Universität Bielefeld

Technische Fakultät
Abteilung Informationstechnik
Forschungsberichte

Suffix Tree Construction and Storage with Limited Main Memory

Klaus-Bernd Schürmann Jens Stoye

Report 2003-06

Impressum: Herausgeber:
Robert Giegerich, Ralf Hofestädt, Franz Kummert, Peter Ladkin,
Helge Ritter, Gerhard Sagerer, Jens Stoye, Ipke Wachsmuth

Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31,
33501 Bielefeld, Germany

ISSN 0946-7831

Suffix Tree Construction and Storage with Limited Main Memory

Klaus-Bernd Schürmann and Jens Stoye

AG Genominformatik, Technische Fakultät, Universität Bielefeld, Germany
klaus@techfak.uni-bielefeld.de

Abstract. Suffix trees have been established as one of the most versatile index structures for unstructured string data like genomic sequences and other strings. In this work, our goal is the development of algorithms for the efficient construction of suffix trees for very large strings and their convenient storage regarding fast access when main memory is limited. We present a construction algorithm which, to the best of our knowledge, is currently the fastest practical construction method for large suffix trees.
Further we propose a clustered storage scheme for the suffix tree that takes into account the locality behavior of typical query types, which leads to a significant speed up particularly for the exact string matching problem. For very large strings the query time is faster than that of other recent index structures like the enhanced suffix array.

1 Introduction

Our application area is the analysis of genomic data like DNA or protein sequences. Since the 1990's when high-throughput technology was introduced to sequencing of genomic DNA, the amount of such data grows exponentially, similar to Moore's law which states that chip density doubles every eighteen months, just with a shorter doubling interval: Since 1995 the amount of base pairs contained in *GenBank* [3] doubles about every twelve to sixteen months. This rapid rise demands the development of more efficient algorithms and data structures for genomic sequence analysis. In addition, suffix trees are widely used in other large scale application areas like data mining and information retrieval.

In many applications of string processing it is essential to have an index on top of the raw string. Commonly used string indices can be divided in two major fields – word-based and full text indices. Word-based indices like inverted files [4] are not suitable for our application area, since a DNA sequence does not consist of natural individual words. Hence, the suffix tree, being the most powerful full text index available, seems to be the best suited index structure. Construction and storage for the suffix tree of a text t takes $O(|t|)$ time and space. It allows the efficient access to all substrings of t and therefore is the basis for many applications. A typical one is the location of a given substring p in t, which can be done in $O(|p|)$ time, independent of t, assuming an alphabet of constant size. The suffix array [11], a related data structure, is also not word based, but requires $O(|p| \log n)$ steps for the same operation.

Even though the linear time suffix tree construction algorithms by Weiner [18], McCreight [14], Ukkonen [17], and Farach [5] show optimal asymptotic behavior, they do not explicitly consider the memory hierarchy of multi-level cache, main memory, and (cached) hard disk, which leads to unfavorable effects on current computer architectures. If the suffix tree grows over main memory size, its construction is not feasible any more. With respect to the high memory requirements of the linear time algorithms it is hardly possible in practice to construct suffix trees for very large strings like the human genome with about 3.2×10^9 bp (base pairs). Therefore practical construction algorithms must take into account the locality of data access.

Nevertheless, the construction of a large suffix tree can take several hours. Thus it is not reasonable to build an index for a few search operations only, and only if the considered string changes very rarely, the construction cost amortizes over several searches. Thus, our research is directed not only at the development of efficient construction algorithms, but also at a persistent storage mechanism that allows fast query processing.

In Section 2 we give some definitions concerning suffix trees. Section 3 is devoted to existing and new algorithms. In Section 3.1 we review a suffix tree construction algorithm by Hunt *et al.* [9] and add a detailed average case analysis. In Section 3.2 we describe our improvements of this algorithm. In Section 4 we describe a suitable persistent storage scheme for the fast access of suffix tree data. Experimental results are presented in Section 5, and Section 6 concludes.

2 Suffix Trees – Definition and Terminology

Let Σ be a finite alphabet of fixed size and $t = t_1 t_2 \dots t_n \in \Sigma^n$ a text over Σ. Let $t^+ = t\$$ denote the extension of string t by a character \$ ($\$ \notin \Sigma$). For $1 \le i \le n$, let $s_i(t^+) = t_i \dots t_n\$$ indicate the i^{th} (non-empty) suffix of t^+.

The suffix tree $\mathcal{T}(t)$ of the text t is a rooted tree. $V_{\mathcal{T}(t)}$ denotes the vertices and $E_{\mathcal{T}(t)}$ the edges of $\mathcal{T}(t)$. Each internal node of the suffix tree is branching, and each edge is labeled by a non-empty substring of t^+. For each internal node $u \in V_{\mathcal{T}(t)}$, outgoing edge labels have different initial character. Each of the $n+1$ leaves is labeled with a unique index i, $1 \le i \le n+1$, such that the concatenation of the edge labels on the path from the root to that leaf equals $s_i(t^+)$.

The path label $plabel(u)$ of a node $u \in V_{\mathcal{T}(t)}$ is defined as the concatenation of the edge labels on the path from the root of $\mathcal{T}(t)$ to u. The depth $depth(u)$ of a node $u \in V_{\mathcal{T}(t)}$ is defined as the length of its path label, $|plabel(u)|$.

For $w \in \Sigma^*$, the *w-branch* of $\mathcal{T}(t)$, $\mathcal{T}_w(t)$ is defined as the induced subgraph of $\mathcal{T}(t)$ that contains the set of vertices $V_{\mathcal{T}_w(t)} = \{u \in V_{\mathcal{T}(t)} \,|\, plabel(u) = wx^+ \; for\ some\ x \in \Sigma^*\}$. This is the subtree of $\mathcal{T}(t)$ that represents all substrings of t with prefix w. $\mathcal{S}_w(t) = \{i \,|\, wx^+ = s_i(t^+) \; for\ some\ 1 \le i \le n,\, x \in \Sigma^*\}$ denotes the corresponding set of suffix numbers. The partitioning of all suffixes with respect to a given prefix length d is denoted by $\mathcal{P}_d(t) = \{\mathcal{S}_w(t) \,|\, w \in \Sigma^d\}$, and the set of corresponding suffix tree branches is indicated by $\widehat{\mathcal{P}_d}(t) = \{\mathcal{T}_w(t) \,|\, w \in \Sigma^d\}$.

The *d-trunk* of $\mathcal{T}(t)$, $\mathcal{R}_d(t)$, is defined as the subgraph of $\mathcal{T}(t)$ induced by the set of vertices $V_{\mathcal{R}_d(t)} = \{u \in V_{\mathcal{T}(t)} \mid depth(u) \leq d, \}$. Joining $\mathcal{R}_d(t)$ and all subtrees in $\widehat{\mathcal{P}_d}(t)$ gives the complete suffix tree $\mathcal{T}(t)$.

3 Construction of Suffix Trees

Construction algorithms of suffix trees are well explored in theory. Weiner [18], McCreight [14], Ukkonen [17], and Farach [5] introduced linear time algorithms. Unfortunately those algorithms do not explicitly consider the locality of memory reference which is very important on current computer architectures. Hence, in practice suffix trees are limited to main memory size. Combined with their large memory requirements of about 10 bytes per input character on average, it is currently not possible to build suffix trees for very large strings. On a common desktop computer with 512 MBytes of main memory size, for example, it is not feasible to construct, the suffix tree of a text longer than 55×10^6 characters.

An important issue in linear time suffix tree construction is the use of suffix links. A suffix link connects a node with path label ax ($a \in \Sigma$, $x \in \Sigma^*$) with the node with path label x. All of the mentioned algorithms take advantage of this information during the construction. Unfortunately their use involves random memory access and thus leads to a high ratio of cache misses for the linear time construction algorithms. Further on, suffix links are a waste of space, since many applications do not require their use and therefore they are just used during construction.

There are fast practical approaches to construct suffix trees without the use of suffix links, such as the Hashed-Position-Tree [8] and the write-only top-down (*wotd*) algorithm [13]. A strong benefit of the wotd-algorithm is its ideal locality behavior concerning tree access. It reduces the problem of suffix tree construction to suffix sorting, which is well understood. Although its worst-case time complexity is $O(n^2)$, in the expected case it takes $O(n \log n)$ time.

The Hashed-Position-Tree has the same time bounds as the wotd-algorithm. As its name promises, it is a hybrid, which combines the advantages of hashing and the suffix tree index. In addition to that, it uses explicit secondary memory management.

The construction algorithm we present in this paper is an improvement of results by Hunt *et al.* [9] who introduced an algorithm that constructs different parts of the suffix tree independently. In the following we will describe properties of their approach and give an average case time analysis of their algorithm. Then we will discuss further ideas how to modify the algorithm in order to improve the expected construction time.

3.1 A review of the Partitioning Algorithm

The following partitioning algorithm was introduced by Hunt *et al.* [9]. It sacrifices optimal $O(n)$ time complexity for locality of memory reference by omitting the use of suffix links and performing multiple passes over the text t.

It is based on the property that leaves which correspond to suffixes with common prefix w are located in the same subtree $\mathcal{T}_w(t)$. Suffix numbers are mapped to sets $\mathcal{S}_w(t)$ of the partition $\mathcal{P}_d(t)$ concerning the first $d = |w|$ characters of the corresponding suffix. The prefix length d is fixed and should be chosen long enough to ensure that the size of each w-branch $\mathcal{T}_w(t) \in \mathcal{P}_d(t)$ does not exceed the main memory size. Otherwise the algorithm fails. The algorithm proceeds as illustrated in Figure 1. For each prefix $w \in \Sigma^d$, all d-length substrings of t are scanned (lines 1+2). If an occurrence of w at position i is found, then the suffix $s_i(t^+)$ is inserted into the incrementally growing tree $\mathcal{T}(t)$ (lines 3-5). After t is traversed, the subtree $\mathcal{T}_w(t)$ is completely built, and the storage management of the used platform takes care of its persistent storage, indicated by a database *checkpoint*. (In their implementation, Hunt *et al.* [9] use the persistent Java based platform PJama [15].)

Analysis. Since subtrees $\mathcal{T}_w(t)$ and $\mathcal{T}_{w'}(t)$ $(w, w' \in \Sigma^d, w \neq w')$ are disjoint, they can be built and stored independently. In contrast to the linear time algorithms, where the size of the suffix tree is limited to main memory size, this approach can also be applied for very large strings, as long as the partitions can be expected to be small. Hunt *et al.* [9] showed that even for moderate size texts their algorithm has a competitive running time, which is due to its good locality behavior.

However, the average case analysis of the running time of their algorithm reveals a disadvantage. A simple consequence of a result by Apostolico and Szpankowski [2] and Szpankowski [16] is the expected insertion depth for a suffix of $O(\log n)$. Although this leads to an expected time of $O(n \log n)$ to perform the n insertions, the repeated scan of all suffixes is responsible for an overall $\Theta(n^2)$ running time. Since the size of a w-branch $\mathcal{T}_w(t)$ must not exceed the main memory size, a lower bound on the number of sets of the partitioning, $|\mathcal{P}_d(t)|$, is σ_n/M, where σ_n is the size of a minimal main memory instantiation of a suffix tree of a text of length n, and M is the main memory size. Since $\sigma_n \in \Theta(n)$ and the memory has fixed size M, the number of partitions $|\mathcal{P}_d(t)|$ grows linearly with the string length n. For each of the $|\mathcal{P}_d(t)| \in \Theta(n)$ partitions $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$, the algorithm scans the whole sequence of length n. Hence the overall time needed to scan the partitions is $\Theta(n^2)$. Since the n insertions can be performed in $O(n \log n)$ time, the complete algorithm takes $\Theta(n^2)$ time, regardless of the input string t.

3.2 The Clustered Construction Algorithm

In this section we present an algorithm which improves the partitioning algorithm. Our algorithm shows expected time $O(n \log n)$, for any d. The worst-case

Algorithm - BuildPartitionTree(t,d)

1: **for all** partitions $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ **do**
2: **for all** $i = 1 \ldots n+1$ **do**
3: **if** $i \in \mathcal{S}_w(t)$ **then**
4: insertSuffix($s_i(t^+)$)
5: **end if**
6: **end for**
7: checkpoint
8: **end for**

Fig. 1. Partitioning algorithm by Hunt *et al.* [9].

Algorithm - BuildClusteredTree(t,d)

1: $\mathcal{P}_d(t)$ = computePartitions(d)
2: **for all** partitions $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ **do**
3: $\mathcal{T}_w(t)$ = buildSubtree($\mathcal{S}_w(t)$)
4: writeClusterToDisk($\mathcal{T}_w(t)$)
5: **end for**
6: $\mathcal{R}_d(t)$ = buildSuffixTreeTrunk($\mathcal{P}_d(t)$)
7: writeTrunkToDisk($\mathcal{R}_d(t)$)

Fig. 2. Improved clustered algorithm.

time, however, is still $O(n^2)$. We call it the *clustered algorithm*, since we build the whole tree by constructing independent subtrees that are stored in clusters.

The algorithm is shown in Figure 2. As in the original algorithm, partitioning starts with a fixed prefix length $d = |w|$. This d is called the *clustering depth*. But here, before actually creating suffix tree branches, in a preprocessing step each suffix is mapped to its respective partition (line 1). This is performed by a serial scan of t. Now the independent subtrees can be built. For each set $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ our algorithm performs an independent construction of the respective suffix tree branch $\mathcal{T}_w(t)$. This is done via insertion of the respective suffixes $s_i(t^+)$, $i \in \mathcal{S}_w(t)$, into the initially empty branch. The main improvement compared to the basic algorithm is that the insertions do not start at the root of the suffix tree but directly in a node that is found at depth $d = |w|$ of the tree. After construction, the memory representation of the w-branch $\mathcal{T}_w(t)$ is converted to its secondary memory representation and written to disk. Finally, the trunk $\mathcal{R}_d(t)$ is built by inserting the corresponding prefixes w for each non-empty partition $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ and stored separately.

However, it is possible to use a different construction method for the trunk than for the w-branches, for example, the wotd-algorithm [13]. Manzini and Ferragina [12] proposed a similar approach for suffix arrays, which they call *deep-shallow* suffix sorting. They mix an algorithm for sorting suffixes with small longest common prefix (*shallow sorter*) with an algorithm for sorting suffixes with large longest common prefix (*deep sorter*). In our algorithm the suffixes with small longest common prefix are considered in the trunk construction, and the suffixes with large longest common prefix are considered in the construction of the w-branches.

Analysis. We show the average case analysis of our algorithm. The computation of all partitions $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ is done by once shifting a window of width d over the string t. Using a simple hash function for the prefix w that can be updated in constant time when the window is shifted by one position, the whole partitioning can easily be computed in $\Theta(n)$ time. If we suppose a uniform distribution of the

prefixes, the n suffixes are equally distributed over all partitions, and for each partition $\mathcal{S}_w(t) \in \mathcal{P}_d(t)$ we construct the appropriate w-branch $\mathcal{T}_w(t)$ via insertion of the $|\mathcal{S}_w(t)|$ suffixes. Since the expected insertion time for one suffix is $O(\log n)$ and there are n suffixes to be inserted over all partitions, the computation of the n insertions can be performed in $O(n \log n)$ expected time. The same time bound of $O(n \log n)$ holds for the construction of the trunk $\mathcal{R}_d(t)$, since there are $|\mathcal{P}_d(t)| \in O(n)$ prefixes to be inserted into it. By adding the expected times of all three parts of our algorithm, we get an overall expected running time of $\Theta(n) + O(n \log n) + O(n \log n) = O(n \log n)$, for arbitrary d.

There are other algorithms like the wotd-algorithm with the same time bound. However, the main benefit of our algorithm is the practical efficiency gained by the direct insertion into the w-branches. Since the search of the insertion position is usually starting at the root and proceeding through the dense trunk we obtain significant time savings depending on $d = |w|$ for each insertion operation.

4 Clustered Storage and Substring Search

Despite fast practical construction methods, it is not reasonable to construct suffix trees for a few search operations only. To avoid the repeated construction, the development of convenient storage schemes for suffix trees is another challenge. The requirements are fast access and space usage as small as possible. We propose a storage scheme that stores the independent suffix tree parts constructed by our algorithm in clusters, such that the w-branches are stored consecutively with respect to their construction order. The trunk is stored independently. This storage scheme is due to the behavior of applications which commonly traverse the suffix tree starting at the root. The standard application we consider is to decide if a given pattern p is a substring of t. Each search operation touches only the trunk and exactly one w-branch. During multiple search operations we keep the repeatedly touched trunk in memory such that just one w-branch has to be loaded from disk. Each w-branch is expected to fit in one disk block, and thus we just need one disk access per search operation. To save space for the storage of the suffix tree, we adopted a suffix tree format introduced by Giegerich *et al.* [6] which requires 8 Bytes per internal node and 4 Bytes per leaf. Just the interface between the trunk and the w-branches adds a few more bytes.

5 Experimental Results

In this section we investigate the practical behavior of our algorithms. We first analyse the time needed for different suffix tree construction methods for randomly generated strings regarding different sequence parameters like alphabet size and string length. Moreover, we use these underlying investigations to determine the optimal clustering depth $d = |w|$, the only parameter for the clustered

construction algorithm. For the experiments on real application data we collected a set of DNA sequences and other real world strings and compared the suffix trees with an additional data structure.

Concerning the application of suffix trees, we investigate the times needed for multiple substring searches for different storage schemes.

The experiments were performed on different computers with x86 architecture, running the Linux operating system. Programs were written in C and compiled with the *gcc*-compiler with optimization option '-O3'. In all experiments the total construction times in seconds are presented for each program and data set. The *cpu-time* is not considered since it just takes less than five percent of the overall time and therefore is not very informative.

All of our own programs are available from the authors upon request.

5.1 Suffix Tree Construction for Random Texts

The suffix tree construction for the random strings were performed on a computer with *Intel Pentium™2 (Klamath) 266 MHz CPU* and 128 MB of main memory.

Clustered construction with different clustering depths. In our first experiment, we investigate the clustered construction algorithm with different clustering depth $2 \leq d \leq 10$ for alphabet size 4, the DNA alphabet size, and clustering depth $1 \leq d \leq 3$ for alphabet size 90, the usual alphabet size of natural language text. The length of the randomly generated strings varies between 1 Million and 30 Million characters.

Figure 3 shows a diagram for the construction times with alphabet size 4. The upper diagram shows the construction times by the clustered algorithm where the complete construction can be done in main memory, and the lower diagram shows the construction times for large suffix trees leading to main memory overflow. For the shortest string length examined, 1 million characters, the construction for d between 2 and 8 takes between 4 ($d = 6$) and 12 ($d = 2$) seconds. Interestingly with clustering depth 10 the construction takes 25 seconds and diverges from the other depths. The optimal setting for string lengths up to 23 million characters is clustering depth 6, 7 or 8. The running times for clustering depth 7 are not shown in Figure 3, but its maximum deviation is about 1% from the times with clustering depths 6 and 8. The construction times for all settings rise slightly above linear with respect to the string lengths. Thereby, the increase for larger clustering depths is less than the increase for shorter clustering depths. This almost linear time behavior changes when the string length grows over a value of 22 million characters. Beyond this value the running time escalates and seems to increase exponentially. For clustering depth $d = 2$ the construction times clearly differ from the other clustering depths. In the beginning the construction time rises more significantly than for larger depths, but there is no exponential growth beyond a certain sequence length, such that the clustered construction for a string of 30 million characters can be performed in less than 40 minutes.

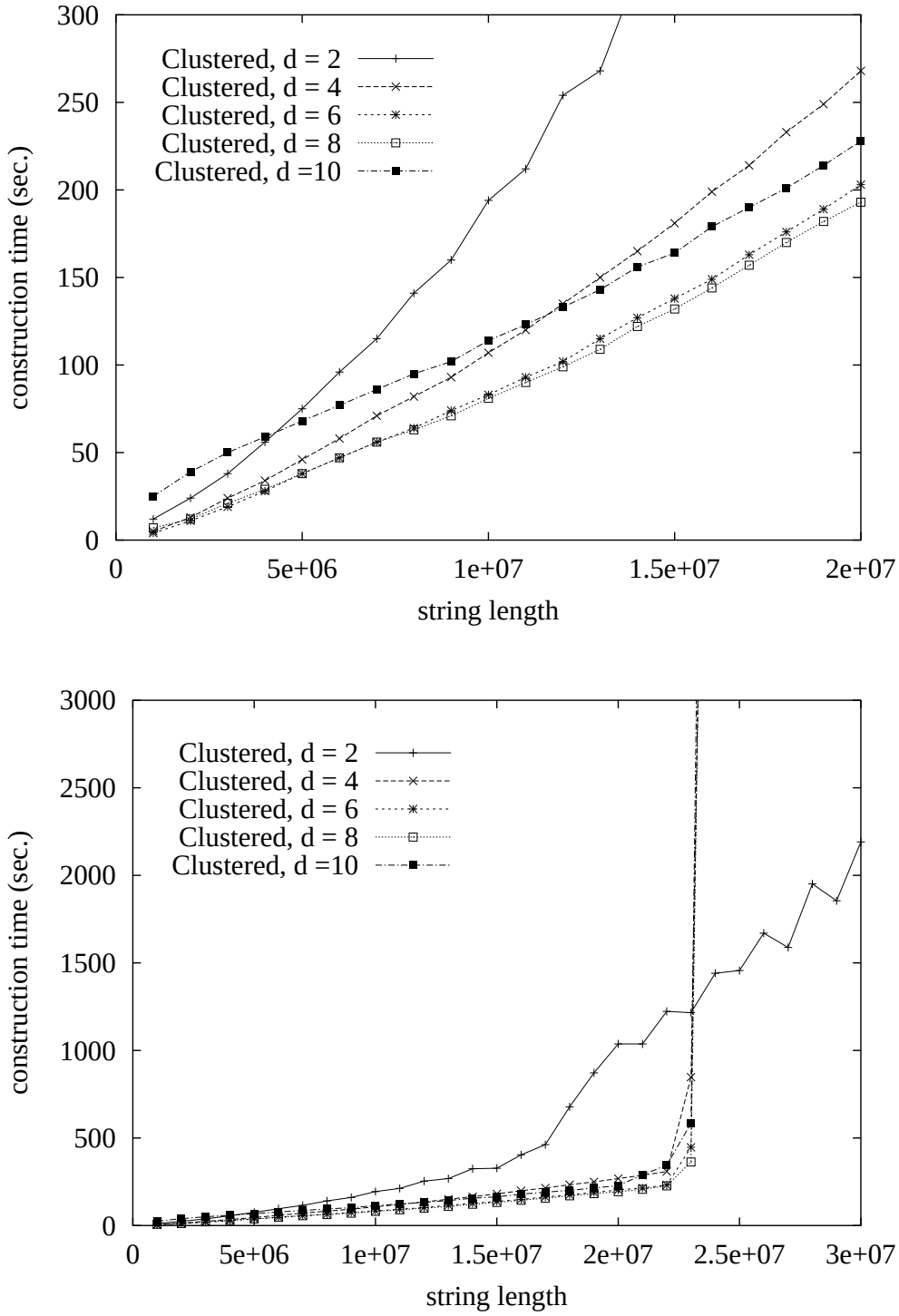

Fig. 3. Construction times of the clustered algorithm for alphabet size 4 and different clustering depths with linearly growing string length.

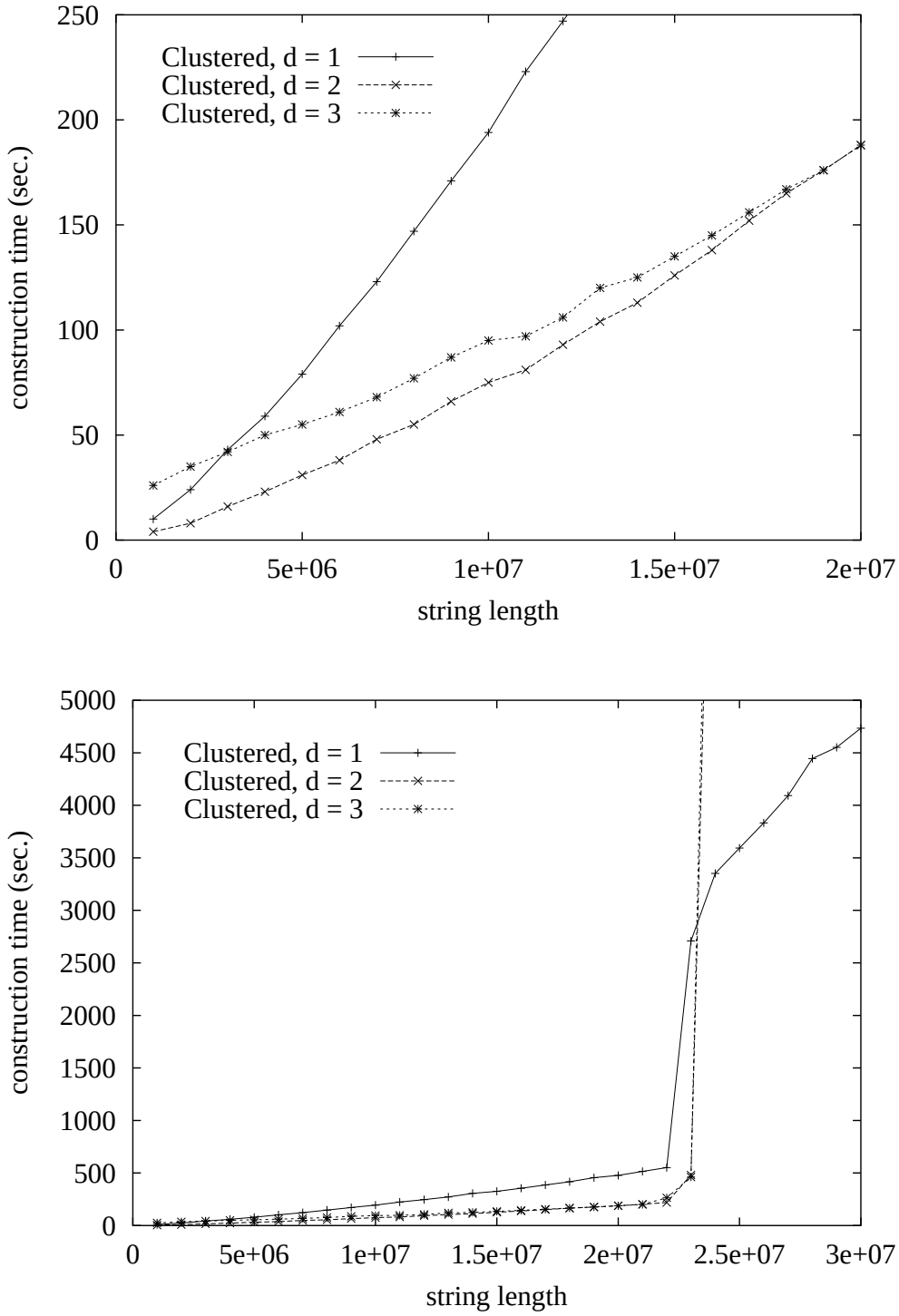

Fig. 4. Construction times of the clustered algorithm for alphabet size 90 and different clustering depths with linearly growing string length.

Figure 4 illustrates the construction time of the clustered algorithm for alphabet size 90. The upper diagram again shows the times for shorter input sequences, while the lower diagram shows the running times at the borders of main memory. For a string with 1 million characters the construction with clustering depth 3 takes 26 seconds. Hence, it is significantly slower than the construction with clustering depth 1 (10 sec.) and 2 (4 sec.). For the shorter clustering depth 1 the increase of the construction time is initially steeper than for the clustering depths $d = 2$ and $d = 3$, but similar as in the case of $d = 2$ at alphabet size 4, $d = 1$ is the only choice where still suffix trees for string lengths of 30 million characters could be constructed within reasonable time.

The escalating construction times for larger clustering depth (≥ 4 for alphabet size 4; ≥ 2 for alphabet size 90) beyond a sequence length of about 22 million characters is due to the increasing memory requirements exceeding the main memory size. The most space consuming structure is an array which stores the mapping of suffixes to clusters. In practice such construction times are not tolerable.

The slower construction time for large clustering depths and short sequences is due to the effort for the bookkeeping of clustering information such that, for example, with alphabet size 4 and clustering depth 10 there are $4^{10} = 1,048,576$ w-branches to be processed. In addition, the clustering depth has influence on the size of the d-trunk and the number and size of w-branches. A smaller clustering depth results in a smaller d-trunk and smaller number of larger w-branches. Whereas a larger clustering depth leads to a larger d-trunk and higher number of small w-branches. To determine the optimal choice of d one has to find the balance between the size of the d-trunk, the size of the w-branches and the effort for the bookkeeping of clustering information to get an optimal running time.

Clustered construction for different alphabet sizes. The alphabet size does not directly influence the structure of the suffix tree. But for the clustered construction algorithm the clustering depth combined with the alphabet size determines the numbers of w-branches. In the previous experiment we investigated different values of the clustering depth. Here we investigate the behavior of the clustered algorithm concerning the alphabet size. The strings have length 5–20 million characters. The alphabet sizes are 4, 8, 16 and 64. The clustering depth was chosen such that there are 4096 w-branches to be constructed, independent of the alphabet size. Given an alphabet of size 4, for example, we have chosen a clustering depth of $d = 6$ such that $4^6 = 4096$.

Figure 5 visualizes the results of this experiment. Given an input string of 5 million characters, the clustered construction with alphabet of size 4 takes 37 seconds. This is significantly slower than for alphabet size 8, where the algorithm takes 32 seconds or for alphabet sizes 16 and 64 where the construction takes 29 seconds. Increasing the string length has a negative effect on the construction time for alphabet size 64 compared to smaller alphabet size. For a string of 20 million characters and alphabet size 64 the construction time takes

215 seconds, though the times for alphabet sizes 4, 8 and 16 just take 209, 191 and 186 seconds, respectively. The fastest construction is performed for strings with alphabet size 16 independent of the string length.

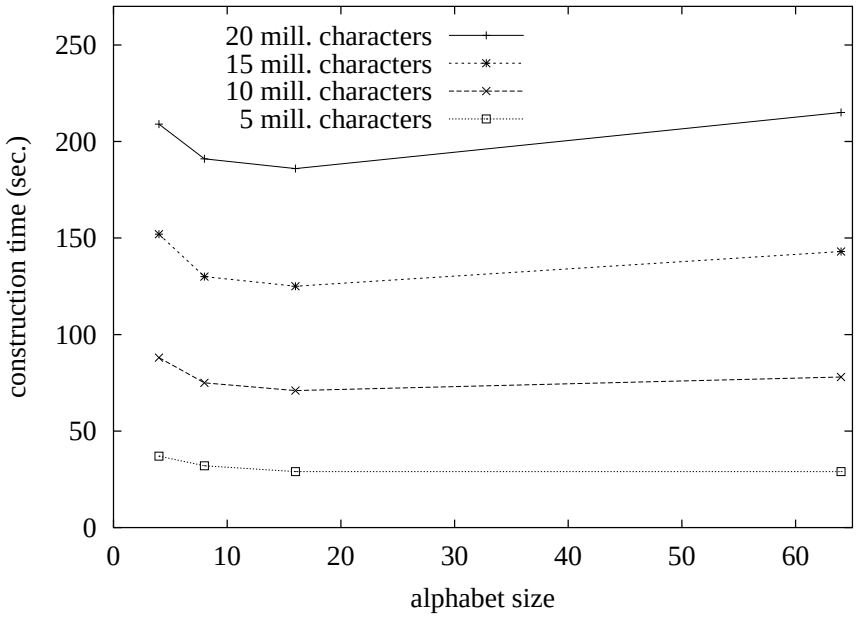

Fig. 5. Construction times of the clustered algorithm for alphabet sizes between 4 and 64 and string lengths between 5 and 20 million characters.

The faster clustered construction for strings with alphabet size 16 compared to alphabet size 4 resp. 8 is due to the structure of strings with smaller alphabet. With smaller alphabet there is a higher probability for longer repeats in the string, such that there are comparatively many nodes in the suffix tree, since the average number of children per node is small. Compared to strings with alphabet size 64 the suffix tree construction for strings with alphabet size 16 is faster for increasing string length, due to similar reasons. In this case also the expected number of children per node is bigger for the larger alphabet, since for each insertion of a suffix into the tree the list of children have to be scanned for each node on the path from the root towards the insertion position to find the right insertion point. So there are two aspects of a larger alphabet size and thus a larger number of children for the internal nodes. On the one hand this reduces the expected number of internal nodes, but on the other hand the expected number of nodes to be touched during insertion is higher.

Comparison of different suffix tree construction algorithms. In our next experiment we compare different suffix tree construction algorithms for random

strings of increasing length. The investigated suffix tree construction methods are Ukkonen's algorithm [17] a representative of the linear time construction methods, in an implementation by Stefan Kurtz supplemented by our storage management, the wotd-algorithm [13], and the clustered construction algorithm, the latter two in our own implementation. The partitioning algorithm by Hunt *et al.* [9] could not be considered since their implementation in Java using a high-level interface for persistent memory management is not comparable to the other C implementations. Figures 6 and 7 illustrate the results of the investigation for random strings with alphabet size 4 resp. 90. The clustering depth is chosen as proposed optimal by the previous investigation for the clustered algorithm. Hence, for alphabet size 4 we chose the clustering depth 2 for very large strings or 8 for medium sized strings, and for alphabet size 90 the clustering depth 1 respectively 3.

With alphabet size 4, Ukkonen's algorithm takes 4 seconds for a string of 1 million characters. Thus, it is about twice as fast as the clustered algorithm taking 12 resp. 7 seconds and the wotd-algorithm needing 8 seconds. But with increasing length of the input string the running time of Ukkonen's algorithm increases tremendously. For a string length of 15 million characters it needs more than 15 hours.

The clustered algorithm with clustering depth 2 is slower than the wotd-algorithm up to a string length of 10 million characters. For string lengths between 11 and 17 million characters it is faster than the wotd-algorithm and above a string length of 17 million characters it performs like the wotd-algorithm.

Concerning clustering depth 8 the clustered algorithm is significantly faster than the wotd-algorithm up to a string length of 23 million characters. For string lengths between 13 and 22 million characters it is even 3 times faster. Only for input strings of size more than 24 million characters the running time for the clustered construction increases tremendously.

For the larger alphabet size 90 the wotd-algorithm performs up to 2 times faster than the clustered algorithm with clustering depth 3 for strings shorter than 12 million characters. But for string lengths between 13 and 23 million characters the clustered construction is again significantly faster than the wotd-algorithm until the construction time escalates by building suffix trees for strings larger than 23 million characters. For smaller clustering depth 1 and string length between 17 and 22 million characters the construction times by the clustered algorithm are similar compared to those by the wotd-algorithm. But again, for larger strings the construction time of the clustered algorithm escalates.

The time curves for the clustered algorithm have been analysed before. To explain the different running time behavior it is necessary to go into the structure of suffix tree construction algorithms.

The suffix tree representation for Ukkonen's algorithm takes 20 bytes per node. Hence, the suffix tree grows over main memory size quickly. Combined with the non-local access of suffix tree nodes due to the usage of suffix links, the

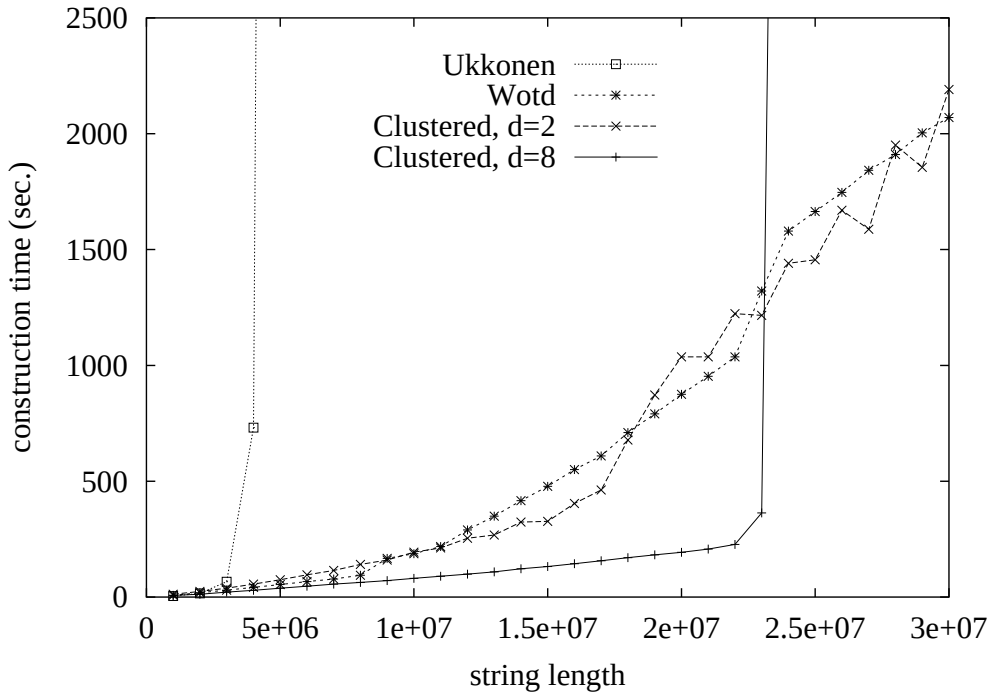

Fig. 6. Suffix tree construction times of different algorithms for alphabet size 4 and increasing string length.

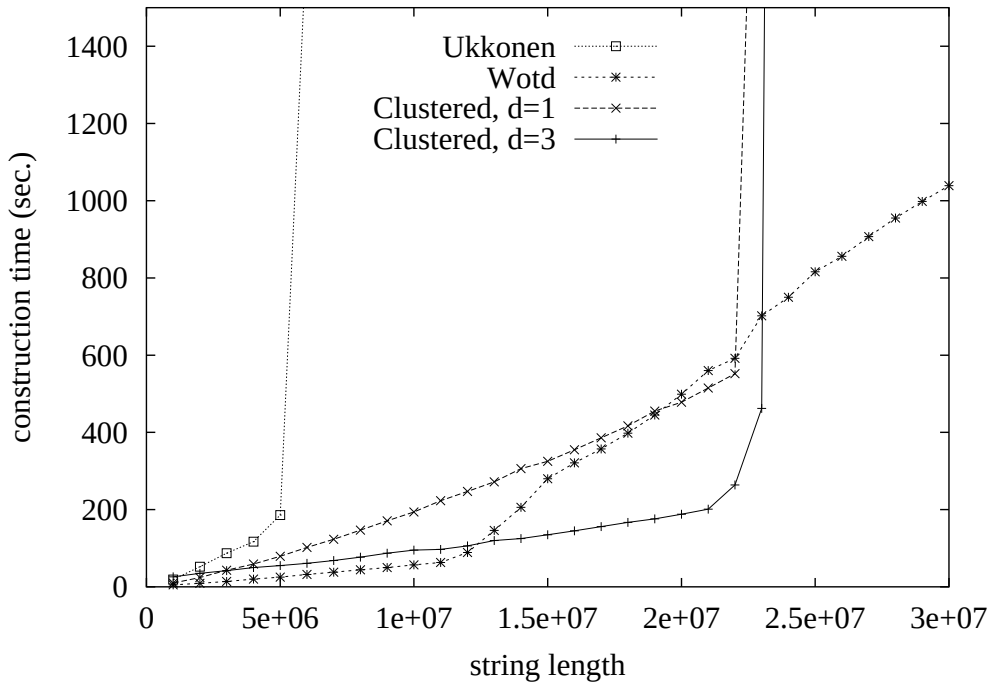

Fig. 7. Suffix tree construction times of different algorithms for alphabet size 90 and increasing string length.

running time seems to increase exponentially, since repeated disk access slows down the processing.

The clustered algorithm does not need to hold the complete suffix tree in main memory during construction. On the other hand the wotd-algorithm constructs the suffix tree en bloc. But it achieves its fast construction by a good locality of memory access and by using a suffix tree representation which needs small space. For alphabet size 4 and clustering depth 8 the clustered algorithm is significantly faster than the wotd-algorithm as long as the largest w-branch and the bookkeeping data structures completely fit into main memory. This behavior is due to the independent construction and storage of parts of the whole suffix tree. The construction of w-branches is done via direct insertion of suffixes into the w-branches. Since the clustered algorithm just performs one insertion per w-branch into the d-trunk this is a significant improvement. Additionally, the expected density of nodes in the d-trunk is bigger than in the w-branches such that during insertion more nodes of the d-trunk are touched than of the w-branches. In practice, this leads to additional speed up.

For alphabet size 90 and short input strings the wotd-algorithm is faster than the clustered construction since the performance of the clustered algorithm depends on the out-degree of internal nodes, but the wotd-algorithm does not. Additionally, the main memory representation of the suffix tree required by the clustered algorithm needs to be serialized before it can be stored on disk, while the representation for the wotd-algorithm is in storable form. But with increasing string length the clustered algorithm outperforms the wotd-algorithm due to the advantages of the clustered algorithm and since the suffix tree in the wotd-algorithm grows over main memory size.

For very large strings, if also the memory requirements for the clustered algorithm exceed main memory size, the wotd-algorithm outperforms the clustered algorithm due to its very good behavior of local memory access.

The clustered algorithm benefits from the independent construction and storage of suffix tree parts. These parts can be built completely in main memory. On the other hand, the wotd-algorithm benefits from its excellent locality of memory access, also allowing the construction of suffix trees for very large strings, even if the hard disk has to be accessed during construction

5.2 Suffix Tree Construction for Fibonacci Strings

The previous investigations have been performed for randomly generated strings. The resulting suffix trees for these strings are expected to be balanced. But strings in practical applications do not necessarily have random structure. Sometimes there are long repeats leading to an unbalanced tree structure. Hence, in this section we investigate Fibonacci strings: $f(i)$ is the i-th Fibonacci string where $f(i)$ is recursively defined as follows: $f(1) = a$, $f(2) = b$ and $f(i) =$

$f(i-2)f(i-1)$ for $i > 2$. These strings have alphabet size 2 and are well known for their long repeated substrings. See [10] for further information.

The system settings are the same as in the previous experiments. The experiments have been performed for Fibonacci strings of lengths between 50,000 and 1 million characters and the same construction algorithms as in the previous investigation. For the clustered algorithm we have chosen the clustering depths 10 and 16 since the alphabet is smaller than for previous investigations.

Figure 8 illustrates the construction times. For a Fibonacci string with 500,000 characters, Ukkonen's algorithm takes about 3 seconds while the suffix tree construction by the wotd-algorithm respectively the clustered algorithm takes about 2560 resp. 1550 seconds. For 1 million characters Ukkonen's algorithm takes about 8 seconds while the wotd-algorithm and the clustered algorithm did not terminate within the first 2 hours. By watching the time curves for the clustered algorithm, one recognizes that it is almost independent of the clustering depth.

Additional observations for a string length of 500,000 characters have shown some additional information. Although the expected size for a set of suffix numbers $\mathcal{S}_w(t)$ in the partitioning is 488.28 for clustering depth 10, the largest set has size 72,948. For clustering depth 16 the expected size is 7.63, but the largest set contains of 45,084 suffixes.

Overall the clustered algorithm is significantly faster than the wotd-algorithm.

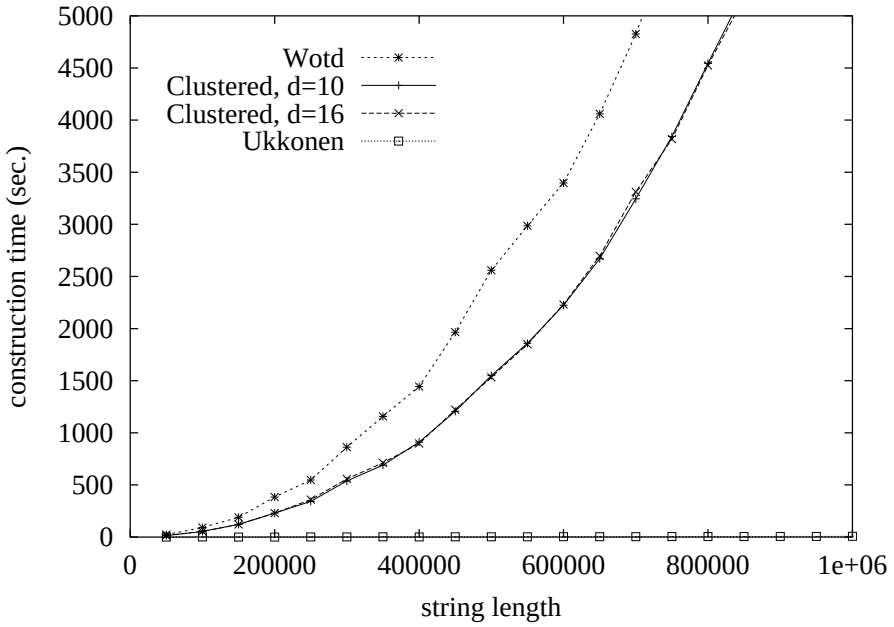

Fig. 8. Construction times of different algorithms for Fibonacci strings of increasing length.

It is easy to see that neither the clustered algorithm nor the wotd-algorithm are competitive for the construction of degenerated suffix trees, since their performance depends on the suffix tree structure. The parabola shape of the construction time curves indicates that the construction reaches its worst case time bound. Ukkonen's algorithm is independent of such degeneration and is thus the fastest construction method for unbalanced suffix trees. In this manner the clustered algorithm and the wotd-algorithm are not suitable for the construction of degenerated suffix trees. But Giegerich *et al.* [7] suggest a variant of the wotd-algorithm allowing the reuse of already computed longest common prefix information. It would be interesting to measure, if this technique leads to a significant speed-up of the wotd-algorithm since it uses some kind of suffix links and hence trades algorithmic improvement for locality of memory reference.

5.3 Suffix Tree Construction for Real Sequences

The experiments so far were useful to investigate the dependency of the suffix tree construction algorithms on particular parameters. In this section we investigate their behavior on strings as they occur in practice. This is done on the same computer system as before. First we analyse the suffix tree construction for different DNA sequences, then we investigate the construction for other kinds of strings like natural language text and source code.

Suffix tree construction for DNA sequences. We investigate different DNA sequences: The complete genome of the bacterium *Escherichia coli (E. coli)* consisting of about 4.6 Mbp (mega base pairs) as well as the third and fifth chromosome of the nematode *Caenorhabditis elegans (C. elegans)* with approximate length 12.8 respectively 20.5 Mbp.

The running times for the different suffix tree construction algorithms and different settings are shown in Table 1. Dashes indicate that on the *C. elegans* chromosomes, Ukkonen's algorithm did not terminate within 24 hours. For the clustered algorithm the clustering depths 6 and 8 were confirmed to be the optimal settings. They are given in Table 1. Additionally the construction times are illustrated in Figure 9, normalized to a string length of 1 million characters.

string	string length	alphabet size	construction time (sec.)			
			Clustered d=6	Clustered d=8	Wotd	Ukkonen
E. coli genome	4,638,690	4	34	38	48	10,237
C. elegans chr. 3	12,836,730	4	132	122	283	–
C. elegans chr. 5	20,551,922	4	282	265	1,125	–

Table 1. Suffix tree construction times for different DNA sequences by different algorithms.

The clustered algorithm clearly shows the fastest construction among the different methods. The normalized construction times for the clustered algorithm do not highly increase with rising string length. In contrast, the wotd-algorithm takes 48 seconds for the construction of the *E. coli* suffix tree. Hence, it is little slower than the clustered algorithm. But for larger input strings the construction times highly increase. Ukkonen's algorithm takes more than 2 hours for the suffix tree construction for *E. coli*. Hence it is clearly the slowest construction method.

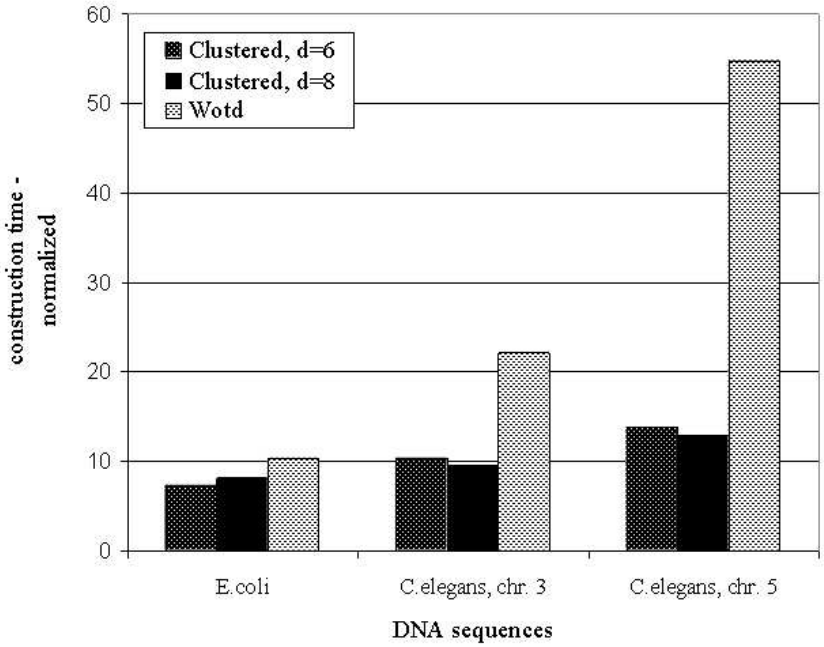

Fig. 9. Running time in seconds of different suffix tree construction algorithms for the DNA sequences normalized with respect to string length 1 million.

The results of the experiments for different DNA sequences mirror the results of the experiments for the randomly generated sequences. Hence, we assume that also the suffix trees for DNA sequences are balanced. Whenever the data necessary for the construction by the clustered algorithm completely fits into main memory and the string is too large for Ukkonen's algorithm, the clustered algorithm is the fastest construction method for suffix trees of DNA sequences.

Suffix tree construction for different strings. Next we investigate strings important for other application areas. The texts are four bibles of different language, the source code of the robocup team (robocup) of the Free University in Berlin, the source code of the gcc-compiler (gcc), a part of the linux source code (linux), a part of a protein database (protein DB), the ftp-index of the web

server of the Technical University in Berlin (ftp), consisting of file and directory paths, and the CIA World Fact Book (cia), containing facts of all countries of the world. All these strings have larger alphabets than DNA sequences. Thus the dedicated clustering depths are 2 and 3.

The running times for the different suffix tree construction algorithms are given in Table 2. It shows the name of the sequence, its length and alphabet size, and the construction times of the different construction algorithms. As above, there are missing values for Ukkonen's algorithm with the large data files.

For the given set of sequences, the clustered algorithm is the fastest among all suffix tree construction algorithms. Only the construction for the robocup source code needs unexpectedly long time. For the clustered method it takes 2620 respectively 2638 seconds and for the wotd-algorithm 5417 seconds. In contrast, the linux source code is nearly 3 times bigger than the robocup source code, but the construction by the clustered algorithm just takes 322 respectively 316 seconds, and by the wotd-algorithm it needs 1219 seconds.

string	string length	alphabet size	construction time (sec.)			
			Clustered d=2	Clustered d=3	Wotd	Ukkonen
bible (danish)	3,838,509	83	32	32	42	71
bible (english)	4,047,392	63	32	35	46	151
bible (german)	4,638,707	91	40	42	56	4,785
bible (french)	5,122,590	78	48	48	64	15,143
robocup	7,243,042	108	2,620	2,638	5,417	–
gcc	14,893,334	99	187	185	427	–
linux	20,775,894	105	322	316	1,219	–
protein DB	20,000,000	83	221	232	824	–
ftp	4,862,809	88	51	51	88	–
cia	2,473,400	94	16	15	26	18

Table 2. Running time in seconds by different suffix tree construction algorithms for different kind of strings.

The reason for the long robocup time is the structure of this file, consisting of several concatenated *C++* source code files. Besides the usual *.cpp* respectively *.h* files the code consists of *.ui* files for the description of the graphical user interface. These files are generated automatically by an IDE. Hence, these files consist of many long repeated patterns. By considering the result for the suffix tree construction of Fibonacci strings, it is obvious that this is the reason for the slower running times of the clustered and the wotd-algorithm.

The construction times for the other strings accord to the time seen for the suffix tree construction for random strings. All these texts result in balanced suffix trees. Thus, they can be built fast by the clustered respectively wotd-algorithm. The clustered algorithm is the fastest construction method for all

strings of the given data set. This observation holds as long as the main memory space requirements for our clustered construction, which are about six bytes per input character, do not exceed the main memory size. Exceeding this limit escalates the construction time. Until this boundary, our construction algorithm is the fastest practical suffix tree construction algorithm we are aware of.

5.4 Comparison with the Enhanced Suffix Array

In the previous experiments, we have compared different suffix tree construction algorithms and stated that, in practice, the clustered construction is the fastest construction method. But suffix trees are not the only data structure for full text indexing. Thus, in this section, we compare the different suffix tree construction algorithms with an additional index structure, the enhanced suffix array (*Esa*), which was introduced by Abouelhoda *et al.* [1]. The enhanced suffix array is based on the suffix array [11] and reaches the same functionality as the suffix tree by the addition of a few annotations. The programs concerning this data structure that were used in our tests were kindly provided by Stefan Kurtz.

For these experiments we collected DNA sequences of different length: the three DNA sequences we used before (E. coli and C. elegans chr. 3 and 5) and additionally sections of 40 respectively 80 Mbp of the human chromosome 10. In these experiments the index construction was performed on a computer with *AMD Athlon™ 1.3 GHz CPU* and 512 MBytes of main memory.

The results are illustrated in Table 3. For the clustered suffix tree construction the clustering depth has been set to 8 respectively 10. 10 is a suitable value for the following search operations. The enhanced suffix array uses an additional bucket table on top of the suffix array, which is needed to access the bucket of suffixes with equal prefix in constant time. This prefix length is also set to $d = 10$.

data set	construction time (sec.)				
	Clustered ($d = 8$)	Clustered ($d = 10$)	Wotd	Ukkonen	Esa ($d = 10$)
E. coli genome (4.6 Mbp)	9	17	16	16	8
C. elegans chr. 3 (12.8 Mbp)	40	59	54	169	40
C. elegans chr. 5 (20.6 Mbp)	103	117	147	54434	97
H. sapiens chr. 10 (40 Mbp)	177	209	466	–	151
H. sapiens chr. 10 (80 Mbp)	461	520	2171	–	358

Table 3. Construction time in seconds of the different algorithms for the DNA sequence files.

As before, among the suffix tree construction algorithms, our clustered construction is by far the fastest method for large sequences. For the 80 MB part

of human chromosome 10, for example, with $d = 10$ it takes 520 seconds which is a speed-up by a factor of four compared to the wotd-algorithm. Ukkonen's algorithm took even longer than 24 hours.

Comparing the construction times of the clustered algorithm with $d = 10$ which is used for the search experiments and the enhanced suffix array shows that the enhanced suffix array construction is about 20% faster for the mid-sized sequences of the *C. elegans* chromosomes and for the 40 MB part of the human chromosome 10, and it is about 45% faster for the 80 MB part of the human chromosome 10.

Furthermore, for the clustering depth 8 the time difference is smaller. For the short sequences *E. coli* and *C. elegans* chr. 3 the clustered algorithm is as fast as the construction of the enhanced suffix array. For *C. elegans* chr. 5, human chr. 10 (40 Mbp) and human chr. 10 (80 Mbp) the enhanced suffix array is about 6, 17 resp. 29 % faster.

5.5 Disk Space Requirements

The space requirements for the persistent representations of the clustered suffix tree, the enhanced suffix array, and the unclustered suffix tree are illustrated in Table 4. The persistent space requirements are given in bytes per input character. The enhanced suffix array consists of various tables, and we just collected the size of data which are essential for our dedicated application, the exact substring search. The space requirements are between 9.14 and 9.51 bytes per input character for the suffix trees. The clustered storage scheme adds a few bytes for the interface between the trunk and the w-branches. The space requirements for the enhanced suffix array are between 6.14 and 7.87, where the size of the previously mentioned bucket table depends on the respective prefix length d. Due to the choice of $d = 10$, the size of the bucket table is $|\Sigma|^d = 4^{10}$ bytes. However, to get the full functionality of the enhanced suffix array, tables have to be added and thus the space requirements would rise.

data set	space per character (bytes)		
	clustered ($d = 10$)	unclustered	esa ($d = 10$)
E. coli genome (4.6 MB)	9.51	9.14	7.87
C. elegans chr. 3 (12.8 MB)	9.39	9.32	6.69
C. elegans chr. 5 (20.6 MB)	9.34	9.32	6.50
H. sapiens chr. 10 (40 MB)	9.39	9.37	6.25
H. sapiens chr. 10 (80 MB)	9.37	9.36	6.14

Table 4. Space requirement for the different indices per input character for different s sequence files.

5.6 Substring Search

Since the index is built just once but repeatedly used, the amortization of construction cost mainly depends on the effectiveness of the applications the suffix tree serves. For our experiments, we have chosen a classical application of suffix trees, the determination if a pattern p is a substring of t. We investigated this application by measuring the time for 10^5 search operations for the unclustered suffix tree, the enhanced suffix array, and our clustered suffix tree. Patterns were randomly chosen substrings of t of varying length. These experiments were performed on a laptop computer with *Intel Pentium™4 Mobile 1.6 GHz CPU* and 256 MB of main memory.

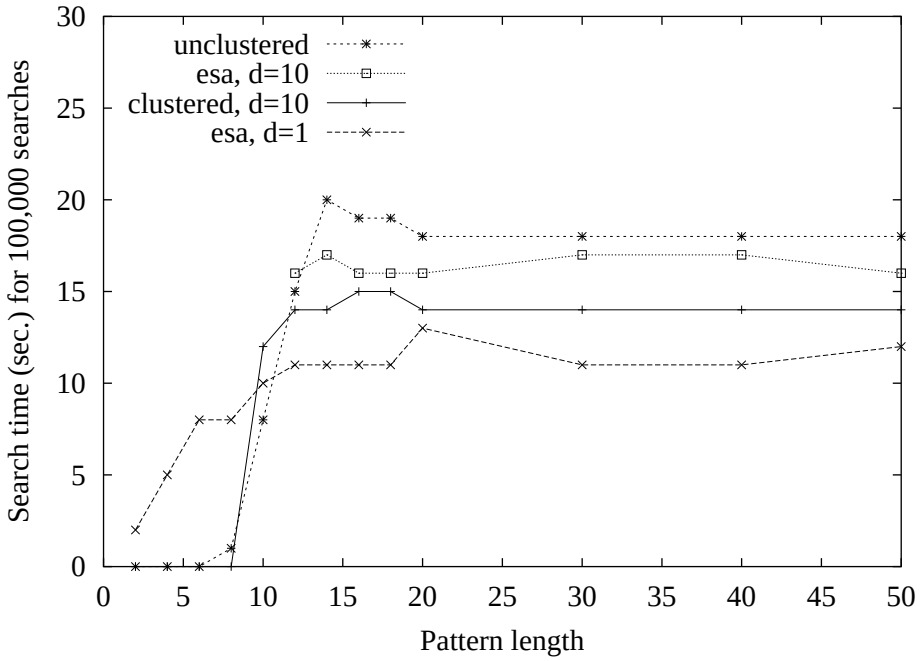

Fig. 10. Time for 10^5 search operations on the complete *E. coli* genome (4.6 MB).

Figure 10 illustrates the search times for different pattern lengths on the complete genome of *E. coli*, and Figure 11 shows the same information for the 80 MB part of the human chromosome 10. For the clustered suffix tree scheme the depth $d = 10$ of the trunk is such that the trunk of size about 1.1 MB for the 80 MB part of the human chromosome fits easily into the main memory of most currently used computers. Other values of d would result either in longer search times ($d < 10$) or clearly higher construction times ($d > 10$), therefore results for such values are not presented. The search times for the enhanced suffix array are shown with respect to the prefix lengths $d = 1$ and $d = 10$. For $d = 10$, only

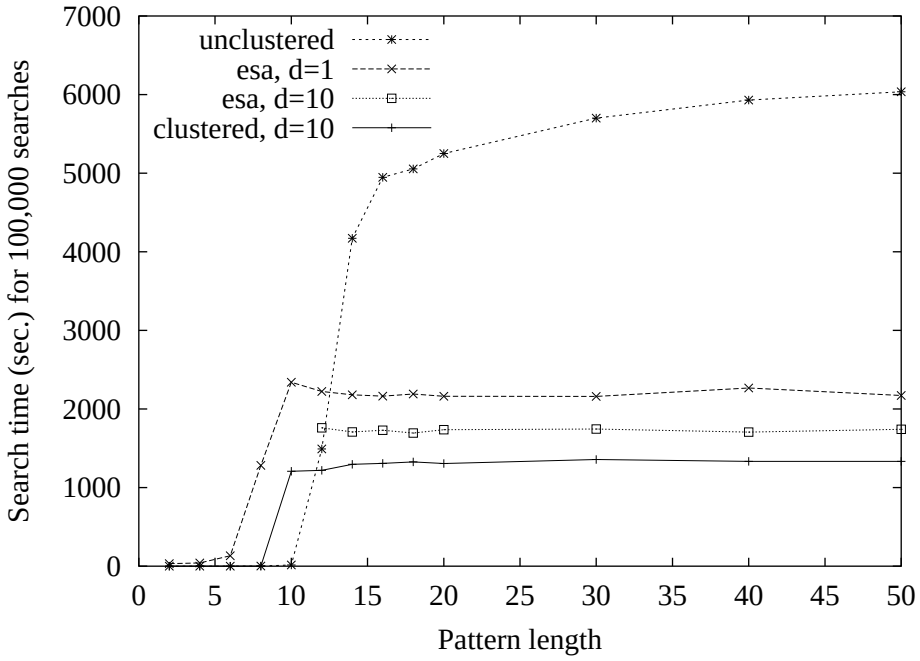

Fig. 11. Time for 10^5 search operations on the *H. sapiens* chromosome 10 (80 MB).

patterns longer than 10 could be tested, since the enhanced suffix array requires a pattern length of at least the prefix length.

For *E. coli* (Figure 10) the search with the different suffix trees is faster than with the enhanced suffix array, as long as the pattern is short enough to ensure that the searched part of the suffix tree completely fits into main memory. For the clustered suffix tree the search times jump up at a pattern length of 10, when the clusters below the trunk have to be accessed. The search times for the enhanced suffix array are initially slower than for the suffix trees. However, with increasing pattern length the increase is less drastical than for the suffix trees, such that beyond a pattern length of 10 searching the enhanced suffix array with prefix depth $d = 1$ is about 25% faster than searching the clustered suffix tree.

Figure 11 illustrates times for the repeated search in the large 80 MB data set. Here neither the enhanced suffix array nor the suffix tree representations completely fit into main memory. For patterns of length 20 the search times for the 10^5 patterns are 1306 seconds for the clustered suffix tree, 1737 seconds for the enhanced suffix array with prefix depth 10, and 5250 seconds for the unclustered suffix tree. Hence the clustered suffix tree representation leads to a speedup of about 33% compared to the enhanced suffix array and 400% compared to the unclustered representation for the 80 MB part of human chromosome 10.

In summary, one can say that as long as both indices, the clustered suffix tree and the enhanced suffix array, fit completely into main memory, the search times slightly favor the enhanced suffix array with prefix length $d = 1$. For medium

sized texts, where only the enhanced suffix array fits into main memory, the search times of the enhanced suffix array are faster than for the clustered suffix tree. As soon as none of the data structures fits in main memory, the clustered suffix tree shows the fastest search times. Comparing the absolute times, one can easily see that the gain of about 400 seconds for searching a large text 10^5 times more than compensates for the small loss in time (by less than 5 seconds) when searching a small text.

6 Conclusion

We have presented a suffix tree construction method with expected time complexity of $O(n \log n)$. Our algorithm is well suited for the construction of large suffix trees in bioinformatics and other applications, as long as the main memory size is six times as big as the sequence length. For other types of strings it is faster than other approaches including the wotd-algorithm or Ukkonen's algorithm. Hence, it is the currently fastest practical suffix tree construction algorithm and even competitive with construction algorithms for other practical index structures like the enhanced suffix array.

Concerning the exact string matching problem, the clustered suffix tree representation is the most efficient storage scheme for suffix trees when the searched part of the suffix tree exceeds the main memory size. If also the size of the enhanced suffix array grows over main memory size, the search operations are faster using the clustered suffix tree compared to the enhanced suffix array.

Besides the substring search there are many other applications on suffix trees. For these applications it has to be investigated if our clustered suffix tree representation also leads to an efficiency gain. Moreover, there are some applications for suffix trees for which the usage of suffix links is essential, like the matching statistics. Thus we have to find practical ways to enrich our suffix tree with suffix links.

Acknowledgment. We thank Stefan Kurtz for providing his library for the encapsulation of the *gcc-mmap* functions and his programs for the enhanced suffix array.

References

1. M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer Verlag, September 2002.
2. A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. *Journal of Algorithms*, 13(3):446–467, 1992.
3. D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. Genbank. *Nucleic Acids Research*, 31(1):23–27, 2003.
4. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM (JACM)*, 34(3):578–595, 1987.

5. M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 97)*, pages 137–143, October 1997.
6. R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE 1999)*, volume 1668 of *Lecture Notes in Computer Science*, pages 30–42. Springer Verlag, 1999.
7. R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, 2003.
8. K. Heumann and H. W. Mewes. The hashed position tree (HPT): A suffix tree variant for large data sets stored on slow mass storage devices. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 101–115, 2002.
9. E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, pages 139–148. Morgan Kaufmann, 2001.
10. C. S. Iliopoulos, D. Moore, and W. F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1-2):281–291, February 1997.
11. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
12. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 698–710. Springer Verlag, September 2002.
13. H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11(13):4629–4634, 1983.
14. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.
15. T. Printezis, M. Atkinson, L. Daynes, S. Spence, and P. Bailey. The design of a new persistent object store for PJama. In *Proceedings of the 2nd International Workshop on Persistence and Java*, pages 61–74, 1997. Published as SunLabs Technical Report TR-97-63.
16. W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEETIT: IEEE Transactions on Information Theory*, 39(5):1647–1659, 1993.
17. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.
18. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Bisher erschienene Reports an der Technischen Fakultät
Stand: 2003-08-19

94-01 Modular Properties of Composable Term Rewriting Systems
(Enno Ohlebusch)

94-02 Analysis and Applications of the Direct Cascade Architecture
(Enno Littmann, Helge Ritter)

94-03 From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction
(Robert Giegerich, Stefan Kurtz)

94-04 Die Verwendung unscharfer Maße zur Korrespondenzanalyse in Stereo Farbbildern
(André Wolfram, Alois Knoll)

94-05 Searching Correspondences in Colour Stereo Images – Recent Results Using the Fuzzy Integral
(André Wolfram, Alois Knoll)

94-06 A Basic Semantics for Computer Arithmetic
(Markus Freericks, A. Fauth, Alois Knoll)

94-07 Reverse Restructuring: Another Method of Solving Algebraic Equations
(Bernd Bütow, Stephan Thesing)

95-01 PaNaMa User Manual V1.3
(Bernd Bütow, Stephan Thesing)

95-02 Computer Based Training-Software: ein interaktiver Sequenzierkurs
(Frank Meier, Garrit Skrock, Robert Giegerich)

95-03 Fundamental Algorithms for a Declarative Pattern Matching System
(Stefan Kurtz)

95-04 On the Equivalence of E-Pattern Languages
(Enno Ohlebusch, Esko Ukkonen)

96-01 Static and Dynamic Filtering Methods for Approximate String Matching
(Robert Giegerich, Frank Hischke, Stefan Kurtz, Enno Ohlebusch)

96-02 Instructing Cooperating Assembly Robots through Situated Dialogues in Natural Language
(Alois Knoll, Bernd Hildebrand, Jianwei Zhang)

96-03 Correctness in System Engineering
(Peter Ladkin)

96-04 An Algebraic Approach to General Boolean Constraint Problems
(Hans-Werner Güsgen, Peter Ladkin)

96-05 Future University Computing Resources
(Peter Ladkin)

96-06 Lazy Cache Implements Complete Cache
(Peter Ladkin)

96-07 Formal but Lively Buffers in TLA+
(Peter Ladkin)

96-08 The X-31 and A320 Warsaw Crashes: Whodunnit?
(Peter Ladkin)

96-09 Reasons and Causes
(Peter Ladkin)

96-10 Comments on Confusing Conversation at Cali
(Dafydd Gibbon, Peter Ladkin)

96-11 On Needing Models
(Peter Ladkin)

96-12 Formalism Helps in Describing Accidents
(Peter Ladkin)

96-13 Explaining Failure with Tense Logic
(Peter Ladkin)

96-14 Some Dubious Theses in the Tense Logic of Accidents
(Peter Ladkin)

96-15 A Note on a Note on a Lemma of Ladkin
(Peter Ladkin)

96-16 News and Comment on the AeroPeru B757 Accident
(Peter Ladkin)

97-01 Analysing the Cali Accident With a WB-Graph
(Peter Ladkin)

97-02 Divide-and-Conquer Multiple Sequence Alignment
(Jens Stoye)

97-03 A System for the Content-Based Retrieval of Textual and Non-Textual Documents Based on Natural Language Queries
(Alois Knoll, Ingo Glöckner, Hermann Helbig, Sven Hartrumpf)

97-04 Rose: Generating Sequence Families
(Jens Stoye, Dirk Evers, Folker Meyer)

97-05 Fuzzy Quantifiers for Processing Natural Language Queries in Content-Based Multimedia Retrieval Systems
(Ingo Glöckner, Alois Knoll)

97-06 DFS – An Axiomatic Approach to Fuzzy Quantification
(Ingo Glöckner)

98-01 Kognitive Aspekte bei der Realisierung eines robusten Robotersystems für Konstruktionsaufgaben
(Alois Knoll, Bernd Hildebrandt)

98-02 A Declarative Approach to the Development of Dynamic Programming Algorithms, applied to RNA Folding
(Robert Giegerich)

98-03 Reducing the Space Requirement of Suffix Trees
(Stefan Kurtz)

99-01 Entscheidungskalküle
(Axel Saalbach, Christian Lange, Sascha Wendt, Mathias Katzer, Guillaume Dubois, Michael Höhl, Oliver Kuhn, Sven Wachsmuth, Gerhard Sagerer)

99-02 Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems
(Enno Ohlebusch)

99-03 A Framework for Evaluating Approaches to Fuzzy Quantification
(Ingo Glöckner)

99-04 Towards Evaluation of Docking Hypotheses using elastic Matching
(Steffen Neumann, Stefan Posch, Gerhard Sagerer)

99-05 A Systematic Approach to Dynamic Programming in Bioinformatics. Part 1 and 2: Sequence Comparison and RNA Folding
(Robert Giegerich)

99-06 Autonomie für situierte Robotersysteme – Stand und Entwicklungslinien
(Alois Knoll)

2000-01 Advances in DFS Theory
(Ingo Glöckner)

2000-02 A Broad Class of DFS Models
(Ingo Glöckner)

2000-03 An Axiomatic Theory of Fuzzy Quantifiers in Natural Languages
(Ingo Glöckner)

2000-04 Affix Trees
(Jens Stoye)

2000-05 Computergestützte Auswertung von Spektren organischer Verbindungen
(Annika Büscher, Michaela Hohenner, Sascha Wendt, Markus Wiesecke, Frank Zöllner, Arne Wegener, Frank Bettenworth, Thorsten Twellmann, Jan Kleinlützum, Mathias Katzer, Sven Wachsmuth, Gerhard Sagerer)

2000-06 The Syntax and Semantics of a Language for Describing Complex Patterns in Biological Sequences
(Dirk Strothmann, Stefan Kurtz, Stefan Gräf, Gerhard Steger)

2000-07 Systematic Dynamic Programming in Bioinformatics (ISMB 2000 Tutorial Notes)
(Dirk J. Evers, Robert Giegerich)

2000-08 Difficulties when Aligning Structure Based RNAs with the Standard Edit Distance Method
(Christian Büschking)

2001-01 Standard Models of Fuzzy Quantification
(Ingo Glöckner)

2001-02 Causal System Analysis
(Peter B. Ladkin)

2001-03 A Rotamer Library for Protein-Protein Docking Using Energy Calculations and Statistics
(Kerstin Koch, Frank Zöllner, Gerhard Sagerer)

2001-04 Eine asynchrone Implementierung eines Microprozessors auf einem FPGA
(Marco Balke, Thomas Dettbarn, Robert Homann, Sebastian Jaenicke, Tim Köhler, Henning Mersch, Holger Weiss)

2001-05 Hierarchical Termination Revisited
(Enno Ohlebusch)

2002-01 Persistent Objects with O2DBI
(Jörn Clausen)

2002-02 Simulation von Phasenübergängen in Proteinmonoschichten
(Johanna Alichniewicz, Gabriele Holzschneider, Morris Michael, Ulf Schiller, Jan Stallkamp)

2002-03 Lecture Notes on Algebraic Dynamic Programming 2002
(Robert Giegerich)

2002-04 Side chain flexibility for 1:n protein-protein docking
(Kerstin Koch, Steffen Neumann, Frank Zöllner, Gerhard Sagerer)

2002-05 ElMaR: A Protein Docking System using Flexibility Information
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-06 Calculating Residue Flexibility Information from Statistics and Energy based Prediction
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-07 Fundamentals of Fuzzy Quantification: Plausible Models, Constructive Principles, and Efficient Implementation
(Ingo Glöckner)

2002-08 Branching of Fuzzy Quantifiers and Multiple Variable Binding: An Extension of DFS Theory
(Ingo Glöckner)

2003-01 On the Similarity of Sets of Permutations and its Applications to Genome Comparison
(Anne Bergeron, Jens Stoye)

2003-02 SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry
(Sebastian Böcker)

2003-03 From RNA Folding to Thermodynamic Matching, including Pseudoknots
(Robert Giegerich, Jens Reeder)

2003-04 Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt
(Sebastian Böcker)

2003-05 Systematic Investigation of Jumping Alignments
(Constantin Bannert)