable structure for a persistent implementation on disk. I show that this optimal structure 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 introduce a space efficient representation by using a tree coding scheme which optimizes disk saving and therefore prevents unneeded disk access.
I show, as a first evaluation step, that this implementation compares favourably to other implementations.

1 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 size of approximately 30 GB. To deal efficiently with such data, the idea is to develop index structures which are able to process very large string data. In particular, it is necessary to support approximate substring matches. Nowadays, in bioinformatics, this is the most fundamental operation which in future will become even more important.

The managing of string data in databases appears in various application fields. There are two kinds of such string data. Firstly, string data can 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, text or web documents as well as DNA sequences. In databases they are stored as data type CLOB. The predestinated approximate match in most cases is a substring match. One has to distinguish texts which can be broken in words, and those which cannot. The wide class of suffix structures is the most suitable data structure on the top of an unstructured text which cannot be broken in words. This special case appears in a DNA sequence databases. This situation also appears in video database or in large texts for spell-checking.

Let me give an introductory example from functional genomics in order to expresse the situation in bioinformatics. In a typical functional genomic analysis, one may start with an expressed sequence tag (EST) with the aim to find out the presumable 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.

That means, in functional genomics one tries to reveal similar pieces of a DNA sequence from different species. Very popular tools such as BLAST, FASTA and newer BLAT [6] are mostly used. These tools run outside the database engine where the sequences are stored. Moreover, these widely used methods are based on heuristic filter algorithms and work without indices. They just process flat files. Today's algorithms have not the necessary performance for all biological tasks, for example, not for all-versus-all queries. Further, if one sends to the BLAST server 90 queries, then the processing time can be more than 70 hours. Another point is, that for approximate substring matches one always has to export the sequences, because the above algorithms run outside the database engine. This approach is undesirable in general. The goal is to develop structures in which the algorithms could be executed near the data.

In order to performe the approximate substring match exact and very fast, I want to concentrate on filter algorithms. Filter algorithms are based on exact searches at the core. To obtain a high performance, I use index strucutres instead of flat files. For the index structure, I use a variant of suffix trees instead of other suffix strucures. Hereby, the great advantage lies in the following: With a suffix tree the complexity of an exact search of substrings depends only on the length of the query string and its appearances. Definitly, it does not depend on the length of the whole coded text as it is the case, for example, for the suffix array.

Due to its large space requirements, for several genomes or whole DNA sequence databases, a suffix tree cannot be created and held in main memory. The construction of a persistent suffix tree is needed and a storage on disk in a manner that makes fast access possible. To achieve this, I describe a logical data structure most suitable for a persistent implementation. This data structure is a variant of a suffix tree. For the physical implementation, I introduce a coding scheme of the tree which saves disk space and enables fast access by avoiding unnecessary disk access.

The paper is organized as follows. Section 2 summarizes previous work on the field. It contains an introduction to suffix structures and a discussion of those which have been already implemented persistently. Section 3.1 introduces a new variant of a suffix tree and section 3.2 presents how this variant can be stored efficiently on disk, including exact matching. Section 4 contains some implementation aspects, and finally, I present first results of a performance study.

2 Related Work

2.1 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. String location is performed by making a binary search. The search complexity of a binary search 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 for 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 or enhanced suffix arrays (cf. [5], [1]). All these kinds of suffix arrays deals with the introduction of additional array dimensions standing for lcp (longest common prefix) values or skip factors.

In contrast, 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. The most detailed logical data structure is the suffix tree. The most sporadic the suffix array.

For all these structures, 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 (chromosomes are regarded as words) 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 is expected as better, to code the letters of the suffix in the index itself. Also, data structures working with backtracking, such as Patricia tries, are unsuitable in the observed case.

Therefore, one have to find an optimal data structure in relation to both, its theoretical search complexity, and the possibilities of its practical implementation. But a closer look shows, that these two aspects intertwine.

2.2 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 pointed out in [4], 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 an efficient storing mechanisms.

A partitioning algorithm has been introduced in [4], where a large tree is divided in partitions, each of them created in-memory and then written on disk. PJama is used, which makes the in-memory structure automatically persistent. In [4] it has been mentioned that a space efficient implementation is needed.

In [8] 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 [4]. 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.

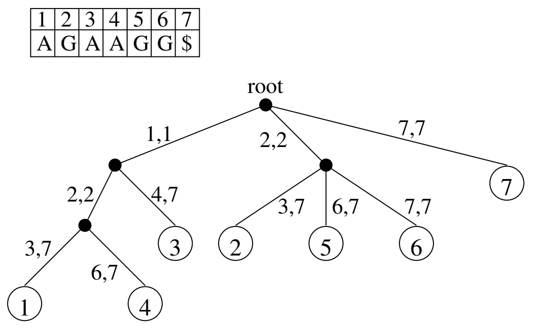

Figure 1: Normal suffix tree

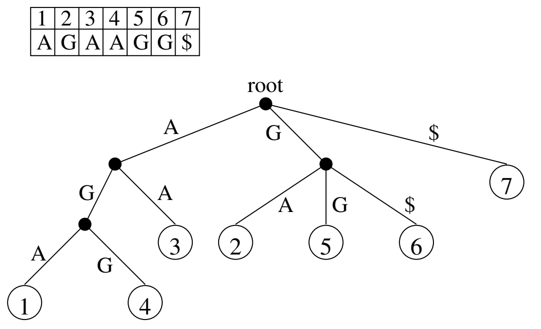

Figure 2: The variant of a suffix tree

3 The new Approach

3.1 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 appropriate. The same holds for structures working with backtracking, unless one is able to controll the backtracking behaviour. Therefore, a data structure supporting the straight forward search is needed. This, for example, is guaranted for a suffix tree. Such trees permit string searches like "Is s a substring of the text T?" to be answered in $O(|s|)$ time.

A *normal suffix tree* is a compressed digital trie. A trie is a tree for storing strings, in my case all suffixes, in which there is one node for every common prefix. A compressed trie 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 linearily 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 it is 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 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 introduce 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.

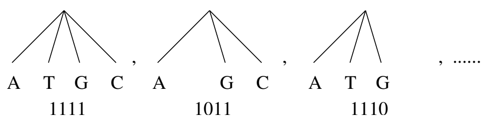

Figure 3: Treecode

3.2 The Physical Representation - The Tree Coding Structure

Since the partitions of the tree are stored on disk, 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 of 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

Table 1: Data type of variable sized record structure

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

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. Therefore, in the topmost node the maximal number of 4 Bytes are needed. But, for the most nodes the numbers are small, e.g. 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.

4 Implementation and Experimental Results

4.1 Implementation

In principle, there are 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.

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. The relational database concept doesn't consider recursive data structures. Today's commercial systems are in practice strongly optimized concerning their secondary storage managment.

A commercial object-oriented DBMS or persistent object-oriented programming languages comes to meet the tree structure more strongly. Since the object-header of the persistent objects would be substantially larger than it's content, it is not clever to make an object out of each node. The index would be further enlarged.

As described above, I have developed my own data format and have written it to index files on operating system level. All the code is written in the C programming language. The implementation has been done exactly in a manner, that I have described in section 3.2. The main implementation advantage lies in the strongly compressed physical representation of the nodes on disk. It ensures that as many nodes as possible fit in one disk page. Therefore one can group together the nodes of an entire subtree.

4.2 Practical Results

Because I want to perform the approximate substring matching with filter algorithms, like q-grams (see [2], [7]), and these techniques are based at their core on exact substring matches, I have optimized the introduced data structure for a fast exact match. Consequently, I first focus on the evaluation of the structure for exact substring matches. I present the average query response time for large batches of randomly generated query patterns of different length. All experiments were processed on a 1200 MHz AMD Athlon PC with 256 MB RAM, running under the Linux operating system. For my experiments I have used a $33,4$ MB DNA sequence, consisting of pieces of chromosome 22 of the human DNA. The construction time for this DNA text is about 4,5 hours. The needed storage space is about 838 MB.

I have examined tests for the exact match, see Table 2 and 3. Table 2 shows the cold store query behaviour. All new disk pages needed for the search have to be loaded into main memory. Therefore, for large batches the average response time per query reduces, because following queries use index disk pages cached in previ-

Table 2: Cold Store Query Behaviour

size of the batch	query length in bases	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	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

Table 3: Warm Store Query Behaviour

size of the batch	query length in bases	average response time per query in ms	total hits
100	12	< 0,1	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	< 0,1	26.113
1.000	17	< 0,1	46.636
5.000	17	0,1	146.656
7.000	17	0,14	1.112.030
100	30	< 0,1	885
1.000	30	< 0,1	2.765
5.000	30	0,1	11.800
7.000	30	0,14	14.247

ous queries. Under the warm store condition, most of the used disk pages are in-memory, and therefore the response time is significant smaller.

These results show that this implementation is one magnitude faster than other results, for example, see [4]. This result does not come as a surprise, since it has not been attempted to optimize the tree representation in previous work. Summing up, a considerable speed-up of query performance can be achieved using the technique described above.

5 Conclusion

This paper shows, how explicite persistent memory management can be performed for large suffix trees. We have given an answer to the question, which suffix structure is the most applicable for persistent indexing. The runtimes for exact string matches, we have observed, compare favorably with other implementations.

Further, we plan to improve the runtime of the construction algorithm in order to obtain a complexity better than quadratic. The goal is to index whole genomes of several species simultaneously. And a real world application would be to reveal all significant matches between the human, the mouse and the rat genome. This will require a thoroughly investigation into approximate pattern searching algorithms. The aim is to improve their interaction with an index. This has to be done in the difficult case that this index lies in secondary memory.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, *The Enhanced Suffix Array and Its Applications to Genome Analysis.* WABI 2002, pp. 449-463.

[2] 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.

[3] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge: University Press, 1997.

[4] 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.

[5] 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.

[6] J. Kent, *BLAT - The BLAST-Like Alignment Tool.* Genome Res. 12: 656-664, 2002.

[7] G. Navarro, *A Guided Tour to Approximate String Matching.* ACM Computing Surveys, vol.33, pp.31–88, 2001.

[8] K.-B. Schürmann, J. Stoye, *Suffix Tree Construction for Large Strings.* In: Proceedings 14. Workshop of Fundamentals of Databases. Rostock, Germany, May 2002.