BIOINFORMATICS
Vol. 18 Suppl. 1 2002
Pages S312–S320

Efficient multiple genome alignment

Michael Höhl, Stefan Kurtz and Enno Ohlebusch

Faculty of Technology, University of Bielefeld, PO Box 10 01 31, Bielefeld, D-33501, Germany

Received on January 22, 2002; revised and accepted on March 29, 2002

ABSTRACT
Motivation: To allow a direct comparison of the genomic DNA sequences of sufficiently similar organisms, there is an urgent need for software tools that can align more than two genomic sequences.
Results: We developed new algorithms and a software tool 'Multiple Genome Aligner' (*MGA* for short) that efficiently computes multiple genome alignments of large, closely related DNA sequences. For example, it can align 85% percent of the complete genomes of six human adenoviruses (average length 35 305 bp.) in 159 seconds. An alignment of 74% of the complete genomes of three of strains of *E. coli* (lengths: 5 528 445; 5 498 450; 4 639 221 bp.) is produced in 30 minutes.
Availability: The software *MGA* is available free of charge for non-commercial research institutions. For details see http://bibiserv.techfak.uni-bielefeld.de/mga/.
Contact: kurtz@techfak.uni-bielefeld.de;
enno@techfak.uni-bielefeld.de
Keywords: genome comparison; multiple alignment; efficient algorithms; graph algorithms; suffix trees.

INTRODUCTION

The DNA sequences of entire genomes are being determined at a rapid rate. Recently, there is a growing scientific interest in sequencing different strains of bacteria and other closely related organisms. For example, the genomes of several strains of *Escherichia coli* and *Chlamydophila pneumoniae* have already been completely sequenced. When the genomic DNA sequences of closely related organisms become available, one of the first questions researchers ask is how the genomes align. Although a number of software tools aimed at comparing *two* genomic DNA sequences exist (Delcher *et al.*, 1999; Batzoglu *et al.*, 2000; Kent and Zahler, 2000), there is an immediate need for 'reliable and automatic software for aligning three or more genomic sequences' (Miller, 2001). To our knowledge, this paper presents the first software tool for this task.

It should be pointed out that an alignment of the genomes (genomic DNA sequences) of several organisms makes sense only if the organisms are closely related. Otherwise, one has to take genome rearrangements into account. In this case, one would first try to identify syntenic regions and then align these instead of the entire genomes.

Numerous multiple alignment methods have been published in the bioinformatics literature, but the vast literature is almost entirely geared toward comparison of protein sequences (because in the past, genomes of several similar organisms were not available). Virtually all of the global multiple alignment methods used in practice are variants of either (1) 'iterative pairwise alignment' or (2) 'anchor-based multiple alignment'. Methods from category (1) follow a general strategy of iteratively merging two multiple alignments of two disjoint subsets of sequences into a single multiple alignment of the union of those subsets. If, as usually done, the global pairwise alignment of two genomic DNA sequences S_1 and S_2 is computed by standard dynamic programming algorithms (which requires $O(|S_1| \cdot |S_2|)$ time, where $|S|$ denotes the length of a sequence S), then such an iterative method cannot be used in practice to align DNA sequences of entire genomes.

Methods from category (2) try to identify substrings of the sequences under consideration that are very likely be part of a global multiple alignment. These substrings form 'anchors' in the sequences to be aligned. Consequently, a method of category (2) first aligns those anchors and subsequently closes the gaps, i.e., it aligns the substrings between the anchors. The latter can be done by applying the same method recursively to the gaps—yielding a divide and conquer method—but also by any other alignment method. Anchor-based alignment methods are well suited for aligning very long sequences. The software tool *MUMmer* (Delcher *et al.*, 1999) is an impressive implementation of such a method. In our opinion, it is a major breakthrough toward the solution of the alignment problem of *two* sufficiently similar genomic DNA sequences. Like any other existing method, however, *MUMmer* cannot align more than two sequences.

In this paper, we present the anchor-based method *MGA* that is capable of aligning three or more genomes. The method is divided into three phases. In the first phase, a novel algorithm detects all *maximal multiple*

© Oxford University Press 2002

exact matches (*multiMEM*s) whose length exceeds a given threshold. Roughly speaking, a *multiMEM* is a sequence that occurs in all genomes to be aligned and cannot simultaneously be extended to the left or right in every genome. Our novel algorithm uses $O(kn + r)$ time in the worst case, where k is the number of genomes, n is their total length, and r is the number of right maximal multiple exact matches. Its efficient implementation is based on the virtual suffix tree (Kasai *et al.*, 2001) which requires only 5 bytes per input character. By contrast, other implementations of the suffix tree require much more space. For example, our experiments show that *MUMmer*'s implementation uses 50 bytes per input character, see the section on Experiments.

In the second phase, *MGA* computes the 'anchors', consisting of the longest non-overlapping sequence of *multiMEM*s that occur in the same order in each of the genomes. This is achieved in time $O(k \cdot m^2)$ in the worst case, where m is the number of *multiMEM*s, by using an efficient graph algorithm that seems to be folk knowledge (Arkin and Silverberg, 1987; Vingron and Argos, 1989; Gusfield, 1997).

In the third phase, *MGA* closes the gaps between the anchors. First, this is done by recursively applying the same method a certain number of times, thereby lowering the length threshold for the *multiMEM*s. The gaps that are left over are handled as follows: Short gaps are closed by the multiple sequence alignment program ClustalW (Thompson *et al.*, 1994), which is a widely used implementation of an iterative multiple alignment method. Long gaps remain unaligned in order to cope with long insertions, deletions, etc.

MGA is not only a multiple alignment tool. It can also efficiently align two genomes. Applied to two sequences, it provides several important advantages over *MUMmer*:

- While *MUMmer* computes maximal unique matches (*MUM*s, i.e., matches that occur once in each genome) to anchor alignments, *MGA* uses maximal exact matches (*MEM*s). These are more general than *MUM*s, as they are allowed to occur more than once in each genome. However, if the user requests *MUM*s, then *MGA* also delivers them as anchors.
- The computation of *MEM*s or *MUM*s is based on a virtual suffix tree. This index structure requires much less space than *MUMmer*s implementation of the suffix tree (see above), and it delivers matches faster.
- Despite the more general notion of matches, *MGA* always delivers the anchors for two sequences in $O(m \log m)$ time, where m is the number of *MUM*s or *MEM*s, using the algorithm of (Joseph *et al.*, 1992).
- To close the gaps, *MGA* applies the greedy alignment algorithm of (Ukkonen, 1985). This aligns two sequences of length r and r' in $O(e \cdot \min(r, r'))$ time, where e is their edit distance. Thus, it gives a speedup over the $O(r \cdot r')$ standard dynamic programming algorithm used in *MUMmer* for the same task, provided the sequences to be aligned are similar.
- While *MUMmer* restricts the length of the gaps to be closed, *MGA* aligns gaps of arbitrary length if they are sufficiently similar according to a percent identity score specified by the user.

Our experiments show that *MGA* is always faster than *MUMmer* (for two strains of *E. coli* over five times faster), using less than 18% of the space (for two strains of *E. coli* only 8% of the space).

The rest of the paper is organized as follows. We start with brief descriptions of existing tools for aligning pairs of large genomic DNA sequences. After introducing some basic definitions, the section on Algorithms and Data Structures outlines the three phases of our tool *MGA*. The new algorithm for finding *multiMEM*s is described in detail, including some important implementation details. Finally, we present our experimental results.

RELATED WORK

To the best of our knowledge, there is no other software tool that can align more than two DNA sequences of the size of entire genomes (e.g., of bacteria or of eukaryotes). For this reason, we must restrict our discussion of related work to software tools that are aimed at aligning *two* genomic DNA sequences. We will briefly describe the software tools *MUMmer*, *GLASS*, and *WABA*. The well-known web-based tool *PipMaker* constructs alignments using *blastz* (Miller, 2001), but there is no description of how this is done. For this reason, we do not further discuss *PipMaker*. Recently Buhler (2001) described a novel approach to align large sequences, but no software tool incorporating this approach seems to be available.

MUMmer

As already mentioned, *MUMmer* is an anchor-based tool. It proceeds in the following three phases: (1) A maximal unique match (*MUM*) decomposition of the two genomes S_1 and S_2 is computed. A *MUM* is a sequence that occurs exactly once in genome S_1 and once in genome S_2, and is not contained in any longer such sequence. Using the suffix tree of $S_1\$S_2$, *MUM*s can be computed in $O(n)$ time and space, where $n = |S_1\$S_2|$ and \$ is a symbol neither occurring in S_1 nor in S_2. (2) The matches found in the *MUM* decomposition are sorted, and the longest possible set of *MUM*s that occur in the same order in both genomes is extracted, yielding the anchors. This can be done using the algorithm of Jacobson and Vo (1992) to find the heaviest increasing subsequence (HIS) of a sequence of

weighted integers. This phase takes $O(m \cdot \log m)$ time in the worst case, where m is the number of *MUM*s. Note, however, that *MUMmer* actually uses a simpler $O(m^2)$ time dynamic programming algorithm. (3) The gaps in the anchor alignment that have length less than or equal to a given limit (5000 bp. is the default in *MUMmer*) are closed with a standard dynamic programming algorithm (Needleman and Wunsch, 1970). For one gap consisting of two sequences of length r and r', this takes $O(r \cdot r')$ time and $O(\min(r, r'))$ space. Gaps that are longer than the limit remain unaligned.

The restriction of using only *MUM*s as anchors seems unnecessary and is not justified. Exact matches occurring more than once in a genome may also be meaningful. Furthermore, the coverage of the sequences increases if other matches are taken into account. Moreover, the use of the HIS algorithm of (Jacobson and Vo, 1992) for chaining the *MUM*s is not adequate. This is because the resulting chain of *MUM*s may contain overlapping *MUM*s, which in turn may lead to inconsistencies (i.e., it may not be possible to find an alignment that is consistent with all selected *MUM*s). *MUMmer* takes an ad hoc approach to handle this: It simply removes the overlapping parts from the *MUM*s, see Figure 1.

GLASS

The acronym *GLASS* stands for *GL*obal *A*lignment *S*y*S*tem (Batzoglu *et al.*, 2000). It has been developed for the preprocessing step of the gene prediction tool *ROSETTA*. *GLASS* is based on a model of eukaryotic DNA sequences containing long, weakly conserved introns and short, strongly conserved exons. Thus, the model is not suitable for prokaryotes.

GLASS also falls into the category of anchor-based alignment tools. It consists of five steps (1)–(5), which deliver a partial alignment of the two input sequences, possibly leaving some regions unaligned.

(1) All pairs of exact matching k-mers (i.e., strings of length k) in the two input sequences S_1 and S_2 are searched for. Initially, $k = 20$. (2) For a given pair of matching k-mers (w_1, w_2), its score $s = s_l + s_r$ is determined as follows: A dynamic programming algorithm is applied to the 12 nucleotides to the left of w_1 and w_2, yielding score s_l, and to the right of w_1 and w_2, yielding score s_r. (3) The highest scoring sequence of k-mers that occur in the same order in both DNA sequences is computed by a dynamic programming algorithm. (4) All k-mer matches in the resulting sequence whose score s is below a given threshold are removed. Furthermore, inconsistent overlapping k-mers are also removed; (w_1, w_2) and (w_1', w_2') are inconsistent if the overlap of w_1 and w_1' differs from the overlap of w_2 and w_2'. (5) The resulting k-mers serve as anchors in the alignment of S_1 and S_2. Steps (1)–(5) are recursively applied to the gaps (all unaligned regions between the anchors) with decreasing values of k, namely 15, 12, 9, 8, 7, 6, 5. Finally, all remaining gaps are aligned by standard dynamic programming.

A major drawback of *GLASS* is the huge space requirement. For example, an alignment of two DNA sequences of human and mouse (length 222 930 bp. and 227 538 bp., see the first sequence set in Experiments) is produced in about 14 minutes using 1.14 gigabytes of main memory. It takes 38 minutes to align a similar pair of sequences of twice the length, using 2.05 gigabytes. Furthermore, it seems that *GLASS* does not take advantage of long identical regions in sequences. For example, to deliver an alignment of the initial 200 000 bp. of two strains of *E. coli*, *GLASS* requires more than 25 hours of computation time (we stopped the job after 25 hours).

WABA

The acronym *WABA* stands for *W*obble *A*ware *B*ulk *A*ligner (Kent and Zahler, 2000). The key feature of *WABA* is that wobble bases are treated differently from other bases. The third base in a codon is called *wobble base* because mutations in this base are often silent in the sense that they do not change the corresponding amino acid (due to the redundancy of the genetic code). *WABA* has been developed specifically for separately aligning 229 different sequences from *C. briggsae* (8 megabases total length, 34 722 bp. average length) against 97 megabases of the *C. elegans* genome.

WABA can be divided into three phases. (1) The smaller of the two input sequences is broken into short, overlapping sequence fragments. Then homologies between the short sequence fragments and the other input sequence are searched for. (2) Homologous regions are aligned in an extended window using a pairwise hidden Markov model (Durbin *et al.*, 1998). (3) If any two of these local alignments overlap by at least 15 bp. and are identical in the overlapping region, then they are merged into one larger alignment.

The homology search in the first phase is carried out in gapped *BLAST*-like fashion (Altschul *et al.*, 1997) with one modification. In *WABA*, *high scoring pairs* are not required to match exactly but may contain a mismatch every three bases. This is justified by the fact that homologous regions in two related DNA sequences are most likely protein coding regions. In these, most point mutations occur in the third base of a codon.

The running time of *WABA* for aligning the 8 megabases of *C. briggsae* and the 97 megabases of *C. elegans* was about 12 days on a Pentium III 450 MHz computer (20 hours + 11 days + 15 minutes for the respective phases). This renders the approach impractical for larger genomes.

BASIC DEFINITIONS

We index the characters of a sequence from 0. That is, a sequence S of length n is written as $S = S[0]S[1]\dots S[n-1] = S[0\dots n-1]$. A *prefix* of S is a sequence $S[0]\dots S[i]$ for some $i \in [0, n-1]$. A *suffix* of S is a sequence $S[j]\dots S[n-1]$ for some $j \in [0, n-1]$. Consider a set $\{G_0, \dots, G_{k-1}\}$ of $k \geq 2$ sequences, the genomes. A *multiple exact match* is a $(k+1)$-tuple $(l, p_0, p_1, \dots, p_{k-1})$ such that $l > 0$, $p_q \in [0, |G_q| - l]$, and $G_q[p_q \dots p_q + l - 1] = G_{q'}[p_{q'} \dots p_{q'} + l - 1]$ for all $q, q' \in [0, k-1]$. A multiple exact match is *left maximal* if for at least one pair $(q, q') \in [0, k-1] \times [0, k-1]$, we either have $p_q = 0$, or $p_{q'} = 0$, or $G_q[p_q - 1] \neq G_{q'}[p_{q'} - 1]$. A multiple exact match is *right maximal* if for at least one pair $(q, q') \in [0, k-1] \times [0, k-1]$, we either have $p_q + l = |G_q|$, or $p_{q'} + l = |G_{q'}|$, or $G_q[p_q + l] \neq G_{q'}[p_{q'} + l]$. A multiple exact match is *maximal* if it is left maximal and right maximal. A maximal multiple exact match is also called *multiMEM*. For $k = 2$, we use the notion *MEM*. Roughly speaking, a *multiMEM* is a sequence of length l that occurs in all sequences $G_0, \dots, G_{k-1}$ (at positions $p_0, \dots, p_{k-1}$), and cannot simultaneously be extended to the left or to the right in every sequence.

ALGORITHMS AND DATA STRUCTURES

Computing *multiMEM*s

Let $\$_0, \dots, \$_{k-1}$ be pairwise different symbols not occurring in any G_q and let $S = G_0\$_0G_1\$_1 \dots G_{k-2}\$_{k-2}G_{k-1}$. That is, $\$_0, \dots, \$_{k-2}$ are used to separate the sequences in the concatenation S. $\$_{k-1}$ will be used as a sentinel attached to the end of S, see below.

Let $n = |S| = k - 1 + \sum_{q=0}^{k-1} |G_q|$. For any $i \in [0, n]$, let $S_i = S[i \dots n-1]\$_{k-1}$ denote the ith non-empty suffix of $S\$_{k-1}$. Hence $S_n = \$_{k-1}$.

Define $t_0 = 0$ and $t_q = t_{q-1} + |G_{q-1}| + 1$ for any $q \in [1, k]$. t_q is the start position of G_q in S for $q \in [0, k-1]$. Let $i \in [0, n-1]$ such that $S[i] \notin \{\$_0, \dots, \$_{k-2}\}$. We define two functions σ and ρ as follows:

- $\sigma(i) = q$ if and only if $t_q \leq i < t_{q+1} - 1$
- $\rho(i) = i - t_{\sigma(i)}$

That is, position i in S is identified with the relative position $\rho(i)$ in sequence $G_{\sigma(i)}$.

We consider trees whose edges are labelled by non-empty sequences. For each character a, every node α in these trees has at most one a-edge $\alpha \xrightarrow{av} \beta$ for some sequence v and some node β. Consider a tree T and let α be a node in T. We denote α by $\overline{w}$ if and only if w is the concatenation of the edge labels on the path from the root of T to α. A sequence w *occurs* in T if and only if T contains a node $\overline{wv}$, for some sequence v. The *suffix tree* for S, denoted by ST, is the tree T with the following properties: (1) each node is either the root, a leaf or a branching node, and (2) a sequence w occurs in T if and only if w is a substring of $S\$_{k-1}$.

There is a one-to-one correspondence between the leaves of the suffix tree and the non-empty suffixes of $S\$_{k-1}$: Leaf $\overline{S_i}$ corresponds to suffix S_i and vice versa. For this reason, we sometimes denote $\overline{S_i}$ by $\ell(i)$. It is well known that a suffix tree can be constructed in linear time and linear space (Weiner, 1973; McCreight, 1976).

For any node $\overline{u}$ of ST (including the leaves), let $\mathcal{P}_{\overline{u}}$ be the set of positions i such that u is a prefix of S_i. In other words, $\mathcal{P}_{\overline{u}}$ is the set of positions in S where the sequence u starts. We divide $\mathcal{P}_{\overline{u}}$ into disjoint and possibly empty position sets according to σ: For any $q \in [0, k-1]$, we define $\mathcal{P}_{\overline{u}}(q) = \{i \in \mathcal{P}_{\overline{u}} \mid \sigma(i) = q\}$, i.e., $\mathcal{P}_{\overline{u}}(q)$ is the set of positions i in S where u starts and i occurs in genome G_q.

We now describe an algorithm to compute all *multiMEM*s, using the suffix tree for S. Our algorithm computes position sets by processing the edges of the suffix tree in a bottom-up strategy. That is, the edge leading to node $\overline{u}$ is processed only after all edges in the subtree below $\overline{u}$ have been processed.

If $\overline{u}$ is a leaf, say $\ell(i)$, then compute $\mathcal{P}_{\ell(i)}(q) = \{i\}$ if $\sigma(i) = q$, and $\mathcal{P}_{\ell(i)}(q) = \emptyset$ otherwise. Now suppose $\overline{u}$ is a branching node. The edges outgoing from $\overline{u}$ are processed from left to right. Consider an edge $\overline{u} \longrightarrow \overline{w}$. Due to the bottom-up strategy, $\mathcal{P}_{\overline{w}}(q)$ is already computed for any $q \in [0, k-1]$. However, only a subset of $\mathcal{P}_{\overline{u}}(q)$ has been computed since only the first, say j, edges outgoing from $\overline{u}$ have been processed. We denote the corresponding subset of $\mathcal{P}_{\overline{u}}(q)$ by $\mathcal{P}^j_{\overline{u}}(q)$. $\overline{u} \longrightarrow \overline{w}$ is processed in the following way: At first *multiMEM*s are output by combining $\mathcal{P}^j_{\overline{u}}$ with $\mathcal{P}_{\overline{w}}$. In particular, all $(k+1)$-tuples $(l, p_0, p_1, \dots, p_{k-1})$ satisfying the following conditions are enumerated:

(1) $l = |u|$

(2) $p_q \in \mathcal{P}^j_{\overline{u}}(q) \cup \mathcal{P}_{\overline{w}}(q)$ for any $q \in [0, k-1]$

(3) $p_q \in \mathcal{P}^j_{\overline{u}}(q)$ for at least one $q \in [0, k-1]$.

(4) $p_q \in \mathcal{P}_{\overline{w}}(q)$ for at least one $q \in [0, k-1]$.

By definition of $\mathcal{P}_{\overline{u}}$, u occurs at the positions $p_0, p_1, \dots, p_{k-1}$ in S. Moreover, for each $q \in [0, k-1]$, $\rho(p_q)$ is a relative position of u in G_q. Hence $(l, \rho(p_0), \rho(p_1), \dots, \rho(p_{k-1}))$ is a multiple exact match. Conditions (3) and (4) guarantee that not all positions are exclusively taken from $\mathcal{P}^j_{\overline{u}}(q)$ or from $\mathcal{P}_{\overline{w}}(q)$. Hence at least two of the positions in $\{p_0, p_1, \dots, p_{k-1}\}$ are taken from different subtrees of $\overline{u}$. This implies right maximality. To guarantee left maximality, we

reject $(l, \rho(p_0), \rho(p_1), \ldots, \rho(p_{k-1}))$, if $p_0 > 0$ and $S[p_0 - 1] = S[p_1 - 1] = \cdots = S[p_{k-1} - 1]$.

As soon as for the current edge $\overline{u} \longrightarrow \overline{w}$ the *multiMEM*s are enumerated, our algorithm adds $\mathcal{P}_{\overline{w}}(q)$ to $\mathcal{P}^{j}_{\overline{u}}(q)$ to obtain position sets $\mathcal{P}^{j+1}_{\overline{u}}(q)$ for all $q \in [0, k-1]$. That is, the position sets are inherited from node $\overline{w}$ to the parent node $\overline{u}$. Finally, $\mathcal{P}_{\overline{u}}(q)$ is obtained as soon as all edges outgoing from $\overline{u}$ are processed.

To analyse the efficiency of our algorithm we describe how to implement position sets and how to accomplish the bottom-up traversal.

Implementation of position sets. Our algorithm performs two operations on position sets: Enumeration of multiple exact matches by combining position sets and adding up position sets. A position set $\mathcal{P}_{\overline{u}}(q)$ is the union of position sets from the subtrees below $\overline{u}$. Recall that we considered processing an edge $\overline{u} \longrightarrow \overline{w}$. If the edges to the children of $\overline{w}$ have been processed, the position sets of the children are obsolete. Hence it is not required to copy position sets. At any time of the algorithm, each position is included in exactly one position set. For each node we store k references to k possibly empty position sets. Hence, the space requirement for the position sets is $O(kn)$. The union operation for the position sets can be implemented in constant time, if we use linked lists. For each node, we have $O(k)$ union operations. Since there are $O(n)$ edges in the suffix tree, the union and add operations thus require $O(kn)$ time.

Each combination of position sets requires to enumerate the Cartesian product

$$\mathop{\times}_{q=0}^{k-1} (\mathcal{P}_{\overline{u}}(q) \cup \mathcal{P}_{\overline{w}}(q)) \setminus \left(\left(\mathop{\times}_{q=0}^{k-1} \mathcal{P}_{\overline{u}}(q) \right) \cup \left(\mathop{\times}_{q=0}^{k-1} \mathcal{P}_{\overline{w}}(q) \right) \right).$$

This can be done in time proportional to its size. With the enumeration of the Cartesian product we maintain the size of the set $C_{\text{left}} = \{S[p_q - 1] \mid q \in [0, k-1], p_q > 0\}$. This takes constant time per enumeration step. Now an enumerated right maximal multiple exact match $(l, \rho(p_0), \rho(p_1), \ldots, \rho(p_{k-1}))$ is left maximal if and only if $p_0 = 0$ or $|C_{\text{left}}| \geq 2$. Thus the left maximality can be decided in constant extra time. Altogether the position sets are maintained and repeats are output in $O(kn + r)$ time, where r is the number of right maximal multiple exact matches. This time bound can be further improved by dividing each $\mathcal{P}_{\overline{u}}(q)$ into subsets according to the character to the left of the corresponding position. This leads to an algorithm that enumerates *multiMEM*s directly without any further test for left maximality, using $O(|\Sigma|kn + m)$ time where m is the number of *multiMEM*s and Σ is the alphabet.

Implementation of the bottom-up traversal. We have described our algorithm based on suffix trees, because these are widely known in the bioinformatics community. However, we actually implemented our algorithm on 'virtual suffix trees', introduced in Kasai *et al.* (2001). This new data structure consists of two tables, suftab and lcptab, which allow to simulate the bottom-up traversal of the suffix tree for S, as required for our algorithm. The two tables are defined as follows:

- suftab is a table of length $n + 1$ indexed from 0 to n such that $S_{\mathsf{suftab}[0]}, S_{\mathsf{suftab}[1]}, \ldots, S_{\mathsf{suftab}[n]}$ is the sequence of suffixes of $S\$_{k-1}$ in ascending lexicographic order.
- lcptab is a table of length n indexed from 1 to n such that, for any $i \in [1, n]$, lcptab$[i]$ is the length of the longest common prefix of the suffixes $S_{\mathsf{suftab}[i-1]}$ and $S_{\mathsf{suftab}[i]}$.

Table suftab is also called suffix array (Manber and Myers, 1993). The simulation algorithm of Kasai *et al.* (2001) runs in $O(n)$ time. The same time bound can be achieved for a bottom-up traversal of the suffix tree. More importantly, the two tables require much less space than the suffix tree. Table suftab can be implemented in $4(n + 1)$ bytes. Table lcptab only requires n bytes, due to the fact that the values in this table are expected to be small. Tables suftab and lcptab can be constructed in $O(kn)$ time and space using a suffix tree, see Gusfield (1997).

Altogether our algorithm runs in $O(kn + r)$ time and $O(kn)$ space, where r is the number of right maximal multiple exact matches. Note that for $k = 2$, our algorithm is very similar to the algorithm of Gusfield (1997, page 147) to compute maximal repeated pairs.

Selecting an optimal set of *multiMEM*s

This problem is an instance of the following more general problem from computational geometry. Let a k-dimensional Euclidean space with rectangular coordinate system be given. We consider k-dimensional rectangular solids, in which the edges of the solids are parallel with the coordinates. Each such rectangular solid can be characterized by a point $p = (p_1, \ldots, p_k)$ and the lengths $l_1, \ldots, l_k$ of its edges. (In the plane, $p = (p_1, p_2)$ is the lower-left corner of a rectangle, l_1 is the length of the edge parallel with the x-coordinate, and l_2 is the length of the edge parallel with the y-coordinate.) Thus, a rectangular solid rs consists of $2k$ components, viz. $rs = (p_1, \ldots, p_k, l_1, \ldots, l_k)$. Moreover, a rectangular solid rs has an associated weight $w(rs)$, which, for example, could be the volume of rs.

For two rectangular solids

$$rs = (p_1, \ldots, p_k, l_1, \ldots, l_k) \text{ and}$$
$$rs' = (p'_1, \ldots, p'_k, l'_1, \ldots, l'_k)$$

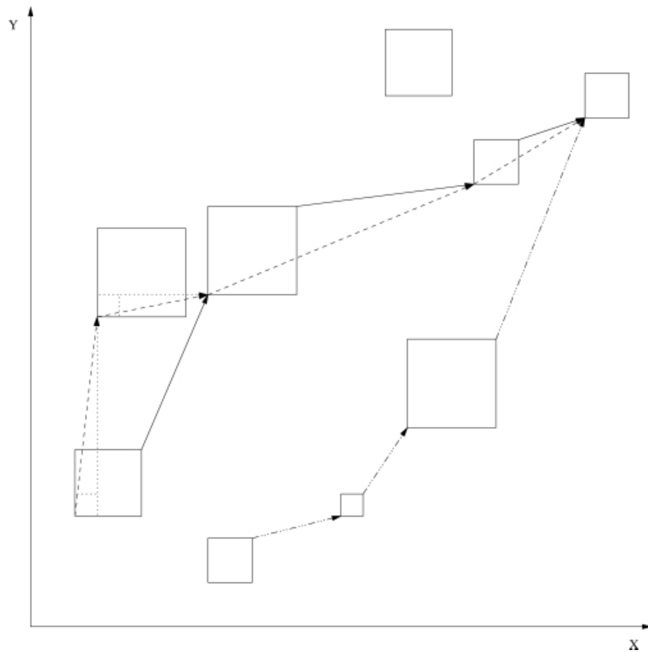

Fig. 1. A maximum weight chain (drawn by solid arrows) of *MEM*s (represented by squares) as delivered by the algorithm applied in phase (2) of *MGA*. The rightmost chain (drawn by dashed-dotted arrows) is suboptimal. Not all chains are shown. Suppose that all *MEM*s are *MUM*s. A HIS-chain of these *MUM*s (depicted by dashed arrows), as computed by *MUMmer*, consists of five squares. The HIS-chain includes a square that overlaps with the squares below it and to its right-hand side. To guarantee that the alignment is consistent, *MUMmer* removes the overlaps, retaining shortened matches (drawn as dotted squares). As a result, the HIS-chain containing shortened matches is suboptimal.

we define $rs \prec rs'$ if and only if for all $i \in [1, k]$ the inequality $p_i + l_i < p_i'$ holds.

In what follows, let a set of rectangular solids $\{rs_1, \ldots, rs_m\}$ be given. A subset $C = \{rs_{i_1}, \ldots, rs_{i_q}\}$ of $\{rs_1, \ldots, rs_m\}$ satisfying $rs_{i_1} \prec rs_{i_2} \prec \cdots \prec rs_{i_q}$ is called a *chain*. The weight of chain C is $\sum_{j=1}^{q} w(rs_{i_j})$. We want to find a chain with maximum weight among all chains. To do so, we transform the problem into a graph theoretical problem. We construct a weighted directed graph $G = (V, E)$ with vertices $V = \{start, rs_1, \ldots, rs_m, stop\}$. The set of edges E is characterized as follows: There is an edge $start \rightarrow rs_j$ with weight 0 for $j \in [1, m]$, an edge $rs_i \rightarrow rs_j$ with weight $w(rs_i)$ if $rs_i \prec rs_j$, and an edge $rs_i \rightarrow stop$ with weight $w(rs_i)$ for $i \in [1, m]$. Constructing the graph takes $O(k \cdot m^2)$ time. Note that the graph is acyclic. A maximum weight chain of rectangular solids corresponds to a path with maximum weight from vertex $start$ to vertex $stop$ in the graph. Because the graph is acyclic, such a path can be computed in $\Theta(|V| + |E|)$ time (Lawler, 1976; Cormen *et al.*, 1990, Section 25.4). All in all, a chain with maximum weight can be computed in $O(k \cdot m^2)$ time because $|V| + |E| \in O(m^2)$.

Viewing a *multiMEM* $(l, p_0, \ldots, p_{k-1})$ as a rectangular solid $(p_0, \ldots, p_{k-1}, l, \ldots, l)$ in the k-dimensional Euclidean space with associated weight l, the longest non-overlapping set of matches that occur in the same order in every genome $G_0, \ldots, G_{k-1}$ can be determined by computing a chain with maximum weight among all chains of rectangular solids, see Figure 1 for the case $k = 2$. As a consequence, the second phase of algorithm *MGA* takes $O(k \cdot m^2)$, where m is the number of *multiMEM*s.

A more efficient chaining algorithm was devised by (Myers and Miller, 1995). It runs in $O(m \cdot \log^k m)$ time and $O(km \cdot \log^{k-1} m)$ space, provided $k < \log m$. It remains unclear whether this algorithm will give a speedup in practice.

Closing the gaps

In the third phase, one has to close the gaps in the alignment by computing a multiple sequence alignment. We choose the program *ClustalW* (Thompson *et al.*, 1994) for this task. *ClustalW* is a widely used implementation of profile-based progressive multiple alignment. It is easy to interface other multiple alignment programs with our software.

EXPERIMENTAL RESULTS

We used the following sets of DNA sequences in our experiments:

Human/Mouse: These are two sequences from *Homo sapiens* and *Mus musculus* consisting of 222 930 bp. and 227 538 bp., respectively. The GenBank accession numbers are `U47924` and `AC002397`.

Mycoplasma: The complete genomes of *Mycoplasma pneumoniae* M129 (816 394 bp., `NC_000912`) and of *Mycoplasma genitalium* G37 (580 074 bp., `L43967`).

Tuberculosis: The complete genomes of two strains of *Mycobacterium tuberculosis* (4 411 529 bp., `AL123456`; 4 403 836 bp., `AE000516`).

Streptococcus: The complete genomes of two strains of *Streptococcus pneumoniae* (2 160 837 bp., `NC_003028.1`; 2 038 615 bp., `NC_003098.1`).

E. coli 2: The complete genomes of two strains of *Escherichia coli* (K-12 MG1655, 4 639 221 bp., `U00096.1`; O157:H7, 5 528 445 bp., `AE005174.1`).

Adenovirus 6: Six complete genomes of human adenoviruses (35 937 bp., `NC_001405.1`; 35 935 bp., `NC_001406.1`; 34 214 bp., `NC_001454.1`; 34 125 bp.,

NC_001460.1; 35 100 bp., NC_002067.1; 36 521 bp., NC_003266.1).

E. coli 3: E. coli 2 plus the complete genome of another strain of E. coli (O157:H7, 5 498 450 bp., BA000007).

C. pneumoniae 3: The complete genomes of three strains of *Chlamydophila pneumoniae* (1 229 853 bp., AE002161; 1 226 565 bp., BA000008; 1 230 230 bp., NC_000922.1).

S. aureus 3: The complete genomes of three strains of *Staphylococcus aureus* (N315, 2 813 641 bp., NC_002745.1; Mu50, 2 878 040 bp., NC_002758.1; EMRSA-16 strain 252, 2 902 619 bp., ftp://ftp.sanger.ac.uk/pub/pathogens/sa/MRSA.dbs).

S. aureus 4: S. aureus 3 plus the complete genome of another strain of S. aureus (NCTC 8325, 2 821 905 bp., ftp://ftp.genome.ou.edu/pub/staph/).

The first three sets of sequences were also used in (Delcher *et al.*, 1999) to evaluate *MUMmer*.

In the first experiment, we applied *MUMmer* and *MGA* to the first five sets of sequences. For a fair comparison, we used options of *MGA* which reproduce the results of *MUMmer*. In particular, *MGA* also computed *MUM*s and did not apply the recursive strategy to close the gaps. Both tools aligned a gap only if its length did not exceed 1000 bp. *MGA* basically delivered the same alignments as *MUMmer* and so we do not comment on them. Small differences in the alignments (if any) are due to the ad hoc handling of overlapping *MUM*s in *MUMmer*, see Figure 1. We report on the coverage, the running time for the three phases of the programs, and the space requirement (only for the first phase, since this requires most of the space). The coverage is the percentage of each sequence, appearing in the delivered alignment (recall that the tools leave parts of the sequences unaligned). The coverage is only reported once, since it is the same for both tools.

The running time (user time plus system time) was measured on a SUN-Solaris computer with a 750 MHz SPARC-CPU and 4 GB of main memory. The results are shown in Tables 1 and 2.

Our experiment shows that *MGA* is by far superior to *MUMmer* in terms of space requirement. In the first phase, *MGA* only requires between 8 and 18% of the space. Thus *MGA* overcomes the memory bottleneck of *MUMmer*, but still has a running time advantage. This advantage can be seen in the chaining phase: Since *MUMmer* applies an $O(m^2)$ algorithm to m *MUM*s, it does not scale up for *Streptococcus* and *E. coli 2*, due to the large number of *MUM*s. The coverage of Mycoplasma is small, due to the large difference in the genome lengths, the lack of long identical matches, and the fact that we closed gaps only if they are at most of length 1000 bp. (to make the results comparable). Increasing the maximal gap length to 5000 bp. improves the coverage to 38.8% and 51.4%, respectively.

In our second experiment, we applied *MGA* to the first five sequence pairs taking *MEM*s as anchors. Gaps were closed only if the sequences in the gaps had an identity of at least 60%. The running time of all three phases was almost the same as in experiment one (data not shown). Due to the different strategy in phase (3), gaps consisting of unconserved sequences are not forced into an alignment. On the other hand, similar large sequences in gaps are aligned very efficiently. For example, in Tuberculosis we found a gap consisting of sequences of length 10 722 and 10 721 with edit distance 2. These had been excluded from a maximal chain, because they are part of a *MEM* of length 10 724 which overlaps another *MEM* of length 17 543.

In our third experiment, we applied *MGA* to the sets Adenovirus 6, E. coli 3, C. pneumoniae 3, and S. aureus 3/4 (see Table 3). The times reported are user times plus system times in minutes. The time for constructing the virtual suffix trees of the sequence sets is not included. This always requires less than 10 minutes, and only has to be done once for each sequence set. The fourth column of Table 3 shows the total running time including ClustalW, while the fifth column gives the running times excluding ClustalW, i.e., the time for computing the *multiMEM*s and chaining them. For Adenovirus 6, the alignment was constructed by recursively applying *MGA* to compute *multiMEM*s of minimal size $10, 9, \ldots, 4$. Gaps of size larger than 1000 bp. remained unaligned. For E. coli 3, *MGA* constructed the alignment by first computing *multiMEM*s of length at least 1000, and then proceeded recursively with *multiMEM*s of minimum length 20 bp. Note that most of the computation time was used by ClustalW. For C. pneumoniae 3, the alignment was constructed by recursively applying *MGA* to compute *multiMEM*s of minimal size $15, 9, 4$. Note that most of the running time was used by the first two phases of *MGA*. This is due to the lack of long *multiMEM*s and the large number of short *multiMEM*s in the genomes. For S. aureus 3 the same length thresholds were used as in E. coli 3. For S. aureus 4 the large number of *multiMEM*s requires to use the length threshold of 25 in the recursive calls. This in turn leads to a reduced running time for phases (1) and (2) compared to S. aureus 3. The coverage remains high, due to the strong similarity in the four Staphylococcus strains.

FUTURE WORK

Of course, *MGA* should be supplemented with a good interactive visualization component. This is a challenging task (see the discussion in Miller (2001)). Even for an alignment of *two* genomes, the existing solutions to this

Table 1. Comparison of *MGA* and *MUMmer* for the first five sequence pairs. We report the running time and the space requirement when both programs compute all *MUM*s of length at least ℓ

Phase (1): Computing *MUM*s of length $\geq \ell$						
sequence set	total length	ℓ	time (seconds)		memory (megabytes)	
			MGA	*MUMmer*	*MGA*	*MUMmer*
Human/Mouse	450 468	15	0.2	1.6	4	23
Mycoplasma	1 396 468	20	0.9	7	8	68
Tuberculosis	8 815 365	50	32	52	36	422
Streptococcus	4 199 452	20	3	22	19	202
E.coli 2	10 167 666	20	13	68	39	487

Table 2. Comparison of *MGA* and *MUMmer* for the first five sequence pairs. We report the number of *MUM*s, the running time used for the chaining phase, and the running time for closing the gaps. The last two columns show the coverage for the sequences, i.e., the percentage of bases, appearing in the delivered alignment. Since both tools apply the same strategy, their coverage does not differ and we only report it once

sequence set	Phase (2): Chaining			Phase (3): Closing Gaps		Coverage	
	#*MUM*s	*MGA*	*MUMmer*	*MGA*	*MUMmer*	seq1	seq2
Human/Mouse	1 037	0.02	0.04	1.9	2.3	40.4%	37.7%
Mycoplasma	433	0.01	0.01	0.7	2.9	7.4%	10.5%
Tuberculosis	1 448	0.03	0.09	0.3	2.9	93.7%	93.9%
Streptococcus	9 351	0.27	4.1	1.0	3.0	88.6%	93.7%
E.coli 2	35 501	1.1	60	3.1	7.1	82.8%	69.4%

problem are not fully satisfactory. However, it should be possible to adapt the visualization techniques employed in the *REPuter*-program (Kurtz *et al.*, 2001) to obtain an alignment browsing system, which gives a good overview of an entire *pairwise* alignment of large sequences and also allows zooming in and out on regions of interest. Another research topic to be addressed is the evaluation of *MGA* from a biological point of view. However, we have to postpone measurements of the quality of the alignments delivered by *MGA* until manually curated datasets and protocols for such an evaluation will become available.

ACKNOWLEDGEMENTS

S.K. was partially supported by grant KU 1257/1-2 from the Deutsche Forschungsgemeinschaft. Robert Giegerich, Mohamed Ibrahim Abouelhoda, and Gordon Gremme carefully read previous versions of the manuscript. Jörn Clausen and Rainer Orth were very helpful when performing the experiments. All this help is appreciated.

REFERENCES

Altschul,S., Madden,T., Schäffer,A., Zhang,J., Zhang,Z., Miller,W. and Lipman,D. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Arkin,E. and Silverberg,E. (1987) Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, **18**, 1–8.

Batzoglu,S., Pachter,L., Mesirov,J., Berger,B. and Lander,E. (2000) Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Res.*, **10**, 950–958.

Buhler,J. (2001) Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, **17**, 419–428.

Cormen,T., Leiserson,C. and Rivest,R. (1990) *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Delcher,A., Kasif,S., Fleischmann,R., Peterson,J., White,O. and Salzberg,S. (1999) Alignment of whole genomes. *Nucleic Acids Res.*, **27**, 2369–2376.

Durbin,R., Eddy,S., Krogh,A. and Mitchison,G. (1998) *Biological Sequence Analysis*. Cambridge University Press.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

Jacobson,G. and Vo,K. (1992) Heaviest increasing/common subsequence problems. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, Tucson, Arizona, April/May 1992*, Lecture Notes in Computer Science, 644, Springer, pp. 52–66.

Joseph,D., Meidanis,J. and Tiwari,P. (1992) Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory, Helsinki 1992*, Lecture Notes in Computer Science, 621, Springer, pp. 326–337.

Kasai,T., Lee,G., Arimura,H., Arikawa,S. and Park,K. (2001) Linear-Time Longest-Common-Prefix Computation in Suffix Ar-

Table 3. Application of *MGA* to the Multiple Genome Sets. The third column shows the length thresholds for the *multiMEM*s in the recursive applications of *MGA*. The times reported are user times plus system times in minutes. The fourth column shows the total running time including ClustalW, while the fifth column gives the running times excluding ClustalW, i.e., the time for computing the *multiMEM*s and chaining them. The last three columns show the coverage for the sequences, i.e., the percentage of bases, appearing in the delivered alignment

sequence set	total length	ℓ	total time (minutes)	Phases (1) + (2) (minutes)	Space (megabytes)	Coverage		
						min	avg	max
Adenovirus 6	211 832	10; 9; 8; 7; 6; 5; 4	2:39	1:06	46	83.1%	85.1%	86.8%
E. coli 3	15 666 116	1000; 20	30:29	3:17	151	70.0%	74.2%	83.7%
C. pneumoniae 3	3 686 648	15; 9; 4	93:59	61:16	110	72.7%	80.1%	83.9%
S. aureus 3	8 594 300	1000; 20	22:14	1:47	75	91.4%	92.6%	94.3%
S. aureus 4	11 416 205	1000; 25	27:45	1:17	116	87.4%	88.9%	90.2%

rays and its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, Jerusalem, Israel, July 2001*, Lecture Notes in Computer Science, 2089, Springer, pp. 181–192.

Kent,W. and Zahler,A. (2000) Conservation, regulation, synteny, and introns in large-scale *C. briggsae–C. elegans* genomic alignment. *Genome Res.*, **10**, 1115–1125.

Kurtz,S., Choudhuri,J., Ohlebusch,E., Schleiermacher,C., Stoye,J. and Giegerich,R. (2001) REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res.*, **29**, 4633–4642.

Lawler,E. (1976) *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.

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

McCreight,E. (1976) A space-economical suffix tree construction algorithm. *J. ACM*, **23**, 262–272.

Miller,W. (2001) Comparison of genomic DNA sequences: solved and unsolved problems. *Bioinformatics*, **17**, 391–397.

Myers,E. and Miller,W. (1995) Chaining multiple-alignment fragments in sub-quadratic time. In *Proceedings of the Sixth ACM-SIAM Symposium on Discrete Algorithms*. San Francisco, pp. 38–47.

Needleman,S. and Wunsch,C. (1970) A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Thompson,J., Higgins,D. and Gibson,T. (1994) *CLUSTAL W*: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, **22**, 4673–4680.

Ukkonen,E. (1985) Algorithms for approximate string matching. *Information and Control*, **64**, 100–118.

Vingron,M. and Argos,P. (1989) A fast and sensitive multiple alignment algorithm. *Comput. Appl. Biosci.*, **5**, 115–121.

Weiner,P. (1973) Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*. The University of Iowa, pp. 1–11.