The VLDB Journal (2002) / Digital Object Identifier (DOI) 10.1007/s007780200064

Database indexing for large DNA and protein sequence collections

Ela Hunt, Malcolm P. Atkinson, Robert W. Irving

Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK; e-mail: {ela,mpa,rwi}@dcs.gla.ac.uk

Edited by ♣. Received: 6 March 2002 / Accepted: ♣ 2002
– Published online: ♣ – © Springer-Verlag 2002

Abstract. Our aim is to develop new database technologies for the approximate matching of unstructured string data using indexes. We explore the potential of the suffix tree data structure in this context. We present a new method of building suffix trees, allowing us to build trees in excess of RAM size, which has hitherto not been possible. We show that this method performs in practice as well as the $O(n)$ method of Ukkonen [70]. Using this method we build indexes for 200 Mb of protein and 300 Mbp of DNA, whose disk-image exceeds the available RAM. We show experimentally that suffix trees can be effectively used in approximate string matching with biological data. For a range of query lengths and error bounds the suffix tree reduces the size of the unoptimised $O(mn)$ dynamic programming calculation required in the evaluation of string similarity, and the gain from indexing increases with index size. In the indexes we built this reduction is significant, and less than 0.3% of the expected matrix is evaluated. We detail the requirements for further database and algorithmic research to support efficient use of large suffix indexes in biological applications.

Keywords: Database index – Suffix tree – Approximate matching – Biological sequence

1 Introduction

1.1 The potential for indexing

Indexing technologies speed up data searching and have been very successfully applied in many areas of data processing. Different indexing mechanisms have been developed, each particularly suited to the type of data it is indexing and the type of search that is required. Indexes like B-trees [24] are now standard in database systems, and some newer indexing structures[1] are making their way into commercial systems. Text indexing for English text is very well advanced [74] but some data types, in particular biological sequence data or images, still elude indexing, and no commercially available database system known to us can index DNA or protein strings. The challenge in this area is the fact that biological sequences are searched not exactly, but by using approximate matching techniques. Navarro [54], in his recent survey, says that approximate string matching using indexes is an important but underdeveloped area of research. Our interest lies in this area, and, in particular, in the application of string indexing to biological sequences. We investigate the suffix tree structure, show how to build suffix trees for any size of data, and demonstrate that this data structure can speed up biological sequence searching.

[1] http://solutions.altavista.com/

1.2 Biological sequences

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*. Proteins, which use an alphabet of 20 symbols, are translations from selected stretches of DNA, using a predefined translation table where each 3 letters of DNA translate to one amino-acid (AA).

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. At the same time, proteins from those species are also being investigated and conceptual translations of entire genomes or their parts from DNA to protein are also made. 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 is measured in terms of the number of base pairs (bp), and only one base in each pair is represented, as the other base is its complement (A complements T, and C complements G). Because of large genome sizes, gigabase pairs (Gbp) or megabase pairs (Mbp) are more con-

venient units. For example, mammalian genomes are typically 3 Gbp in length. The largest public database of DNA[2], which contains over 20 Gbp (March 2002), 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[3] have sequenced several small organisms, the human genome, and four different mouse strains. The volume of protein data is smaller, and the latest release of SWISSPROT and TREMBL databases[4] counts around 200 million AAs, i.e., 200 megabases (Mb), as AAs are stringed into single strands and not paired helices. However, protein searching dominates because functional similarities between distant species are best seen at the level of protein, where similar stretches of amino-acids code for similar spatial structures and chemically active sites.

1.3 Sequential scanning

Both DNA and protein sequences are accessed as flat files. Searching for similar sequences is usually carried out by sequentially scanning the data using a filtering approach [59,3, 2], and discarding areas of low string similarity. Typically, this approach uses a large infrastructure of parallel computers. At the Sanger Centre, http://www.sanger.ac.uk, a farm of over 400 computers is available, and a large proportion of them are used in sequence similarity searching. The viability of this approach to searching 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 Sect. 5 (and all four processors), we compared 99 queries[5] (predicted human genes of length between 429 bp and 5999 bp) to a BLAST "database"[6] for three human chromosomes (294 Mbp, 10% of the human genome). The search took 62 h (average 37 min per query), with default BLAST parameters, and delivered 6,559 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. Seventeen 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 both the volume of data and the demand for searches by researchers investigating the mechanisms of cancer and inherited diseases like hypertension and diabetes, it is worth investigating the possibility of accelerating these searches using indexes.

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

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

[4] http://www.expasy.org

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

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

1.4 The indexing potential

The appropriate indexes over large sequences can take many hours to construct, hence it is infeasible to construct them for each search[7]. 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 approximate searches over large DNA strings and the feasibility of indexed searches over large strings is an open research question [54,57]. Inverted files [74] are not suitable, because DNA cannot be broken into words. For the same reason a prefix index [37] may not be appropriate. The String B-tree [29] is currently tailored to exact matching and not suitable in this context yet. Approaches based on q-grams [68, 55,19,52,58] are fast and proven, but cannot deliver matches that have low similarity[8] to the query [54]. The suffix array [47] is the closest competing structure, as it needs less space than a suffix tree. This structure is under investigation [14] and might deliver fast searching for large sequence repositories. However, it is not obvious how best to scale it up. Other competing structures include the LC-trie [5] which is a compressed suffix tree and the suffix binary search tree (SBST) [35,36]. The SBST can be viewed as a tree implementation of a suffix array and is more space efficient than a suffix tree. It performs very well in exact matching tests [34].

It appears that the suffix tree [73,49,70,31] is a good candidate data structure 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" [28] or "thrashing" [17]. Baeza-Yates and Navarro [14] 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", and use a suffix array instead, which reduces the storage required for the index. In this paper we respond to this challenge, and show how to build large suffix trees. We also adapt the algorithm of [14] to the needs of biological sequence searching, i.e., to the calculation of sequence similarity and not edit distance. We focus on the indexing gain, i.e., the actual reduction in the size of the matrix comparison problem, which in the worst case is $O(mn)$, and show that for index sizes of 200–300 million letters, the actual matrix size is reduced from mn to $0.003mn$ or less, assuming suitable combinations of query length and error level.

1.5 Overview of the paper

The rest of this paper is structured as follows: Section 2 summarises previous work, and Sect. 3 introduces the suffix tree. Section 4 presents our new algorithm for the construction of very large suffix trees. The test data and experimental results with tree construction and exact matching are described in

[7] For example, the most space-efficient main-memory index would take 9 h and 45 Gbytes of RAM to index the human genome [41].

[8] Low similarities are often biologically significant.

Sect. 5. Section 6 discusses our algorithm for suffix-tree-based string similarity searching using the dynamic programming method (DP), and Sect. 7 presents the approximate matching results. The discussion of our work is in Sect. 8. The paper closes with plans for further work, in Sect. 9, and Conclusions.

2 Previous work

We first review persistent suffix tree construction and suffix tree storage optimisations. We then position the dynamic programming technique in the context of approximate matching. Finally, we focus briefly on biological applications which use approximate matching techniques.

2.1 Persistent trees

Persistent indexes to small sequences have been built previously. Bieganski [16] built persistent suffix trees up to 1 Mbp. Recently, Baeza-Yates and Navarro [56,14] 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 [28] reduces suffix tree creation complexity to that of sorting and extends the computational model to take into account disk access. An empirical evaluation of their 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 [33,34].

2.2 Optimisations

Optimisations of suffix tree structure were undertaken by McCreight [49], and more recently by Kurtz [41]. 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. Since Kurtz's tree uses suffix links, it may suffer from the same "memory bottleneck" if moved into the database world. It appears that further investigation in this direction is warranted.

Compact encodings of the suffix tree, based on a binary representation of the text, have been investigated by Munro and Clark [20,21,53] and Larsson [4,43], but Munro [53] states that compact suffix trees will require too many disk accesses to make the structure viable for secondary memory use.

2.3 Approximate matching techniques

Recent overviews of approximate text searching methods [54, 57] are available and present a full classification of the available techniques. The techniques can be divided into dynamic programming (DP), automata, bit-parallelism and filtering.

DP is the technique of choice in the biological context. It involves the calculation of a matrix where one dimension is the text and the other the pattern. By using a cost function which rewards a match between any two characters, and punishes

Table 1. A fragment of BLOSUM62

	A	R	N	D	C	Q	→
A	4	-1	-2	-2	0	-1	
R	-1	5	0	-2	-3	1	
N	-2	0	6	1	-3	0	
D	-2	-2	1	6	-3	0	
C	0	-3	-3	-3	9	-3	
↓							

UNIT COSTS

EDIT COST MATRIX

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

C[0,j] = 0
C[i,0] = i
C[i,j] = if (Pattern[i]== Text[j]) then C[i–1,j–1]
else 1 + min (C[i–1,j], C[i,j–1], C[i–1,j–1])

SELECT MINIMUM COST
in the bottom row

SIMILARITY MATRIX

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	0	1	0	0	0	0	0	0
u	0	0	2	1	0	0	0	0
r	0	0	1	3	2	1	1	0
v	0	0	0	2	2	1	0	0
e	0	0	0	1	1	3	2	1
y	0	0	0	0	0	2	2	3

S[0,j] = 0
S[i,0] = 0
S[i,j] = if (Pattern[i] == Text[j]) then
max (S[i–1,j–1] +1, S[i–1,j] – 1, S[i,j–1] – 1, 0)
else
max (S[i–1,j–1] –1, S[i–1,j] – 1, S[i,j–1] – 1, 0)

SELECT MAXIMUM SIMILARITY
anywhere

Fig. 1. Edit cost and similarity matrices for the comparison of the pattern *survey* with the text *surgery*

a mismatch or a character skip in the text or the pattern, an overall measure of sequence divergence can be calculated. In most theoretical work an edit cost is calculated which defines the number of transformations (inserts, deletes, and replacements) which will mutate the text into the pattern, and unit edit cost functions are used. In many biological contexts it is the similarity function which is of interest, and this is calculated in a similar manner but using a matrix which has a similarity value for each possible match or mismatch, such as the BLOSUM62 matrix [64] shown in Table 1. We illustrate the difference between the edit cost function and the similarity function in Fig. 1, where we show unit cost and similarity functions. The edit and similarity functions are further explained in [44,65,31,60]. The complexity of matrix calcula-

tion is $O(mn)$ and optimisations of the DP calculation which reduce this complexity are known and widely used [54].

Automata can represent the pattern and be used to find portions of text which match it [50]. Representing the text as an automaton moves the problem into the area of text indexing in a database context which is our focus. A suffix tree [73] or a suffix array [47] can be viewed as such an automaton, and the factor oracle is a recent extension of the use of automata in text searching [1]. Our work combines an automaton with the DP calculation.

Bit-parallelism can be used in combination with a pattern automaton. This technique represents the text and the pattern as bit sequences, divides them into computer words, and performs fast comparisons based on register logic, for instance the SHIFT and OR functions [12,15]. It is a challenge to find appropriate logic functions to represent the automaton computation. In comparing biological sequences the cost function itself may be read from a matrix, so that bit parallelism combined with a pattern automaton is currently of limited use. Future work may however change this.

Filtering techniques [67] can be used to focus the search on parts of the text which potentially could harbour a match so that the DP calculation applies to less data and the overall complexity of text comparison is reduced below $O(mn)$. This approach can use different methods of text scanning or partitioning. Filtering used on its own is considered to deliver efficiently only the matches which are very close to the query [54].

These strictly algorithmic approaches to pattern matching are recently being complemented with approaches borrowed from the area of signal processing and data compression. Interesting new avenues have been opened by Kahveci and Singh [40] and Ferragina and Manzini [30]. Kahveci and Singh use the wavelet transform to map genomic strings to their local frequencies for different resolutions. They develop algorithms for both range and nearest neighbour queries and present experimental results for up to 30 Mbp of DNA sequence. Their technique is very promising. It needs to be investigated how to scale it up, and how to deal with gapped alignments. Ferragina and Manzini combine compression with a suffix array data structure and show that the performance of exact matching can be significantly improved. Approximate matching, however, is still a challenge in that context, and their DNA index is not large (4.6 million base pairs).

2.4 Biological applications

Biological applications use different combinations of these methods, and Gusfield [31] provides an in-depth treatment of most areas of biological string processing. We mention applications which are close to our focus of interest. BLAST [3,2] combines a DP calculation with q-grams, filtering, automata, and bit-based comparisons. It is a heuristic approach which cannot guarantee that all the significant matches are reported. BLAST is used very widely in many contexts and has been used extensively in human genome analysis [72,23]. In pattern discovery and gene prediction suffix trees can be used [18,26,48,71], along with other methods. Rocke [63], for instance, combines Gibbs sampling with a suffix tree index in the area of gapped motif discovery. An application of persistent hash position trees which are close relatives of suffix trees was reported in the context of sequence assembly [51]. Persistent suffix arrays have been used in EST analysis [19,52] and transient suffix trees in repeat finding [42]. Considerably less research has been done in the use of suffix trees for protein analysis, with the exception of recent work in the area of protein classification [27]. The constraint in the use of suffix trees so far has been the difficulty of building them on disk in excess of RAM size. We solve this problem and can therefore test the indexing gains on large indexes to both DNA and protein data.

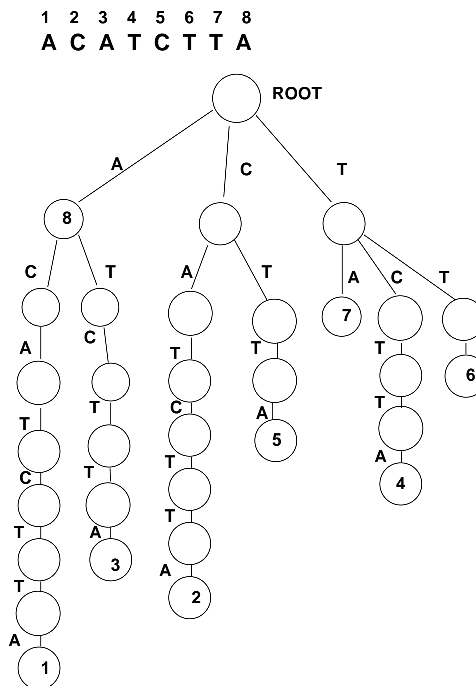

Fig. 2. An example trie on **ACATCTTA**

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.

An example digital trie representing **ACATCTTA** is shown in Fig. 2. 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 Fig. 3.
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 Fig. 3).

Fig. 3. A suffix tree on **ACATCTTA**

Fig. 4. Suffix tree and links on **ACACACAC$**

Fig. 5. A fan-like structure of the partitioned suffix tree, with nine partitions, using prefix length 2

The change from a trie to a suffix tree reduces the storage requirement from $O(n^2)$ to $O(n)$ [73,49,70].

Most implementations of the suffix tree also use the notion of the suffix link [70]. 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" [28]. Suffix links, shown in Fig. 4, 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 disks 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 of DNA sequence. Using the suffix-link-based algorithm under PJama, checkpointing trees indexing more than 21 Mbp has not been possible [33,34] (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 [8,45]). We have therefore investigated incremental construction algorithms in which we forego the guarantee of $O(n)$ complexity.

3.1 Exact matching

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 $O(k+x)$ where k is the query length and x the number of matches in the index. Looking for queries of length k may be expected to bring back a $1/a^k$ fraction of the whole tree, where a is here the size of the active alphabet. For example, a query of length 4 over a DNA index might retrieve $1/256$ of the tree, on average. 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 for exact matching 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 10–12 letters for single human chromosomes. For proteins, with a larger alphabet, much shorter queries can be evaluated efficiently.

4 The new tree 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, and they result in a fan-like tree structure in which partitions can be built either consecutively or in parallel, see Fig. 5. 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 partitioned 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 Phased tree construction

Several $O(n)$, suffix-link-based, tree-building algorithms are known [73,49,70,28,46], but they have not proved appropriate for the large persistent tree construction undertaken by Navarro [56] 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 $O(n \log n)$ *for this application* [66].

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, 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 286 Mbp of DNA 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 [24]. Alternatively, we can assume that, given the pseudo-random nature of DNA, the tree is uniformly populated. To uniformly partition, we calculate

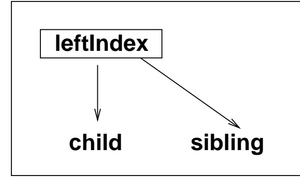

Fig. 6. Node of a thin naive tree

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[9]. 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 $a^l - 1$. Thus, 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 as follows.

```
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
```

A suffix tree node in our implementation consists of three fields: *child* reference, *sibling* reference, and an integer *leftIndex*, shown in Fig. 6. A new node represents a suffix stretching from position i to the end of the text string. It has *null* child and sibling fields, 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 Fig. 7.

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

[9] 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.

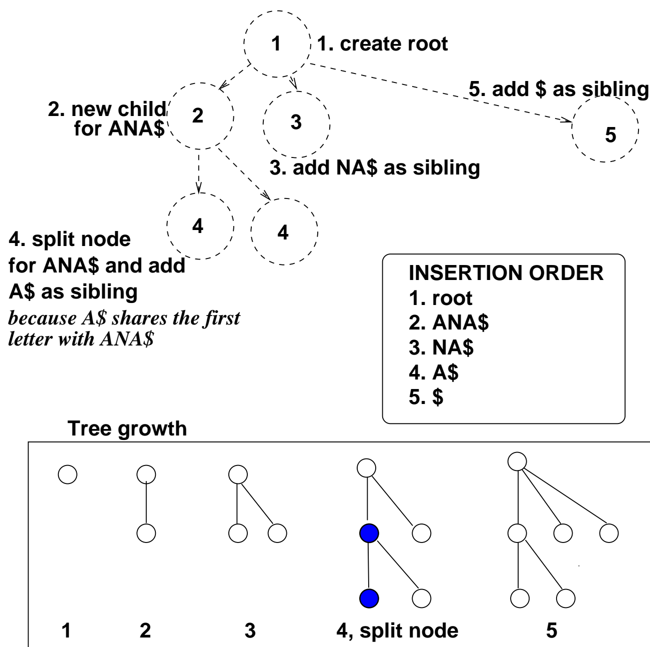

Fig. 7. Tree creation for **ANA$**

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 [45].

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 and proteins remains between $1.6n$ and $1.8n$, where n is the length of the sequence, giving an expectation of between 58 bytes and 65 bytes per node. Some of this space may well be free space in partitions, and some is used for housekeeping [61]. 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. However, further compression could be obtained by using techniques similar to those proposed by Kurtz [41].

5 Tree building and exact matching in practice

In this section we summarise our experiments in tree building and exact matching on DNA strings. Results for protein data were analogous, and we do not report them here. We used DNA from six single chromosomes of the worm *C. elegans*, of 20.5 Mbp maximum[10] and some 286 Mbp merged DNA from human chromosomes 21, 22, and 1[11]. As queries we used short sequences, from the STS division of Entrez[12] for human data, and for the worm queries, short sequences called cDNAs. From each sequence initial characters were taken to be used as query strings.

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, see Sect. 5.1, which is derived from Java 1.2 and uses JIT, for the persistence measurements. All timing measurements were obtained using SunOS 5.7 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.

The total number of lines of Java code for the five data structures examined was 3,216, which includes over 10% lines of comments and print statements. The naive tree accounts for less than 550 lines of Java code.

5.1 The persistence platform

The first set of experimental trials of our algorithms was conducted using the PJama[13] platform [8,6,9,32,10,7,39,62,61]. We selected PJama to minimise the software engineering cost of our experiments. 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, using a different command-line parameter for PJama indicating whether a persistent store is being used.

Although tuned, purpose-built mechanisms will 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 payoffs in reduced implementation times and more rapid deployment.

We are investigating other persistence mechanisms, including an object-oriented database, Gemstone/J[14], 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 allow rapid experiments with a variety of index structures and matching algorithms.

5.2 Tree construction in memory

We compared two versions of the tree built using Ukkonen's algorithm [70], two versions of the naive tree of our construction, and the suffix binary search tree [35,36]. The two trees constructed using Ukkonen's algorithm, time complexity of

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

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

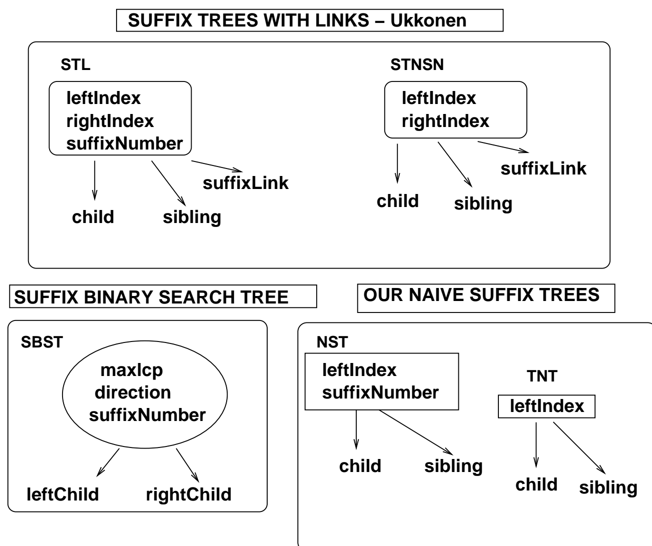

Fig. 8. Transient indexes built in memory

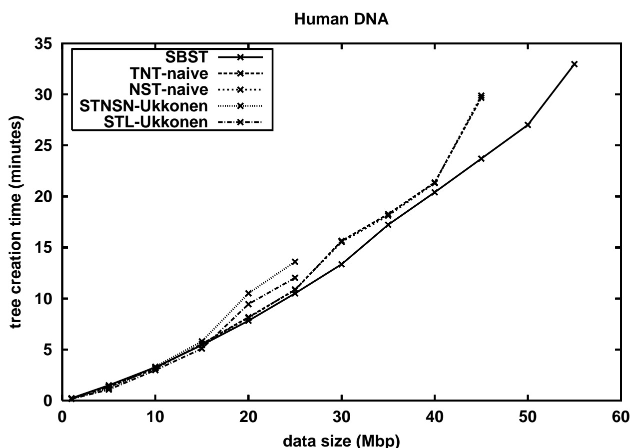

Fig. 9. Time required to build an index for human DNA; the graph for TNT is superimposed on the graph for NST

$O(n)$, differed only in one detail: one of them did not explicitly store the suffix number. The two naive trees, time complexity of $O(n^2)$, had an analogous structural relationship: no suffix number in one of the trees. The suffix binary search tree [35,36] which builds in $O(n \log n)$ time had nodes consisting of a left child, right child, suffix number, maximum longest common prefix, and a direction bit. It had the smallest overall space requirement, because per each text character only one node is needed, while the suffix tree needs up to two nodes per text character. We show the node layout of the structures we investigated in Fig. 8, and the comparison of index build times in Fig. 9. Surprisingly, tree creation times seem to be influenced predominantly by the space complexity of the data structure, and there appears to be no difference between linear and worse than linear construction algorithms. Our data for protein trees (not shown) exhibit the same behaviour, except that protein suffix trees are slightly more compact (have fewer nodes) and the suffix trees and the SBST take a little longer to construct.

Table 2. Cold store, a batch of 10,000 exact queries over 20.5 Mbp of worm DNA using an $O(n)$ index

Query length	Avg time per query (ms)	Total hits per batch
8	920	8 568 303
9	263	2 553 520
10	142	758 523
15	36	3687
50	34	394
100	34	305
200	33	107

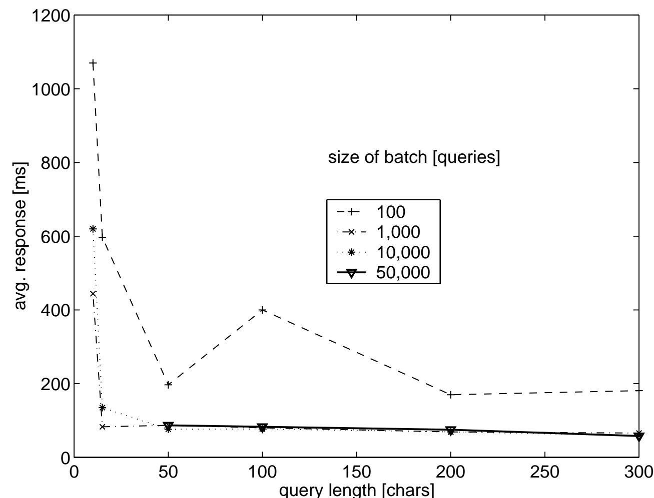

Fig. 10. Cold store query performance, using 286 Mbp of indexed DNA

5.3 A small persistent tree

We carried out tests with our implementation of the $O(n)$ tree-building algorithm [70]. A tree for 20.5 Mbp of DNA was created in memory in 7 min on average. However, on disk, the creation time was around 34 h, and checkpoints at 12 million and then every 0.5 million nodes were required. We used a 2 GB log file, and one store file of 2 GB. This was the largest tree of this type that we could build. It fitted mostly in memory (2 GB RAM, 2 GB store, some space needed for the JVM). Table 2 shows the results obtained for a batch of 10,000 exact matching queries run on a cold store.

5.4 A large persistent tree

We then indexed 286 Mbp of DNA using the new suffix tree construction algorithm presented in this paper. The store required a 2 GB log and 19 GB in files of 2 GB maximum. Store creation time was 19 h in our first run, and later 13.5 h. Queries of the same length were sent in batches, without the use of multithreading[15].

We carried out exact string matching experiments on a cold store, see Fig. 10, and on a warm store, see Fig. 11. We

[15] In other experiments [34], we have demonstrated a significant speedup by using multiple threads to handle a batch of queries over a forest of suffix binary search trees.

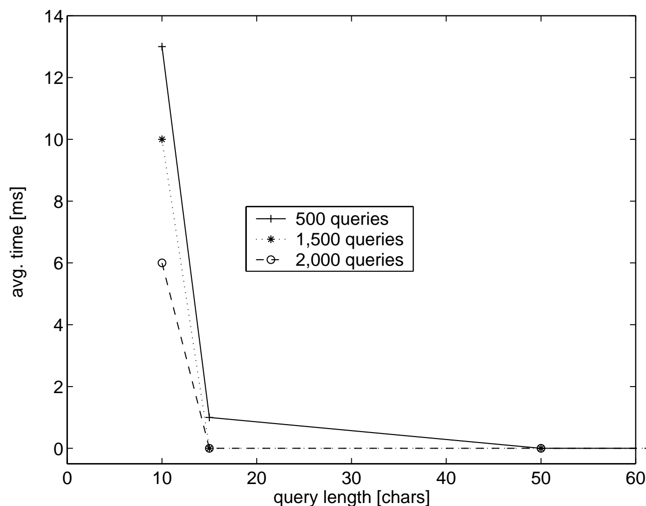

Fig. 11. Queries run over a warm store, using 286 Mbp of indexed DNA

Table 3. Cold store, comparison of batches of queries of length 10 and 50 evaluated over a persistent index to 286 Mbp of DNA

Batch size	Query length	avg time per query (ms)	Total hits per batch
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	25 376

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. In particular, the performance of a batch of 100 queries on a cold store shows that small batches may be inappropriate in this context, and performance is likely to fluctuate. This perceived irregularity may also be due to the interaction between PJama, the garbage collector, and the cache eviction algorithm.

Table 3 demonstrates why the short queries take a long time to return results. The time of query evaluation 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, many results are reported and the reporting time dominates because of slower access to secondary memory. For longer queries, fewer results are found, and the average query response improves. The observed slow performance for large result sets in our $O(n)$ tree (Table 2) is partly due to the use of the Java.lang.Vector class to store the matches. Our large tree does not suffer from this problem, because we only count the number of matches found.

6 Approximate matching algorithm

Approximate searching in a suffix tree has been traditionally optimised based on the tree with suffix links (see the work by Ukkonen [69] and Cobbs [22]). As the DP calculation is carried out in a suffix tree with links, the last column of the alignment matrix is preserved along with a reference to the node it applies to. Following suffix links one can exclude from calculation all shorter suffixes. The space overhead of this optimisation is considerable as a record of nodes visited and the relevant matrix column have to be kept, and suffix links are needed. Baeza-Yates and Navarro [14] show that Cobbs' implementation of his method is slower than the depth-first tree traversal which they implement.

The approach we take follows [13,14], and is closely related to that of [63]. [14] assumes unit edit costs and uses a suffix array to simulate a suffix tree. A long query is broken into smaller substrings which are then used as individual queries, and when the index returns potential matches those will have to assembled into longer string alignments. The top of the suffix tree is scanned depth-first down to a maximum string length $k + m$ where k is the number of errors allowed in each small fragment, and m is the query length (a part of the original query). The DP calculation comparing the indexed text and the query is carried out. The minimum depth of traversal is $m-k$. After traversing $m-k$ characters if the edit cost is still 0, we have matched the query with k errors and can output a match. As consecutive columns are being calculated, if the edit cost becomes too high, the calculation can stop early. The space overhead of this traversal is small, as the DP matrix has size

$$(m + 1) \times (m + k + 1)\ .$$

The matrix is used as a stack with the index pointing to the current string depth of a node as measured from the root.

Our work builds on this algorithm but we make the following restrictions and changes. We use a suffix tree and not a suffix array and build a much larger index. We do not consider an edit function but a similarity function [65] so that our solution can be extended to deal with non-unit costs and gaps (i.e., an "an arbitrary score matrix" [63]). This precludes the use of fast bit arithmetic or an automaton for the pattern [12, 13]. We do not use filtration yet, as more work is required to refine this approach for use with arbitrary cost matrices. We implement the optimisation suggested by the authors which explores only the children of the root which start with the first $k+1$ letters of the query. Our main interest is in measuring the actual gain from using the suffix tree as opposed to carrying out the full matrix calculation.

We redefine the problem as follows: given the pattern of length m find all pattern occurrences which reach the threshold t, given a similarity function (our current function is $+1$ for match and -1 for mismatch or character skip). This implies using a matrix of size $(m + 1) \times (2m - t + 1)$.

The suffix tree is traversed depth first and the DP calculation carried out using a rectangular matrix. The row and column zero are filled with nulls, as we are interested in finding local alignments. An index points at a matrix column and reflects current distance from the root in characters. The matrix is evaluated column-wise. Three conditions limit the depth of the tree traversal (and matrix calculation). These are:

1. Break the DP calculation whenever the required similarity threshold is reached, find matches by traversing children.
2. Stop calculating the matrix whenever we are certain that the current calculation will never reach the threshold.
3. Break on reaching a separator or terminator character.

Condition 2 which is evaluated after each matrix column has been calculated is expressed as follows. *currentTextIndex* is the array column number (equivalent to the distance in characters from the root). $2m - t$ is the last text position in the matrix, and we access the maximum score which was calculated in the current column, *maximumScoreInColumn*.

```
if currentTextIndex >= (2m - t) - t
then if (maximumScoreInColumn <=
        currentTextIndex - (2m - t) + t
        return
     endif
endif
```

This condition, will have to be refined for use with similarity functions used in biology, like the protein cost matrices PAM or BLOSUM [25,64].

A full traversal of a suffix tree, down to the depth $2m - t$, could lead to performing more DP calculations than needed for the evaluation of the matrix spanning the entire text, i.e., $O(mn^2)$ (or $O(mn \log n)$ if we consider the pseudo-randomness of DNA data). This issue has been approached from the point of view of theory [14,63] but our work concentrates on the engineering aspect and clearly distinguishes the gains from indexing from the effects of efficient matrix calculation or the possible loss of speed due to the use of a disk-resident index. We aim to discover what query and threshold lengths are appropriate so that we can guarantee that significantly fewer matrix columns are calculated than would be needed otherwise. To this aim we experiment with both DNA and protein indexes and with different query lengths and similarity thresholds.

7 Approximate matching results

7.1 A transient protein tree

Our first test concerns a transient naive tree indexing 36 Mb of protein, i.e. the entire SWISSPROT database. We use human genes as queries, and break them into strings of 5–11 characters. We measure the number of DP matrix columns calculated during query evaluation. We take the *maximum* observed number of columns calculated in a given query and threshold combination and derive the *maximum* of the observed ratios relative to the DP matrix for the text of length 36 mln. Our results based on a sample of 1,425 queries of varying lengths and thresholds are summarised in Table 4. We observe that combinations of query length m and a threshold $m - 1$ or $m - 2$ deliver a speedup in comparison to full matrix calculation. A threshold value of $m - 1$ (shown in bold) delivers a high efficiency gain and we observe that $less\ than$ 1% of the full matrix is being evaluated. For the threshold of $m - 2$ the speedup is limited. We notice approximately two orders of magnitude ratio between the number of columns reported for $m - 2$ and $m - 1$. Thresholds of $m - 3$ and $m - 4$ increase the size of the matrix calculation.

Table 4. The fraction of 36 mln columns calculated for a range of thresholds and query lengths over a transient suffix tree index for 36 Mb of protein sequence

Threshold	Query	Ratio
4	5	**0.0015**
4	6	0.2489
5	6	**0.0022**
4	7	2.5200
5	7	0.3222
6	7	**0.0033**
4	8	6.9729
5	8	2.8402
6	8	0.3770
7	8	**0.0039**
6	9	2.9439
7	9	0.4234
8	9	**0.0052**
7	10	3.4541
8	10	0.5437
9	10	**0.0062**
9	11	0.5786
10	11	**0.006**

7.2 A persistent protein index

A more significant indexing gain obtains for the persistent tree indexing 200 Mb of protein (all of SWISSPROT and TREMBL data) which we now present. This data is based on the evaluation of 10–15 queries for each combination of query length and threshold, and the *maximum* observed number of columns evaluated divided by 200 mln, shown in Table 5. The total number of queries executed over this data set was 312. In a larger tree at the threshold equal to $m - 1$ (in bold) an even smaller portion, $less\ than$ 0.3%, of the full DP matrix is evaluated. For the threshold of $m - 2$ the indexing gain is slightly more significant as well. This leads us to believe that for larger protein indexes the threshold of $m - 2$ may also be beneficial in practice.

7.3 A persistent human DNA index

Table 6 presents the indexing gain for a DNA suffix tree indexing 286 Mbp, i.e., 10% of the human genome, where both human and yeast DNA sequences[16] were used as queries. Short DNA queries report too many matches to be of use in sequence searching and we do not show them here. The results are based on 1,334 queries of varying lengths and thresholds. We find that indexing DNA pays off significantly, and queries with the threshold of $m - 2$ reduce the size of the DP calculation to *less than 1%* of the original matrix, while queries with just one mismatch (shown in bold), i.e., threshold equal to $m - 1$, evaluate between 0.01% and 0.09% of the matrix. It appears that this behaviour holds irrespective of the origins of the DNA in the index and in the query. The threshold of $m - 3$ (shown for query length 14, threshold 11) also offers some reduction in matrix size, but may bring back too many matches.

These results demonstrate clearly that the potential of suffix tree indexing might be considerable by delivering the benefits of the full DP calculation at a reduced cost. As the DP

[16] We used yeast chromosome 1 from ftp://genomeftp.stanford.edu/pub/yeast/.

Table 5. Ratio of columns calculated to 200 mln, based on a suffix tree indexing 200 Mb of protein data

Threshold	Query	Ratio
3	5	0.0474
4	5	**0.0003**
4	6	0.0697
5	6	**0.0004**
5	7	0.0807
6	7	**0.0006**
6	8	0.1088
7	8	**0.0007**
6	9	1.4185
7	9	0.1274
8	9	**0.0010**
7	10	1.6888
8	10	0.1606
9	10	**0.0012**
9	11	0.1830
10	11	**0.0013**
10	12	0.2035
11	12	**0.0016**
10	13	2.4061
11	13	0.2260
12	13	**0.0022**
12	14	0.2636
13	14	**0.0024**
13	15	0.3012
14	15	**0.0026**
14	16	0.3166
15	16	**0.0029**

Table 6. Ratio of MAX columns calculated to 286 mln, based on a suffix tree for 286 Mbp indexing human genomic DNA and human and yeast DNA queries

Threshold	Query	Yeast ratio	Human ratio
7	8	**0.0001**	**0.0001**
8	9	**0.0001**	**0.0001**
8	10	0.0029	0.0027
9	10	**0.0001**	**0.0001**
9	11	0.004	0.003
10	11	**0.0002**	**0.0002**
10	12	0.0059	0.0055
11	12	**0.0003**	**0.0003**
11	13	0.0076	0.0073
12	13	**0.0003**	**0.0003**
11	14	0.1103	0.121
12	14	0.0111	0.0104
13	14	**0.0004**	**0.0004**
13	15	0.0129	0.0116
14	15	**0.0005**	**0.0006**
15	16	**0.0007**	**0.0008**
16	17	**0.0008**	na
17	18	**0.0009**	na

calculation generally dominates the time needed to perform a search, these data point to the fact that further work in this direction might lead to the development of a more efficient solution to approximate pattern matching.

7.4 Indexing performance

We consider further issues relevant to the performance of index-based approximate matching.

- Is the prototype index implementation fast enough and how does it compare with carrying out the same DP calculation in memory without an index?
- Which of the query and threshold combinations deliver manageable numbers of matches, as all matches for a longer query broken into smaller parts will have to be merged?

We now address these questions. Our performance comparisons are calculated as follows: we measure the query execution time and divide it by the size of the DP matrix which represents the product of the text and query. The formula we use is

$$time\ (sec)\ /\ (text\ (Mb) \times query)\ .$$

We use two benchmarks. One is BLAST which is optimised for use with multiprocessor machines. Running BLAST on the same data set and queries (using protein data) we measure the size of the data processed (query size × database size in Mb, referring to uncompressed data sizes) and the time needed to process the query using 4 processors. Analysis of several runs on our computer, using protein data, yields the average ratio of time in seconds to matrix size to be:

$$2.39 \times 10^{-7}\ sec\ /\ Mb\ .$$

The other benchmark is a full DP matrix calculation, using the same data and matrix evaluation software (but without a suffix tree) and one processor (as in our suffix tree tests). To calibrate the DP calculation in memory we used a circular buffer twice the query length and Java version 1.3. This yielded the equation for the same relationship as being

$$time\ (sec) = \mathbf{1.16} \times matrixSize\ (Mb) + 66.0\ .$$

As expected, the performance difference between BLAST and an unoptimised full matrix calculation is several orders of magnitude. The purpose of the benchmarks is first to measure the gain of our unoptimised prototype implementation against a similarly unoptimised matrix calculation, i.e., we are comparing like to like, with both implementations built under similar assumptions in Java. The second benchmark, BLAST, serves as a comparison with a fully tuned software program which was developed over the years by a large team of specialists, and uses most of the known optimisations.

To calibrate the size of the result set returned by the query, we imagine a hypothetical query of 300 characters broken into short queries. We can then calculate the number of matches returned for a query of length 300. As our data have a symmetrical distribution with no outliers, we use the $average$ number of results, and $query$ length, and obtain the expected result set size as

$$matches = averageResultSize \times 300\ /\ query\ .$$

Table 7. Average expected number of partial matches for a query of length 300, and time to DP matrix size ratio in seconds per Mb for the index to 200 Mb of protein

Threshold	Query	Matches	Sec:Mb
3	5	15 610 540	1.8536
4	5	446 953	0.0187
4	6	679 267	1.2842
5	6	18 150	0.0178
5	7	52 780	0.9954
6	7	1719	0.0179
6	8	2087	0.9773
7	**8**	**258**	**0.0103**
7	9	405	0.8622
8	**9**	**136**	**0.0137**
8	10	273	0.9787
9	10	147	0.0475
9	11	164	0.8038
10	11	67	0.0223
10	12	244	0.8529
11	12	128	0.0352
11	13	185	0.7919
12	13	87	0.0401
12	14	133	0.5443
13	14	62	0.0504
13	15	136	0.5683
14	15	60	0.0468
14	16	135	0.6819
15	16	65	0.0604

Table 8. Average expected number of partial matches for a query of length 300, and time to DP matrix size ratio in seconds per Mb, for the index to 286 Mbp of human DNA

Threshold	Query	Human matches	Human sec:Mb	Yeast matches	Yeast sec:Mb
7	8	2 212 283	0.0074	2 291 426	0.0064
8	9	832 903	0.0029	702 183	0.0023
8	10	2 018 651	0.0442	1 617 593	0.035
9	10	51 945	0.0024	179 303	0.0011
9	11	881 884	0.0322	599 274	0.0382
10	11	179 075	0.0015	45 276	0.001
9	12	1 172 425	0.9131	na	na
10	12	434 412	0.0448	110 891	0.0348
11	12	154 098	0.0021	9846	0.001
10	13	448 850	1.0419	na	na
11	13	130 782	0.0443	52 516	0.0386
12	**13**	**85 296**	**0.003**	**74 928**	**0.0012**
11	14	242 862	0.9658	221 089	0.7255
12	14	65 970	0.0459	7485	0.055
13	**14**	**58 277**	**0.003**	**5041**	**0.0011**
12	15	676 255	0.896	na	na
13	15	216 599	0.0549	1820	0.05
14	**15**	**53 030**	**0.0039**	**534**	**0.0009**
14	16	13 7673	0.0624	na	na
15	16	49 543	0.004	169	0.001
16	17	na	na	54	0.0009
17	18	na	na	0	0.0013

We present the expected number of hits for a query of 300 characters and the ratio of time (seconds) to Mb (query size × database size) in Table 7, for the protein data set of 200 Mb.

In Table 8 we present the analogous results for the DNA index over 286 Mbp of the human genome and human and yeast queries. A comparison of both tables reveals that the performance of the protein index is much slower than the performance of the DNA index. This mirrors the difference in the indexing gain observed for both data sets, and is related to the topology of both indexes, which reflects two different alphabets. We believe that the relationship between the alphabet size, the size of the data set, and the indexing gain requires further analysis, and might benefit from large scale tests. One could gather statistics on the number of nodes present at each string depth, as measured from the root, and compare them with the actual number of nodes visited for all the combinations of query length and threshold under different cost models.

We focus first on the speed of calculation shown in both tables. We knew that reaching the speed of BLAST would not be possible, first because we are using Java, and second because of the unoptimised matrix calculation. On the other hand, the comparison with the unoptimised DP calculation shows considerable efficiency gains. Current measurements indicate that the persistent platform always delivers a faster match for the threshold of $m - 1$ for both DNA and proteins. The threshold of $m - 2$ which offers a speedup in DNA matching is currently not attractive for proteins, but might become so for larger indexes.

We now look at the size of the returned result set. This is a significant consideration, as partial hits will have to be merged in search for longer string alignments. We assume here that dealing with results sets larger than 100,000 hits is not feasible, and base our considerations on a hypothetical query of 300 characters. It turns out that for the current size of SWISSPROT and TREMBL databases the minimum viable query length is around seven characters, and this guarantees fewer than 100,000 partial hits. Probably, with query length 8 and thresholds 6 and 7, the number of hits returned will be easier to process. However, the indexing gain for query length 8 and threshold 6 is modest, so threshold 7 and query length 8 or threshold 8 and query length 9 seem to be viable options (shown in bold). For longer queries too few matches are returned, and important similarities could be missed. For the DNA index we tested, useful query and threshold values are query lengths 13, 14, and 15 (shown in bold). The indexing gain is satisfactory, i.e., 0.03–0.06% of the matrix is executed. The number of partial hits is then below 100 000, and the speed seems to be satisfactory for a prototype implementation. For longer queries the indexing gain is more significant but the sensitivity will decrease.

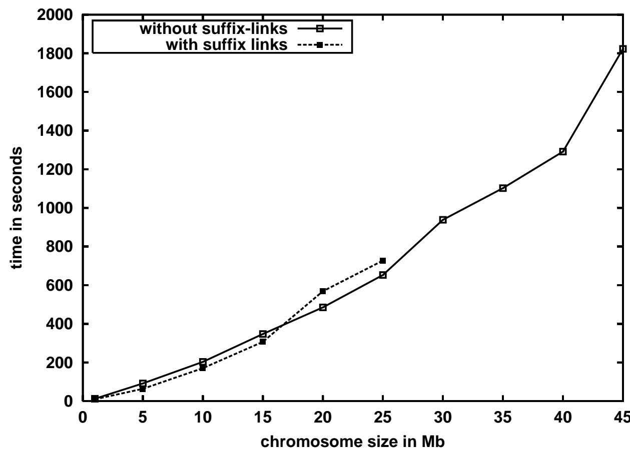

Fig. 12. Tree creation in memory

8 Discussion

We first discuss the new algorithm for the tree construction, and then turn our attention to the approximate matching problem. 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 Fig. 12 we show *in-memory* performance comparison of suffix trees with and without suffix links. We show a close-up of tree creation times for two trees: a modified version of Ukkonen's algorithm [70] 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).

The incremental partitioned construction algorithm uses a partition size which we select. So far, our experiments suggest that this should correspond to between 5 Mbp and 20 Mbp. This means that we are building the tree in a region where the $O(n)$ suffix-link algorithm offers no significant advantage.

The comparison of unoptimised *persistent* tree building times shows that our algorithm outperforms the suffix-link tree in terms of size, and we believe that building times in the region of 5 h 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.

The approximate matching algorithm which we implemented and tested with large biological data sets shows clearly that indexing of sequence suffixes is beneficial, and can significantly reduce the size of the DP matrix calculation needed in query evaluation. We have no access to similar results for any of the known filtering approaches, including BLAST, and cannot make an appropriate comparison. Our approach combines filtering and the similarity calculation in one step, and this might also make a comparison with other approaches difficult, until we develop a fully optimised data structure, a well-tuned database layer, and a robust implementation of the matrix calculation.

The actual performance of the tree compares well with the analogous matrix calculation in Java, and future speedups can be delivered by using a combination of optimisation techniques which we are investigating. Those may include compression of the data structure itself (and departure from our simple object-oriented implementation), data clustering techniques, caching techniques and algorithmic optimisations.

The BLAST implementation we used as a benchmark shows how far our prototype is from becoming a product, and that further research is needed. We believe that implementing BLAST on top of a suffix index, for large datasets like the human genome, could deliver both faster searches, and, more significantly, would require fewer CPUs to carry them out. Our work clearly demonstrates that indexing may bring expected benefits in the area of biological searching.

9 Future work

Future work can be divided into the following interrelated parts:

- Improvements to the tree representation (data structure compression) and to the incremental construction algorithm.
- Investigation of the interaction between approximate matching algorithms and disk-based suffix trees.
- Investigation of alternative persistent storage solutions.
- Refinement of the cut-off used in the depth traversal of the tree, reflecting different similarity functions and gap costs.
- Post-processing of results to build longer alignments including gap costs and overall similarity measures.
- 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 [45, 10]. This will enable us to operate on other hardware and operating system platforms and to verify that the phenomena so far observed are not artifacts 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 adopting biological measures of sequence similarity [3,65]. A full investigation of all cost matrices will have to follow, so that the usefulness of our index in the biological context can be ascertained. This may require modifications to the way we calculate the cut-off point for the depth of tree traversal, and may lead to different figures for the indexing gain, in response to different cost matrices.

We are also considering alternative persistence implementations. 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, genome maps, protein structure, protein function, and gene 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 [23,72], and it would necessitate up to

$$2\binom{40000}{2}$$

gene alignments to perform full gene comparison within the human genome which has around 40,000 genes[17]. 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 [31].

10 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.

It has also been demonstrated that large suffix trees can reduce the required DP matrix calculation to a small fraction of the whole. Both DNA and protein data sets benefit from indexing, and this benefit increases with the size of the index. This means that for genomic data sets indexing can potentially deliver faster query processing. Much further experimentation and analysis is required to develop confidence in these early, but intriguing results.

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

Acknowledgements. Ela Hunt's research was partly funded by the Medical Research Council of UK, via a special research training fellowship in bioinformatics. We would like to acknowledge the contribution of the members of the PJama team at Glasgow and SUN Labs. We thank the anonymous referees, Paolo Ferragina, and Ricardo Baeza-Yates for their comments.

References

1. C. Allauzen, M. Crochemore, M. Raffinot. Factor oracle: a new structure for pattern matching. In: SOFSEM'99, Lecture Notes in Computer Science, vol. 1725. Springer, Berlin Heidelberg New York, 1999, pp. 291–306
2. S.F. Altschul, T.L. Madden, A.A. Schaeffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25:3389–3402, 1997
3. S.F. Altschul et al. Basic local alignment search tool. J Mol Biol 215:403–10, 1990
4. A. Andersson, N.J. Larsson, K. Swanson. Suffix trees on words. Algorithmica 23(3):246–260, 1999
5. A. Andersson, S. Nilsson. Efficient implementation of suffix trees. Software Pract Exp 25(2):129–141, 1995
6. M. Atkinson, M. Jordan. Providing orthogonal persistence for Java. In: ECOOP98, Lecture Notes in Computer Science, vol. 1445. Springer, Berlin Heidelberg New York, 1998, pp. 383–395
7. M.P. Atkinson. Persistence and Java – a balancing act. In: ECOOP Symposium on Objects and Databases, Lecture Notes in Computer Science, vol. 1944. Springer, Berlin Heidelberg New York, 2000, pp. 1–32
8. M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, S. Spence. An orthogonally persistent Java. ACM Sigmod Rec 25(4):68–75, 1996
9. M.P. Atkinson, M.J. Jordan. Issues raised by three years of developing PJama. In: ICDT99, Lecture Notes in Computer Science, vol. 1540. Springer, Berlin Heidelberg New York, 1999, pp. 1–30
10. M.P. Atkinson, 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 Laboratorie and Dept. Computing Science, Univ. Glasgow, UK, 2000
11. M.P. Atkinson, R. Welland (eds). Fully integrated data environments. Springer, Berlin Heidelberg New York, 1999
12. R. Baeza-Yates, G.H. Gonnet. A new approach to text searching. Commun ACM 35:74–82, 1992
13. R. Baeza-Yates, G.H. Gonnet. Fast text searching for regular expressions or automaton simulation over tries. J ACM 43(6):915-936, 1996
14. R. Baeza-Yates, G. Navarro. A hybrid indexing method for approximate string matching. J Discrete Algorithms 2000. To appear
15. R. A. Baeza-Yates, G. Navarro. Faster approximate string matching. Algorithmica 23(2):127–158, 1999
16. P. Bieganski. Genetic sequence data retrieval and manipulation based on generalised suffix trees. PhD thesis, University of Minnesota, USA, 1995

17. A. Blumer, J. Blumer, D. Haussler, R. McConnell, A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. J ACM 34(3), 1987
18. A. Brazma, I. Jonassen, J. Vilo, E. Ukkonen. Predicting gene regulatory elements in silico on a genomic scale. Genome Res 8:1202–1215, 1998
19. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, M. Vingron. q-gram based database searching using a suffix array. In: RECOMB99, pp. 77–83. ACM, New York, 1999
20. D.R. Clark. Compact pat trees. PhD thesis, University of Waterloo, 1996
21. D.R. Clark, J.I. Munro. Efficient suffix trees on secondary storage (extended abstract). In: Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 383–391, Atlanta, Ga., 28–30 January, 1996
22. A.L. Cobbs. Fast approximate matching using suffix trees. In: CPM95, Lecture Notes in Computer Science, vol. 937. Springer, Berlin Heidelberg New York, 1995, pp. 41–54
23. International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. Nature 409:860–921, 2001
24. T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to algorithms. MIT, Boston, Mass., 1990
25. M. Dayhoff, R. Eck, C. Park. A model of evolutionary change in proteins. In: M. Dayhoff (ed) Atlas of Protein Sequence and Structure, vol. 5. National Biomedical Research Foundation, Silver Springs, Md., 1972
26. A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg. Alignment of whole genomes. Nucleic Acids Res 27:2369–2376, 1999
27. B. Dorohonceanu, C.G. Nevill-Manning. Accelerating protein classification using suffix trees. In: Proc. International Conference on Intelligent Systems for Molecular Biology ISMB00, pp. 128–133, 2000
28. M. Farach, P. Ferragina, S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In: FOCS98, pp. 174–185, 1998
29. P. Ferragina, R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. J ACM 46(2):236–280, 1999
30. P. Ferragina, G. Manzini. Opportunistic data structures with applications. In: Proc. 41st Annual Symposium on Foundations of Computer Science. 12–14 November, 2000, Redondo Beach, California, pp. 390–398. IEEE Computer, New York
31. D. Gusfield. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University, Cambridge, UK, 1997
32. C.G. Hamilton. Recovery management for sphere: recovering a persistent object store. Technical Report TR-1999-51, University of Glasgow, Department of Computing Science, Glasgow, UK, December 1999
33. E. Hunt. PJama Stores and suffix tree indexing for bioinformatics applications. 10th PhD Workshop at ECOOP00, 2000. Available at: http://www.inf.elte.hu/∼phdws/timetable.html
34. E. Hunt, R.W. Irving, M.P. Atkinson. Persistent suffix trees and suffix binary search trees as DNA sequence indexes. Technical report, Univ. of Glasgow, Dept. of Computing Science. TR-2000-63, 2000. Available at: http://www.dcs.gla.ac.uk/∼ela
35. R.W. Irving, L. Love. The suffix binary search tree and suffix AVL tree. Technical Report TR-2000-54, Univ. of Glasgow, Dept. of Computing Science, 2000. Available at: http://www.dcs.gla.ac.uk/research/algorithms/sbst/publications/SBST_report.ps
36. R. W. Irving, L. Love. Suffix Binary Search Trees and Suffix Arrays. Technical Report TR-2001-82, University of Glasgow, Department of Computing Science, 2001. Available at: http://www.dcs.gla.ac.uk/research/algorithms/sbst/publica'tions/SA_report.ps
37. H.V. Jagadish, N. Koudas, D. Srivastava. On effective multi-dimensional indexing for strings. In: ACM SIGMOD Conference on Management of Data, pp. 403–414, 2000
38. M.J. Jordan, M.P. Atkinson (eds). Second International Workshop on Persistence and Java. Number TR-97-63 in Technical Report. Sun Microsystems Laboratories, Palo Alto, Calif., USA, 1997
39. M.J. Jordan, M.P. Atkinson. Orthogonal Persistence for the Java Platform — specification. Technical Report SML 2000-94, Sun Microsystems Laboratories, Palo Alto, Calif., USA, 2000
40. T. Kahveci, A.K. Singh. An efficient index structure for string databases. In: VLDB'01, pp. 351–360. Morgan Kaufmann, San Francisco, Calif., 2001
41. S. Kurtz. Reducing the space requirement of suffix trees. Software Pract Exp 29:1149–1171, 1999
42. S. Kurtz, C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. Bioinformatics 15(5):426–427, 1999
43. N.J. Larsson. Structures of string matching and data compression. PhD thesis, Department of Computer Science, Lund University, 1999
44. V.I. Levenstein Binary codes capable of correcting insertions and reversals. Sov Phys Dokl 10:707–10, 1966
45. B. Lewis, B. Mathiske, N. Gafter. Architecture of the PEVM: a high-performance orthogonally persistent java virtual machine. In: 9th Intl Workshop on Persistent Object Systems. TR-2000-93, 2000. Available at: http://research.sun.com/research/-techrep/2000/abstract-93.html
46. M.G. Maass. Linear bidirectional on-line construction of affix trees. In: CPM2000, Lecture Notes in Computer Science, vol. 1848. Springer, Berlin Heidelberg New York, 2000, pp. 320–334
47. U. Manber, G. Myers. Suffix arrays: a new method for on-line string searches. SIAM J Comput 22(5):935–948, 1993
48. L. Marsan, M.-F. Sagot. Extracting structured motifs using a suffix tree – algorithms and application to promoter consensus identification. In: RECOMB00, pp. 210 – 219. ACM, New York, 2000
49. E. M. McCreight. A space-economic suffix tree construction algorithm. J ACM 23(2):262–272, 1976
50. R. McNaughton. Elementary computability, formal languages, and automata. Prentice-Hall, Reading, Mass., USA, 1982
51. H.W. Mewes, K. Heumann. Genome analysis: pattern search in biological macromolecules. In: Lecture Notes in Computer Science, vol., volume 937. Springer, Berlin Heidelberg New York, 1995, pp. 261–285
52. C. Miller, J. Gurd, 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
53. J.I. Munro, V. Raman, S.S. Rao. Space efficient suffix trees. J Algorithms 39:205–222, 2001
54. G. Navarro. A guided tour to approximate string matching. ACM Comput Surv 33:1:31–88, 2000
55. G. Navarro, R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. CLEI Electron J 1(2), 1998
56. G. Navarro, R. Baeza-Yates. A new indexing method for approximate string matching. In: CPM99, Lecture Notes in Computer Science, vol. 1645. Springer, Berlin Heidelberg New York, 1999, pp. 163–185

57. G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio. Indexing methods for approximate text retrieval. IEEE Data Eng Bull 24(4):19–27, 2001
58. G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio. Indexing text with approximate q-grams. In: CPM2000, Lecture Notes in Computer Science, vol. 1848. Springer, Berlin Heidelberg New York, 2000, pp. 350–365
59. W.R. Pearson, D.J. Lipman. Improved tools for biological sequence comparison. Proc Natl Acad Sci USA 85:2444–8, 1988
60. P.A. Pevzner. Computational molecular biology: an algorithmic approach. MIT, Cambridge, Mass., USA, 2000
61. T. Printezis. Management of long-running high-performance persistent object stores. PhD thesis, Dept. of Computing Science, University of Glasgow, Glasgow, UK, 2000
62. T. Printezis, M.P. Atkinson. An efficient object promotion algorithm for persistent object systems. Software Pract Exp 31(10):941–981, 2001
63. E. Rocke. Using Suffix trees for gapped motif discovery. In: CPM00, Lecture Notes in Computer Science, vol. 1848. Springer, Berlin Heidelberg New York, 2000, pp. 335–349
64. S. Henikoff, J.G. Henikoff. Amino acid substitution matrices from protein blocks. Proc Natl Acad Sci USA 89(22):10915–9, 1992
65. T.A. Smith, M.S. Waterman. Identification of common molecular subsequences. J Mol Biol 284, 1981
66. W. Szpankowski. Asymptotic properties of data compression and suffix trees. IEEE Trans Inf Theory 39:5:1647–1659, 1993
67. J. Tarhio, E. Ukkonen. Boyer-Moore approach to approximate string matching. In: J. R. Gilbert, R. Karlsson (eds) Proc. 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, vol. 447. Springer, Berlin Heidelberg New York, 1990, pp. 348–359
68. E. Ukkonen. Approximate string matching with q-grams and maximal matches. Theoret Comput Sci 92(1):191–212, 1992
69. E. Ukkonen. Approximate string matching over suffix trees. In: CPM93, Lecture Notes in Computer Science, vol. 684. Springer, Berlin Heidelberg New York, 1993, pp. 228–242
70. E. Ukkonen. On-line construction of suffix-trees. Algorithmica 14(3):249–260, 1995
71. A. Vanet, L. Marsan, A. Labigne, 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
72. J.C. Venter, et al. The sequence of the human genome. Science 291:1304–1351, 2001
73. P. Weiner. Linear pattern matching algorithm. In: FOCS73, pp. 1–11, 1973
74. I.H. Witten, A. Moffat, T.C. Bell. Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, San Francisco, Calif., USA, 2nd edn., 1999