A Database Index to Large Biological Sequences

Ela Hunt Malcolm P. Atkinson Robert W. Irving

Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK
{*ela,mpa,rwi*}*@dcs.gla.ac.uk*

Abstract

We present an approach to searching genetic DNA sequences using an adaptation of the suffix tree data structure deployed on the general purpose persistent Java platform, PJama. Our implementation technique is novel, in that it allows us to build suffix trees on disk for arbitrarily large sequences, for instance for the longest human chromosome consisting of 263 million letters. We propose to use such indexes as an alternative to the current practice of serial scanning. We describe our tree creation algorithm, analyse the performance of our index, and discuss the interplay of the data structure with object store architectures. Early measurements are presented.

1 Introduction

DNA sequences, which hold the code of life for every living organism, can be abstractly viewed as very long strings over a four-letter alphabet of *A, C, G* and *T*. Many projects to sequence the genome of some species are well advanced or concluded. The very large number of species (and their genetic variations) that are of interest to man, suggest that many new sequences will be revealed as the improved sequencing techniques are deployed. Consequently we are at a technical threshold. Techniques that were capable of exploiting the smaller collections of genetic data, for example via serial search, may require radical revision, or at least complementary techniques. As the geneticists and medical researchers with whom we work seek to search multiple genomes to find model organisms for the gene functions they are studying, we have been investigating the utility of indexes. The fundamental lack of structure in genetic sequences makes it difficult to construct efficient and effective indexes.

The length of a DNA sequence can be measured in terms of the number of base pairs (bp). Because of their size, gigabase pairs (Gbp) is a more convenient unit. For example, mammalian genomes are typically 3 Gbp in length. The largest public database of DNA[1] which contains over 15 Gbp (June 2001), is an archive which holds indexes to fields associated with each DNA entry but does not index the DNA itself. In the industrial domain, Celera Genomics[2] have sequenced several small organisms, the human genome, and four different mouse strains. Their sequences are accessed as flat files.

Searching DNA sequences is usually carried out by sequentially scanning the data using a filtering approach [46, 2, 1], and discarding areas of low string similarity. Typically, this approach uses a large infrastructure of parallel computers. Its viability depends on biologists being able to localise the searches to relatively small sequences, on skill in providing appropriate search parameters, and on batching techniques. Even under these circumstances it cannot always deliver fast and appropriate answers. Using BLAST on the hardware configuration described in section 6 (and all 4 processors), we compared 99 queries[3] (predicted human genes of length between 429 and 5999 bp) to a BLAST "database"[4] for 3 human chromosomes (294 Mbp, 10% of the human genome). The search took 62 hours (average 37 minutes per query) with default BLAST parameters, and delivered 6559 hits with an average of 66.25 hits per query and a median of 34. Some hits spanned only 18 characters but those had very high similarity. 17 out of 99 queries came from the chromosomes stored in the BLAST "database" and they produced several exact hits each (corresponding to the non-contiguous nature of DNA strings contributing to human genes). As there is a rapid rise in

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

[1] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide

[2] http://www.celera.com

[3] ftp://ftp.ensembl.org/current/data/fasta/cdna/ensembl.cdna.gz

[4] BLAST package includes a command *formatbd* which compresses DNA and creates indexes of sequence names and occurrences of non-repetitive and repetitive DNA.

both the volume of data and the demand for searches by researchers investigating functional genomics, it is worth investigating the possibility of accelerating these searches using indexes.

The appropriate indexes over large sequences can take many hours to construct, hence it is infeasible to construct them for each search[5]. On the other hand, the sequences are relatively stable, so that it may be possible to amortise this construction cost over many thousand searches. That depends on developing techniques for storing the indexes persistently, i.e. on disk. As we will explain, that has not proved straightforward, but we believe that we now have the prototype of a viable technology. We focus our attention on persistent suffix trees for reasons given below.

To our knowledge, no existing database technology can support indexed searches over large DNA strings and the feasibility of indexed searches over large strings is an open research question [42, 11]. Inverted files [57] are not suitable, because DNA cannot be broken into words. For similar reasons the String B-tree [22] may not be an appropriate choice. Approaches based on q-grams [15, 39] are fast, but cannot deliver matches that have low similarity[6] to the query [42]. It appears that the suffix tree [56, 38, 53] is the data structure of choice for this type of indexing, but so far, suffix trees on disk could only be built for small sequences, due to the so-called "memory bottleneck" [21]. Baeza-Yates and Navarro [10] state that "suffix trees are not practical except when the text size to handle is so small that the suffix tree fits in main memory". We address exactly this question, and show how to build large suffix trees, and how to deliver fast query responses.

Our initial prototype was built using PJama [8, 31, 5] which provides orthogonal persistence for Java. We are investigating other persistence mechanisms, including an object-oriented database, Gemstone/J[7], and tailored mapping to files. The latter may ultimately be necessary, given the data volumes and performance requirements. However, for the present, the general purpose object-caching mechanisms of PJama and Gemstone/J allow rapid experiments with a variety of index structures.

The rest of this paper is structured as follows: Section 2 summarises previous work, Section 3 introduces the suffix tree and Section 4 introduces the new algorithm. Aspects of PJama, our experimental platform are presented in Section 5. The test data and experimental results are described in Section 6. A discussion of these results and our research plans conclude the paper.

2 Previous work

We review three areas: persistent suffix tree construction, suffix tree storage optimisation, and alternative data structures.

Persistent indexes to small sequences have been built previously. Bieganski [13], built persistent suffix trees up to 1 Mbp. Recently, Baeza-Yates and Navarro [44, 10] built persistent suffix trees for sequences of 1 Mbp using a machine with small memory (64 MB) and concluded that trees in excess of RAM size cannot be built. Farach's theoretical work to remove the I/O problem [21] reduces suffix tree creation complexity to that of sorting and extends the computational model to take into account disk access. The bottleneck is considered to lie in random access to the string being indexed. In our opinion, it is not only the source string itself but the tree data structure and the suffix links which contribute to the bottleneck. An empirical evaluation of that method has not been reported. The only recent accounts of large persistent suffix trees representing sequences of 20.5 Mbp are in our previous work [26, 27].

Optimisations of suffix tree structure were undertaken by McCreight [38], and more recently by Kurtz [32]. Kurtz reduced the RAM required to around 13 bytes per character indexed, for DNA (our measurements using Kurtz's code), but his storage schemes have not been tested on disk yet. We believe that some extra space overhead will be inevitable. More recent work on suffix tree storage optimisation [40] states that compact suffix trees will require too many disk accesses to make the structure viable for secondary memory use.

Alternative data structures include: q-grams [51, 43, 45] the suffix array [36], LC-tries [3], the String B-tree [22], the prefix index [29] and suffix binary search trees [28].

Two recent overviews of approximate text searching methods [42, 11] show that filtering approaches are only suitable for high similarity matching. This prohibits us from using the q-gram structure. Because DNA has no word structure, we exclude the String B-tree and prefix indexes. Other researchers have used suffix arrays [41, 10] to simulate the suffix tree, but have shown results only for up to 10 Mbp. We made an initial investigation of Irving's suffix binary search trees (SBSTs) [27] but have not been able to build persistent trees for large datasets (over 50 Mbp). Using the technique of tree building presented in this paper we may be able to build large SBSTs as well.

We decided to focus on suffix trees because approximate matching algorithms using these structures are known [12, 52, 16, 44], because this data structure is used widely in biological sequence processing [14, 33, 19, 37, 54], and because there is a well-established range of biological methods using them [23].

[5] For example, the most space efficient main-memory index would take 9 hours and 45 Gbytes to index the human genome [32].

[6] Low similarities are often biologically significant.

[7] http://www.gemstone.com/products/j/

3 Suffix trees

Suffix trees are compressed digital tries. Given a string, we index all suffixes, e.g. for a string of length 10, all substrings starting at index 0 through 9 and finishing at index 9 will be indexed. The root of the tree is the entry point, and the starting index for each suffix is stored in a tree leaf. Each suffix can be uniquely traced from the root to the corresponding leaf. Concatenating all characters along the path from the root to a leaf will produce the text of the suffix.

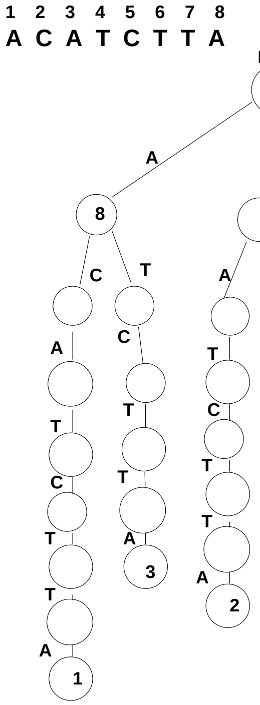

Figure 1: An example trie on **ACATCTTA**.

An example digital trie representing **ACATCTTA** is shown in Figure 1. The number of children per node varies but is limited by the alphabet size. This trie can be compressed to form a suffix tree, shown in Figure 2.

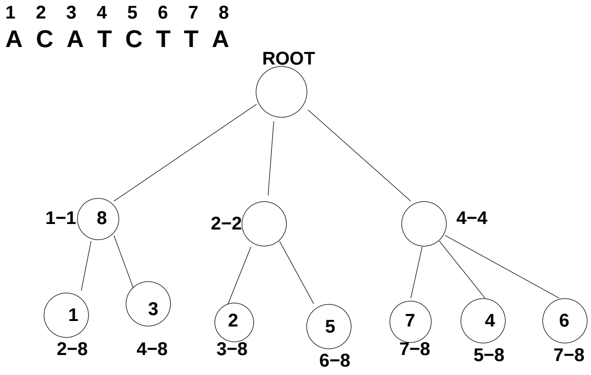

Figure 2: An suffix tree on **ACATCTTA**.

To change a trie into a suffix tree, we conceptually merge each node which has only one child with that child, recursively, and annotate the nodes with the indices of the start and end positions of a substring indexed by that node. Commonly, a special terminator character is also added, to ensure a one-to-one relationship between suffixes and leaves (otherwise a suffix that is a proper prefix of another suffix would not be represented by a leaf — for instance node number 8 in Figure 2). The change from a trie to a suffix tree reduces the storage requirement from $O(n^2)$ to $O(n)$ [56, 38, 53].

Most implementations of the suffix tree also use the notion of the suffix link [53]. A suffix link exists for each internal node, and it points from the tree node indexing aw to the node indexing w, where aw and w are traced from the root and a is of length 1. Suffix links were introduced so that suffix trees could be built in $O(n)$ time. However, in our understanding, they are also the cause of the so-called "memory bottleneck" [21]. Suffix links, shown in Figure 3, traverse the tree horizontally, and together with the downward links of the tree graph, make for a graph with two distinct traversal patterns, both of which are used during construction. Ineluctably, at least one of those traversal patterns must be effectively random access of the memory. At each level of the memory hierarchy this induces cache misses. For example, it makes reliance on virtual memory impractical.

As would be expected from this analysis, we have observed very long tree construction times when using disk with the $O(n)$ suffix-link based algorithms. A first approach is to attempt to build the trees incrementally, checkpointing the tree after each portion has been attempted. Here, the suffix-link based algorithm exhibits another form of pathological behaviour. The construction proceeds by splitting existing nodes, adding siblings to nodes and filling in suffix-link pointers. As a result of the dual-traversal structure, no matter how the tree is divided into portions, a large number of these updates apply to the tree already checkpointed. This has the cost of installation reads and logged writes, if the checkpointed structure is not to be jeopardised. In addition, the checkpointed portions of the tree are repeatedly faulted into main memory by the construction traversals.

These effects combine to limit the size of tree that can be constructed and stored on disk using suffix-link based algorithms to approximately the size of the available main memory. For example, in Java, using 1.8 Gbytes of available main memory we could build transient trees for up to 26 Mbp sequences. Using the suffix-link based algorithm under PJama, checkpointing trees indexing more than 21 Mbp has not been possible [26, 27] (the reduction on using PJama is due to two effects: (i) it increases the object header size, and (ii) it competes for space, e.g. to accommodate the disk buffers and resident object table [6, 34]). We have therefore investigated incremental construction algorithms in which we forego the guarantee of $O(n)$ complexity (see Section 4).

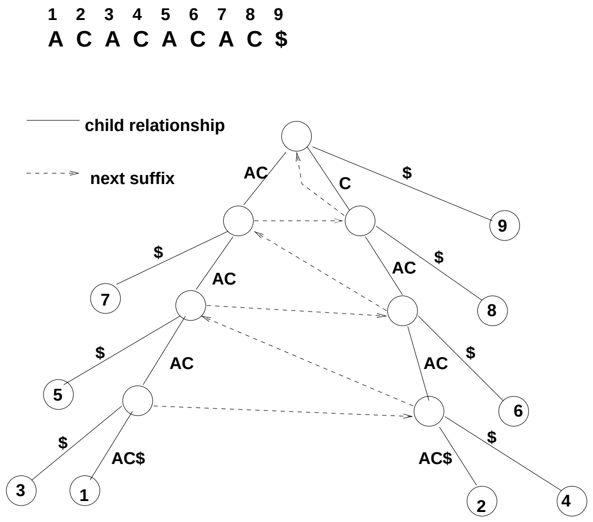

Figure 3: Suffix tree and links on **ACACACAC$**.

3.1 Suffix tree representation

Though their space requirements are $O(n)$, straightforward encodings of suffix trees require substantial space per letter. A recent contribution by Kurtz [32] presents the most efficient main-memory representation to date. He discusses 4 different data structures, based on linked lists and hash tables. Kurtz's tree is a RAM-only tree, coded in C, where every spare bit is used optimally, and approximately 13 bytes are needed per letter indexed. Kurtz's tree uses suffix links, and may suffer from the same "memory bottleneck" if moved into the database world. This requires investigation.

4 The new construction algorithm

The new incremental construction algorithm trades ideal $O(n)$ performance for locality of access on the basis of two decisions:

1. to abandon the use of suffix links, and
2. to perform multiple passes over the sequence, constructing the suffix tree for a subrange of suffixes at each pass.

These are both necessary. Removing the suffix links means that the construction of a new partition corresponding to a different subrange does not need to modify previously checkpointed partitions of the tree. Using multiple passes, each dealing with a disjoint subrange of the suffixes, means that it is not necessary to access or update the previously checkpointed partitions. Data structures for the complete partitions can be evicted from main memory and will not be faulted back in during the rest of the tree's construction. Thus the main memory is available for the next partition and its size is a determinant of the partition size and hence the number of passes needed. An additional benefit of this partition structure is that the probable clustering of contemporaneously checkpointed data will suit the lookup and search algorithms. Further details of our algorithm are now presented.

4.1 Suffix tree construction

Several $O(n)$, suffix-link based, tree building algorithms are known [56, 38, 53, 21, 35], but they have not proved appropriate for large persistent tree construction undertaken by Navarro [44] or ourselves. In contrast, the algorithm we use is $O(n^2)$ in the worst case, but due to the pseudo-random nature of DNA, the average behaviour is $O(nlogn)$ *for this application* [50].

We base our partitions on the prefixes of each suffix, since the suffixes that have the prefix **AA** fall in a different subtree from those starting with **AC**, **AG** or **AT**. The number of partitions and hence the length of the prefix to be used is determined by the size of the expected tree and the available main memory. It may be the case that smaller partitions would be better because their impact on disk clustering would accelerate lookups, but this has yet to be investigated.

The number of partitions required can be computed by estimating the size of a main-memory instantiation S_{mm}, available for tree construction, and the number of partitions, p, is

$$\left\lceil \frac{S_{mm}}{A_{mm}} \right\rceil,$$

where A_{mm} is the available main memory. The actual partitioning can be carried out using either of the two approaches we outline. One way is to scan the sequence once, for instance using a window of size 3 (sufficient for 263 Mbp and 2 GB RAM), count the number of occurrences of each 3-letter pattern, and then pack each partition with different prefixes, using a bin-packing algorithm [18]. Alternatively, we can assume that, given the pseudo-random nature of DNA, the tree is uniformly populated. To uniformly partition, we calculate a prefix code, P_i, for each prefix of sufficient length, l, using the formula:

$$P_i = \sum_{j=0}^{l-1} c_{i+j} a^{l-j-1},$$

where c_k is the code for letter k of the sequence, and

a is the number of characters in the alphabet[8]. The code of a letter is its position in the alphabet, i.e. **A** codes as 0, **C** codes as 1, etc. The minimum value for P_i is 0 and its maximum is $a^l - 1$. So the range of codes for each partition, r, is given by:

$$r = \left\lceil \frac{a^l - 1}{p} \right\rceil .$$

The suffixes that are indexed during the jth pass of the sequence have $jr \leq P_i < (j+1)r$. The structure of the complete algorithm is given as pseudocode below:

```
for j in partitions do
  for i in 0..totalLength do
    if suffix i is in partition j
      new Node(i);
      insert node;
    endif
  endfor
  checkpoint;
endfor
```

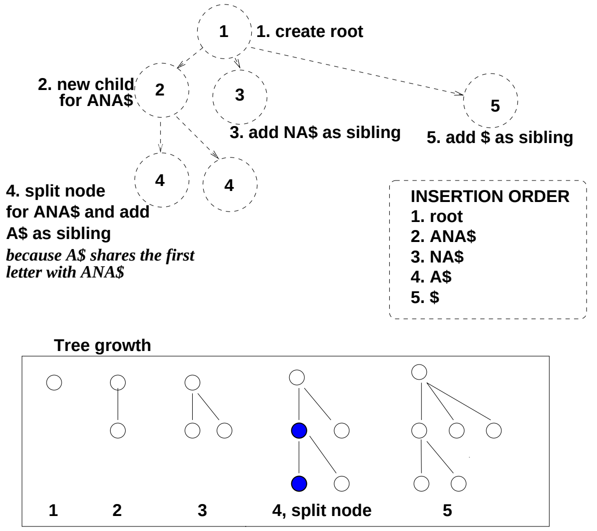

Figure 4: Tree creation for **ANA$**.

A node consists of three fields: **child** node, **sibling** node and an integer **leftIndex**. A new node represents a suffix stretching from position i to the end of the string. It has **null** child and sibling, and its **leftIndex** set to i (its suffix number). Insertion starts from the root, and as the search for the insertion position proceeds down the tree, the left index is updated. This downward traversal matches the new suffix to suffixes which are already in the tree, and which share a prefix with the new suffix. When the place of insertion is determined, the node will either be added as a sibling to an existing node, or will cause a split of an existing node, see Figure 4.

4.2 Space requirements

Our new implementation disposes of suffix links. Further to that, we reduce storage by not storing the suffix number and the right index into the string for each node. The suffix number is calculated during tree traversal (during the search). The right pointer into the string is looked up in the child node, or, in the case of leaves, is equal to the size of the indexed string. Each tree node consists of two object references costing 4 B each (child, sibling), one integer taking up 4 B (leftIndex) and the object header (8 B for the header in a typical implementation of the Java Virtual Machine). The observed space is some 28 B per node in memory. The difference is due to PJama's housekeeping structures, such as the resident object table [34].

PJama's structure on disk adds another 8 B per object over Java, i.e. 36 B per node. The actual disk occupancy of our tree is around 65 B per letter indexed, close to that expected. The observed number of nodes for DNA remains between $1.6n$ and $1.8n$, where n is the length of the DNA, giving an expectation of between 58 and 65 bytes per node. Some of this space may well be free space in partitions, and some is used for housekeeping [47]. If we wanted to encode the tree without making each node an object, we would require 12 B per node, that is around 21 B per character indexed. But further compression could be obtained by using techniques similar to those proposed by Kurtz [32].

4.3 Using the index

Exact pattern matching in a suffix tree involves one partial traversal per query. From the root we trace the query until either a mismatch occurs, or the query is fully traced. In the second case, we then traverse all children and gather suffix numbers representing matches. The complexity of a suffix tree search is $O(k+m)$ where k is the query length and m the number of matches in the index. This means that looking for queries of length q may bring back a $\frac{1}{a^q}$ fraction of the whole tree, where a is here the size of the active alphabet, 4 in this application. For example, a query of length 4 might retrieve $\frac{1}{256}$ of the tree. Composite algorithms may be necessary, where short queries are served by a serial scan of the sequence, and longer queries use the index. The threshold at which indexing begins to show an advantage depends on the precise data structure used, on the query pattern, and on the size of the sequence. We currently estimate this threshold to be in the region of minimum query length of 10 to 12 letters for human chromosomes.

[8] Combinations of * can be used to denote unknowns, sequence concatenation and end of sequence. Hence a can be reduced to 5. In this case l set to 8 provides even division of partitions for all likely sequence length to available memory ratios.

5 The PJama platform

The first set of experimental trials of this new algorithm has been conducted using the PJama[9] platform [6, 4, 7, 8, 5, 31, 48, 24, 47]. We selected PJama to minimise the software engineering cost of providing integrated software environments supporting a very wide range of bioinformatics tasks. PJama enabled easy transitions between different underlying tree representations, and immediate transparent store creation from Java without any intermediate steps. Both transient and persistent trees can be produced using the same compiled code, but a different command-line parameter for PJama indicating whether a persistent store is being used.

Although tuned, purpose-built mechanisms may be appropriate for large-scale indexes, the cost of implementing them and maintaining them would be an impediment to rapid experimentation. In addition, a great many index technologies are proposed and tested, in this area of application, as well as many others. Hence, if we can make the general purpose persistence mechanism work for indexes, there could be considerable pay offs in reduced implementation times and more rapid deployment.

We expect that applications of the suffix trees will require much annotation and other data to make them useful to the biologists. This data, at least, does not have demanding processing and access performance requirements. Consequently, there are advantages to developing as much of the application code as possible in Java, for ease of multi-platform deployment. Here we expect to utilise PJama's schema and object evolution facilities [20, 25].

6 Test data and experimental results

In this section we report results for exact matching on DNA strings. The test data consisted of 6 single chromosomes of the worm *C. elegans*, of 20.5 Mbp maximum[10] and of some 280 Mbp merged DNA fragments from human chromosomes 21, 22 and 1[11]. As queries we used short worm and human sequences, from the STS division of Entrez[12], and from each sequence initial characters were taken to be used as query strings. Similarly, for the worm queries, we used short sequences called cDNAs.

Our alphabet in this experiment consists of **A, C, G, T**, a terminal symbol **$**, and ***** used as a delimiter for merged sequences.

Tests were carried out using production Java 1.3 for transient measurements, and PJama, which is derived from Java 1.2 and uses JIT, for the persistence measurements. All timing measurements were obtained using Solaris 7 on an Enterprise 450 SUN computer with 2 GB RAM, and data residing on local disks. In this experiment our algorithm did not use multithreading and therefore only one of the four 300 MHz SPARC processors was used for the main algorithm. Parts of the Java Virtual Machine, and PJama's object store manager, will have made some use of another processor for housekeeping tasks.

6.1 Trees with suffix links

We first investigated the optimal tree [53] which can be built in $O(n)$ time. A tree for 20.5 Mbp of DNA was created in memory in 7 minutes on average. However, on disk, the creation time was around 34 hours, and checkpoints at 12 million and then every 0.5 million nodes were required. For 20.5 Mbp of worm data we used a 2 GB log, and one store file of 2 GB. This was the largest tree of this type that we could build.

A tree for 20.5 Mbp fitted mostly in memory (2 GB RAM, 2 GB store). Table 1 shows the results obtained for a batch of 10,000 queries run on a cold store.

query length	avg time (ms)	total hits
8	920	8,568,303
9	263	2,553,520
10	142	758,523
15	36	3,687
50	34	394
100	34	305
200	33	107

Table 1: Cold store queries over 20.5 Mbp using an $O(n)$ index.

6.2 Without suffix links

We then indexed 263 Mbp of DNA using the $O(nlogn)$ algorithm presented here. The store required a 2 GB log and 18 GB in files of 2 GB each. Store creation time was 19 hours in our first run, and this could probably be shortened. Queries of the same length were sent in batches, without the use of multithreading[13].

We ran experiments on a cold store, see Figure 5, and on a warm store, see Figure 6. We observed that large batches produced faster response times, due to the benefit of objects that had been faulted in for previous queries still being cached on the heap.

Table 2 shows why the cold store runs for short queries take so long. The time taken, can be divided into *matching* the query's text by descending the tree, and faulting in and traversing the subtree below the matched node to *report* the results. For short queries,

[9] http://www.dcs.gla.ac.uk/pjama

[10] ftp://ftp.sanger.ac.uk/pub/C.elegans_sequences/CHROMOSOMES/

[11] ftp://ncbi.nlm.nih.gov/genomes/H_sapiens

[12] ftp://ncbi.nlm.nih.gov/repository/dbSTS/

[13] In other experiments [27], we have demonstrated a significant speed up by using multiple threads to handle a batch of queries.

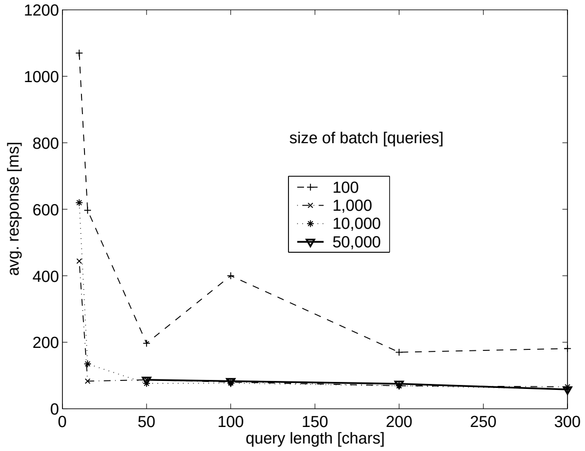

Figure 5: Cold store query performance.

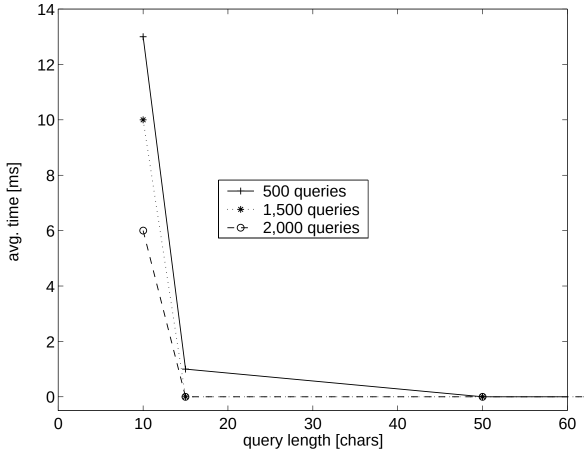

Figure 6: Queries run over a warm store.

many results are reported and the reporting time dominates. For longer queries, fewer results are found, and the average query response improves.

7 Discussion

The new incremental algorithm for constructing disk-resident suffix trees without suffix links appears to have the potential to build arbitrarily large indexes efficiently. We are optimistic that this construction and the subsequent index use behaviour can be made sufficiently efficient that it will be a useful component of biological search systems. Some of the support for this claim is now presented.

Theoretical investigations of suffix tree building indicate that the use of suffix links to obtain an $O(n)$ algorithm is worthwhile. However, suffix links require space, and generate a difficult load on memory, with scattered updates and reads. In Figure 7 we show *in-memory* performance comparison of suffix trees with and without suffix links. We use a modified version of Ukkonen's algorithm [53] which does not perform a final tree scan to update the right text pointer in the leaves, and compare it to our tree without suffix links. We are limited here by 2 GB RAM, and carry out the tests using Java 1.3 with flags -server -Xmx1900m. The largest suffix-link tree we can build in this space is for 25 Mbp. Up to that value, no significant difference in tree construction speed can be observed (times are best times observed over several tree builds).

batch size	query length	avg time (ms)	total hits
100	10	1070	155,007
1000	10	444	1,289,800
10000	10	620	10,217,838
100	50	197	18
1000	50	87	221
10000	50	76	660
50000	50	87	25376

Table 2: Cold store query behaviour.

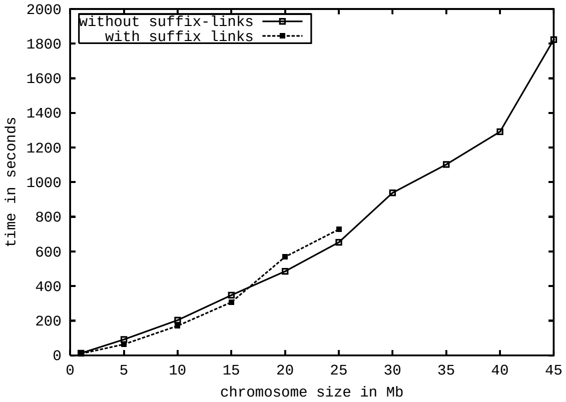

Figure 7: Tree creation in memory.

The incremental partitioned construction algorithm uses a partition size which we select. So far our experiments suggest that this should correspond to about 20 Mbp. This means that we are using the tree builder in a region where the $O(n)$ suffix-link algorithm offers no advantage.

The comparison of unoptimised *persistent* tree building times shows that our algorithm outperforms the suffix-link tree both in terms of time and size, and we believe that building times in the region of 5 hours for the longest human chromosome will be possible. Our algorithm is scalable and can be adjusted to run on computers with different memory characteristics. More work is required to optimise the tree building, and to investigate the object placement on disk and its influence on query performance. Our algorithm opens up the perspective of building suffix trees in parallel, and the simplicity of our approach can make suffix trees more popular. In the parallel context, maintaining suffix links between different tree partitions may not be viable or necessary, as further characterisation of the space-time tradeoff between suffix trees with links and without is needed.

8 Future work

Future work can be divided into four interrelated parts.

- Improvements to the tree representation and incremental construction algorithm.
- Investigation of the interaction between approximate matching algorithms and disk-based suffix trees.
- Investigation of alternative persistent storage solutions.
- Integration of the algorithms with biological research tools and usability studies.

Improving the tree representation is amenable to several strategies. We are investigating the replacement of the top of each tree with a sparse array indexed by P_i. We have also identified significant savings by specialising nodes (similar to some aspects of Kurtz's compression) and we are measuring the gains from storing summaries to accelerate reporting.

At the underlying object store level, we are looking at compressions that remove the object headers, at placement optimisations, and at improved cache management. We are experimenting with direct storage strategies.

As the deployed system will need to be trustworthy for biologists, we started field trials using Gemstone/J rather than PJama which is no longer maintained at the PEVM level [34, 8]. This will enable us to operate on other hardware and operating system platforms and to verify that the phenomena so far observed are not artefacts of PJama. Gemstone/J uses a similar implementation strategy to PJama, modifying the JVM to add read and write barriers. This provides comparable speed for large applications and nearly the same programming convenience. We plan to return to research into optimised persistent virtual machines once an optimised open source VM is available.

We are currently testing approximate matching algorithms similar to that of Baeza-Yates and Navarro [10]. Further work will include adopting biological measures of sequence similarity [2, 49]. Our ultimate aim is to enable comparisons of different species based on DNA and protein sequence similarity. Future matching methods will be accompanied by statistical measures of sequence similarity, and will be presented in the context of other biological knowledge. We see that future to lie in a uniform database approach to all types of biological data, including sequence, protein structure and expression data.

We plan to investigate several applications of suffix trees to biological problems. One of them is the identification of repeating sequence patterns on a genomic scale. Some of those patterns, positioned outside gene sequences, point to regulatory sequences controlling gene activity. We will also use our trees in gene comparison within and across species. Because of the RAM limit on suffix tree size, all-against-all BLAST is traditionally used in this context [17, 55], and it would necessitate up to

$$2\begin{pmatrix} 40000 \\ 2 \end{pmatrix}$$

gene alignments to perform full gene comparison within the human genome which has around 40,000 genes[14]. The use of large suffix trees in this context is likely to be beneficial. Finally, assembly of genomes can be speeded up using suffix trees [23].

9 Conclusions

An algorithm has been developed that promises to overcome a long standing problem in the use of suffix trees. It enables arbitrarily large sequences to be indexed and the suffix tree built incrementally on disk. Surprisingly, there seems to be no measurable disadvantage to abandoning the suffix-links that have been introduced to achieve linear-time construction algorithms. Much further experimentation and analysis is required to develop confidence in these early, but intriguing results.

References

[1] S. F. Altschul, T. L. Madden, A. A. Schaeffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[2] S.F. Altschul et al. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–10, 1990.

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Softw. Pract. Exp.*, 25(2):129–141, 1995.

[4] M. Atkinson and M. Jordan. Providing Orthogonal Persistence for Java. In *ECOOP98*, LNCS 1445, pages 383–395, 1998.

[5] M.P. Atkinson. Persistence and Java – a Balancing Act. In *ECOOP Symp on Objects and Databases*, LNCS 1944, pages 1–32, 2000.

[6] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, 25(4):68–75, 1996.

[14] Blast is run in both directions because it is an asymmetric matching algorithm.

[7] M.P. Atkinson and M.J. Jordan. Issues Raised by Three Years of Developing PJama. In *ICDT99*, LNCS 1540, pages 1–30, 1999.

[8] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical Report TR-2000-90, Sun Microsystems Laboratories Inc and Dept. Computing Science, Univ. Glasgow, 2000.

[9] M.P. Atkinson and R. Welland, editors. *Fully Integrated Data Environments.* Springer-Verlag, 1999.

[10] R. Baeza-Yates and G. Navarro. A Hybrid Indexing Method for Approximate String Matching. *Journal of Discrete Algorithms*, 2000. To appear.

[11] R. Baeza-Yates, G. Navarro, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate Text Retrieval. Technical report, University of Chile, 1997.

[12] R.A. Baeza-Yates and G.H. Gonnet. All-against-all sequence matching. Technical report, Dept. of Computer Science, Universidad de Chile, 1990. ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/papers/all-all.ps.gz.

[13] P. Bieganski. *Genetic Sequence Data Retrieval and Manipulation based on Generalised Suffix Trees.* PhD thesis, University of Minnesota, USA, 1995.

[14] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Predicting Gene Regulatory Elements in Silico on a Genomic Scale. *Genome Research*, 8:1202–1215, 1998.

[15] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. *q*-gram Based Database Searching Using a Suffix Array. In *RECOMB99*, pages 77–83. ACM Press, 1999.

[16] A. L. Cobbs. Fast Approximate Matching using Suffix Trees. In *CPM95*, LNCS 937, pages 41–54, 1995.

[17] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[19] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27:2369–2376, 1999.

[20] M. Dmitriev and M. P. Atkinson. Evolutionary Data Conversion in the PJama Persistent Language. In *1st ECOOP Workshop on Object-Oriented Databases*, pages 25–36, Lisbon, Portugal, 1999. http://www.disi.unige.it/conferences/oodbws99.

[21] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottlenect in Suffix Tree Construction. In *FOCS98*, pages 174–185, 1998.

[22] 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.

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

[24] C. G. Hamilton. Recovery Management for Sphere: Recovering A Persistent Object Store. Technical Report TR-1999-51, University of Glasgow, Dept. of Computing Science, 1999.

[25] C. G. Hamilton, M. P. Atkinson, and M. Dmitriev. Providing Evolution Support for PJama$_1$ within Sphere. Technical Report TR-1999-50, University of Glasgow, Dept. of Computing Science, 1999.

[26] E. Hunt. PJama Stores and Suffix Tree Indexing for Bioinformatics Applications, 2000. 10th PhD Workshop at ECOOP00, http://www.inf.elte.hu/∼phdws/timetable.html.

[27] E. Hunt, R. W. Irving, and M. P. Atkinson. Persistent Suffix Trees and Suffix Binary Search Trees as DNA Sequence Indexes. Technical report, Univ. of Glasgow, Dept. of Computing Science, 2000. TR-2000-63, http://www.dcs.gla.ac.uk/∼ela.

[28] R.W. Irving and L. Love. The Suffix Binary Search Tree and Suffix AVL Tree. Technical Report TR-2000-54, Univ. of Glasgow, Dept. of Computing Science, 2000. http://www.dsc.gla.ac.uk/∼love/tech_report.ps.

[29] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *ACM SIGMOD Conference on Management of Data*, pages 403–414, 2000.

[30] M.J. Jordan and M.P. Atkinson, editors. *2nd Int. Workshop on Persistence and Java.* Number TR-97-63 in Technical Report. Sun Microsystems Laboratories Inc, Palo Alto, CA, USA, 1997.

[31] M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for the Java Platform — Specification. Technical Report SML 2000-94, Sun Microsystems Laboratories Inc, 2000.

[32] S. Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exp.*, 29:1149–1171, 1999.

[33] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, pages 426–427, 1999.

[34] B. Lewis, B. Mathiske, and N. Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In *9th Intl Workshop on Persistent Object Systems*, 2000. TR-2000-93, http://research.sun.com/-research/techrep/2000/abstract-93.html.

[35] M. G. Maass. Linear Bidirectional On-Line Construction of Affix Trees. In *CPM2000*, LNCS 1848, pages 320–334, 2000.

[36] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[37] L. Marsan and M-F. Sagot. Extracting structured motifs using a suffix tree – Algorithms and application to promoter consensus identification. In *RECOMB00*, pages 210 – 219. ACM Press, 2000.

[38] E. M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the A.C.M.*, 23(2):262–272, 1976.

[39] C. Miller, J. Gurd, and A Brass. A RAPID algorithm for sequence database comparisons: application to the identification of vector contamination in the EMBL databases. *Bioinformatics*, 15:111–121, 1999.

[40] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39:205–222, 2001.

[41] E. W. Myers. A sublinear algorithm for approximate key word searching. *Algorithmica*, 12(4/5):345–374, 1994.

[42] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33:1:31–88, 2000.

[43] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electonic Journal*, 1(2), 1998.

[44] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *CPM99*, LNCS 1645, pages 163–185, 1999.

[45] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing Text with Approximate q-grams. In *CPM2000*, LNCS 1848, pages 350–365, 2000.

[46] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85:2444–8, 1988.

[47] T. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Dept. of Computing Science, University of Glasgow, 2000.

[48] T. Printezis and M. P. Atkinson. An Efficient Promotion Algorithm for Persistent Object Systems. *Softw. Pract. Exp.*, 31:941–981, 2001.

[49] T. A. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 284, 1981.

[50] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39:5:1647–1659, 1993.

[51] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–212, 1992.

[52] E. Ukkonen. Approximate string matching over suffix trees. In *CPM93*, LNCS 684, pages 228–242, 1993.

[53] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[54] A. Vanet, L. Marsan, A. Labigne, and M-F. Sagot. Inferring Regulatory Elements from a Whole genome. An Analysis of *Heliobacter pylori* σ^{80} Family of Promoter Signals. *J. Mol. Biol.*, 297:335–353, 2000.

[55] J. Craig Venter et al. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

[56] P. Weiner. Linear pattern matching algorithm. In *FOCS73*, pages 1–11, 1973.

[57] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 2nd edition, 1999.