An iterative method for faster sum-of-pairs multiple sequence alignment

Knut Reinert[1], Jens Stoye[2,*] and Torsten Will[3,4]

[1]Celera Genomics, Informatics Research, 45 West Gude Drive, Rockville, MD 20850, USA, [2]German Cancer Research Center (DKFZ), Theoretical Bioinformatics (H0300), Im Neuenheimer Feld 280, 69120, Heidelberg, Germany and [3]Research Center for Interdisciplinary Studies on Structure Formation (FSPM), University of Bielefeld, Postfach 100131, 33501, Bielefeld, Germany

Received on December 10, 1999; revised on March 13, 2000; accepted on May 5, 2000

Abstract

Motivation: *Multiple sequence alignment is an important tool in computational biology. In order to solve the task of computing multiple alignments in affordable time, the most commonly used multiple alignment methods have to use heuristics. Nevertheless, the computation of* optimal *multiple alignments is important in its own right, and it provides a means of evaluating heuristic approaches or serves as a subprocedure of heuristic alignment methods.*

Results: *We present an algorithm that uses the divide-and-conquer alignment approach together with recent results on search space reduction to speed up the computation of multiple sequence alignments. The method is adaptive in that depending on the time one wants to spend on the alignment, a better, up to optimal alignment can be obtained. To speed up the computation in the optimal alignment step, we apply the $\mathcal{A}^*$ algorithm which leads to a procedure provably more efficient than previous exact algorithms. We also describe our implementation of the algorithm and present results showing the effectiveness and limitations of the procedure.*

Availability: *http://bibiserv.techfak.uni-bielefeld.de/oma/*

Contact: *j.stoye@dkfz.de*

Introduction

Multiple sequence alignment is an important tool in computational biology. Application areas include sequence assembly, molecular modeling, protein structure–function analysis, phylogenetic studies, database search, and primer design. Depending on the application, a cost function is defined that assigns a numerical value to each possible alignment, and the hope is that the lowest scoring alignments reveal important information about the specific problem. One widely used framework for such a cost function is the (*weighted*) *sum of pairs* ((W)SP) score with quasi-natural gap costs (Altschul, 1989; Gupta *et al.*, 1995; Kececioglu and Zhang, 1998). Since the problem of computing optimal multiple alignments according to the SP score is NP complete (Wang and Jiang, 1994), usually in practice heuristic (tree-based) methods are used. Nevertheless, the computation of optimal multiple alignments has its justification as a means of evaluating heuristic approaches or as a subprocedure of heuristic alignment methods like the recursive Divide-and-Conquer Alignment algorithm (DCA; Tönges *et al.* (1996); Stoye *et al.* (1997); Stoye (1998)). In fact we will show how to combine an efficient procedure for computing optimal multiple alignments with this method such that the resulting procedure produces increasingly better alignments that converge to an optimal one.

Formally, a global alignment of $K > 2$ sequences $S_1, \ldots, S_K$ over an alphabet Σ is a $K \times M$ matrix $A = (a_{i,j})$, $a_{i,j} \in \Sigma \cup \{-\}$, such that ignoring the blank characters $-$, the ith row reproduces sequence S_i, and there is no column consisting only of blanks. A maximal run of adjacent blank characters in a row is called a *gap*. By $A_{i_1,i_2,\ldots,i_n}$ we denote the projection of A to the sequences $S_{i_1}, S_{i_2}, \ldots, S_{i_n}$.

For a pairwise projection of A to S_k and $S_{k'}$, let $c(A_{k,k'})$ be the cost of this projection which is usually defined as the sum over all substitution costs weighted by a substitution score matrix, plus a penalty for each gap. Then the overall cost $c(A) := \sum_{k<k'} c(A_{k,k'})$ is called the *sum of pairs* (SP) cost of the alignment A. If, in addition, we assign to each element of this sum a weight $w_{k,k'}$ then the cost is called the *weighted sum of pairs* (WSP) cost. To simplify the discussion, however, we concentrate on the SP cost function. Nevertheless, all results derived in this paper hold for the WSP cost function as well.

The SP multiple alignment problem is defined as follows. Given sequences $S_1, \ldots, S_K$, find an alignment

*To whom correspondence should be addressed.

[4]Present address: mediaWays GmbH, Hülshorstweg 30, 33415 Verl, Germany.

```
S1 : X - - - X
S2 : X X - X X
S3 : X X X X X
```

Fig. 1. Illustration of natural and quasi-natural gap counts. The 'natural' number of gaps summed over each pair of sequences is three (one gap in each pair) while the 'quasi-natural' number is four (between the first and second sequence two gaps are counted).

A that minimizes $c(A)$. We will denote such an optimal alignment by A^{opt}.

The gap penalty implied by the pairwise projection cost $c(A_{k,k'})$ usually depends on the length of the gap. For pairwise alignments, the class of affine gap cost functions with a relatively high *gap initiation cost* (*gapinit*) for the first blank character and a smaller *gap extension cost* (*gapext*) for each additional blank character is widely acknowledged to yield good results. Although it is also possible to use affine gap costs for multiple alignments, Altschul (1989) pointed out that this is impractical for even a modest number of sequences. He proposed instead a simpler gap cost function, called *quasi-natural gap costs*. This function miscounts the true number of gaps in a multiple alignment by only a small amount and hence is regarded a good approximation of the affine gap cost measure. As an example, see Figure 1 for an alignment where the 'natural' number of gaps summed over each pair of sequences is three (one gap in each pair) while the 'quasi-natural' number is four (between the first and second sequence two gaps are counted).

Like most alignment problems, the SP multiple alignment problem with quasi-natural gap costs can be solved by dynamic programming which yields an algorithm with time complexity $O(2^{(2K)}N)$ and space complexity $O(N)$, where $N = \prod_i N_i$ with N_i being the length of sequence S_i. This is feasible only for very small problem instances. Gupta *et al.* (1995) presented a branch-and-bound algorithm whose implementation—called MSA in what follows—can optimally align some examples of six sequences of length 250 in a few minutes. Larger examples, however, require excessive space.

In this paper we apply the so-called $\mathcal{A}^*$ algorithm to multiple alignment. Similar to the algorithm used in MSA, it computes a shortest path in the dynamic programming graph, but with redefined edge weights, which reduces the search space considerably (see Horton (1997); Shibuya and Imai (1997); Lermen and Reinert (to appear)). Our two main contributions are (1) to provide an efficient implementation of the $\mathcal{A}^*$ algorithm for exactly solving the SP alignment problem with quasi-natural gap costs, which can be shown to be superior to the Carrillo–Lipman bounding (Carrillo and Lipman, 1988) applied in MSA, and (2) to combine this algorithm with the DCA approach in order to develop an iterative procedure for computing multiple alignments with a nice time-versus-quality tradeoff. To our knowledge this novel algorithm is the first to improve its result up to optimality—given enough resources—while being able to quickly yield good intermediate results.

The $\mathcal{A}^*$ algorithm in multiple alignment

An alignment of the K sequences can be interpreted as a path in a K-dimensional grid graph $G = (V, E)$ with a source s and a sink t. In addition we add a dummy node d and an edge from d to s. A node $v = (v_1, \ldots, v_K)$ in the grid naturally divides each sequence S_k in a prefix $\alpha_k^v := S_k[1..v_k]$ and a suffix $\sigma_k^v := S_k[(v_k + 1)..N_k]$ where $S[i..j]$ denotes the substring of S starting with the ith and ending with the jth character of S. Each path starting in d and ending with an edge $e = (u, v)$ corresponds to one possible alignment of the prefixes $\alpha_1^v, \ldots, \alpha_K^v$. (The dummy node and edge ensure there is a path corresponding to the empty prefixes of all sequences.) Similarly, each path starting with an edge $e = (u, v)$ and ending with an edge $f = (p, t)$ corresponds to an alignment of the suffixes $\sigma_1^v, \ldots, \sigma_K^v$.

The cost of an edge is the SP cost of the alignment column corresponding to the edge. Let us denote the set of all paths starting with an edge e and ending with an edge f by $e \to f$. We denote the shortest path in $e \to f$ by $e \to^* f$ and its cost by $c(e \to^* f)$.

Let $e = (u, v)$ and $f = (v, w)$ be two adjacent edges. Then the cost of f *preceded by e* is defined as

$$c(f|e) := c(d \to^* e) + c(f) + \textit{gapcost}(e,\ f),$$

and the cost of the shortest path ending with f is the minimum of $c(f|e)$ over all edges e incident to f. In the setting of quasi-natural gap costs that we consider, we have

$$\begin{aligned}&\textit{gapcost}(e,\ f)\\ &:= \sum_{k<k'} \begin{cases} 0 & \text{if } (w_k = v_k \text{ and } w_{k'} = v_{k'}) \\ & \text{or } (w_k = v_k + 1 \text{ and } w_{k'} = v_{k'} + 1) \\ \textit{gapext} & \text{if } (w_k = v_k + 1 = u_k + 2 \text{ and } w_{k'} = v_{k'} = u_{k'}) \\ & \text{or } (w_k = v_k = u_k \text{ and } w_{k'} = v_{k'} + 1 = u_{k'} + 2) \\ \textit{gapinit} & \text{in all other cases} \end{cases}\end{aligned}$$

where $u = (u_1, \ldots, u_K)$, $v = (v_1, \ldots, v_K)$, and $w = (w_1, \ldots, w_K)$.

Of course it is not feasible to compute a shortest path in the full grid graph whose size is $O(N)$, where, as above, $N = \prod_i N_i$. Gupta *et al.* (1995) applied Dijkstra's algorithm together with a bounding procedure that reduces the number of edges that have to be visited. The algorithm uses a priority queue Q in which new edges are only inserted if potentially an optimal path can pass through them. Given an edge $f = (v, w)$ adjacent to the current edge $e = (u, v)$, this can be determined by using an upper bound U on the cost of an optimal alignment (obtained by a heuristic alignment) and a lower bound

$$L(w) := \sum_{k<k'} c(A^{\mathrm{opt}}(\sigma_k^w, \sigma_{k'}^w)) \tag{1}$$

on the cost of an optimal alignment of the suffixes that are induced by w. The edge $f = (v, w)$ is only inserted into Q if $c(f|e) + L(w) \leq U$. That means, if the sum of the cost of the optimal path starting with d and ending with e plus the cost of f, the gap penalty, and the lower bound $L(w)$, is already greater than an upper bound U, then no optimal alignment can go through f. Additionally, their approach employs the so-called Carrillo–Lipman bounding (Carrillo and Lipman, 1988) which further excludes edges from consideration based on the following theorem:

THEOREM 1 (Carrillo, Lipman). *Let A^{opt} be an optimal alignment of the K strings $S_1, \ldots, S_K$, $L := L(s)$ the lower bound defined in Equation (1) and $U = c(A^{\mathrm{heur}})$ an upper bound for $c(A^{\mathrm{opt}})$. Then the following inequality holds for every projection on a pair S_i, S_j of sequences:*

$$c(A_{i,j}^{\mathrm{opt}}) \leq c(A^{\mathrm{opt}}(S_i, S_j)) + U - L.$$

The $\mathcal{A}^*$ algorithm employs basically the same bounding procedure as in Gupta *et al.* (1995), however with redefined edge costs. Thereby it speeds up computations by directing the search of a shortest path more towards the sink node t. (Therefore this technique is also called *Goal Directed Unidirectional Search* (GDUS) (Lengauer, 1990).) It redefines the costs of all edges $e = (u, v) \in E$ as follows: $c'(e) := c(e) - l(u) + l(v)$, where $l(u)$ is a lower bound for the cost of a shortest path starting with some edge adjacent to node u and ending with an edge incident to t. If $l()$ fulfills the *consistency* condition $c(e) + l(v) \geq l(u)$, $\forall e = (u, v) \in E$, then it is easy to show that the redefinition of the edge costs does not change the optimal path and the edge costs are still positive, so Dijkstra's algorithm with the simple bounding procedure can be used as before. We can choose $l(u) := L(u)$, because L fulfills the consistency condition. Generally, the better the lower bound l is in the redefinition of the edge costs, the better the GDUS works.

It is worthwile noting that it can be shown (Horton, 1997; Lermen and Reinert, to appear) that the above redefinition of edge costs implies the Carrillo–Lipman bound from Theorem 1 using L as lower bound.

We used this result to implement an exact multiple sequence alignment algorithm that provably explores at most as many nodes as any algorithm using Carrillo–Lipman bounding. In the next section we describe how we make use of this algorithm in our newly proposed heuristics.

Iterative improvement using DCA

As described above, the $\mathcal{A}^*$ algorithm uses an upper bound U for the alignment cost to speed up the computation of an optimal alignment. For the computation of the upper bound we use the Divide-and-Conquer Alignment algorithm (DCA; Tönges *et al.* (1996); Stoye *et al.* (1997), Stoye (1998)). We first describe the basic DCA algorithm, and then we describe how the interchangeable use of DCA and the optimal alignment procedure applied to parts of the sequences can be used to successively improve an initial heuristic alignment, up to optimality. Alternatively, the algorithm can be stopped after a predetermined time, yielding a heuristic alignment which is provably nearer to the optimum the more time was spent.

The basic DCA method

The DCA method allows to quickly compute heuristic multiple sequence alignments. In contrast to other, tree-based, multiple alignment heuristics, DCA aims at optimizing the SP alignment score with quasi-natural gap costs, which makes it a logical choice to use in combination with the $\mathcal{A}^*$-based optimal alignment algorithm.

A sketch of the DCA method is shown in Figure 2. The sequences (denoted by the horizontal bars) are cut at certain positions near to their center, in the sequel called *cut positions* (denoted by the small vertical tics). This divides the problem of aligning K (long) sequences into the two problems of aligning the (shorter) K prefix and K suffix sequences. Assuming that it is possible to compute optimal alignments of these two sets of shorter sequences, an alignment of the complete sequences is obtained by just concatenating the prefix alignment and the suffix alignment. On the other hand, if the prefix resp. suffix sequences are still too long to be aligned optimally, the procedure is applied recursively until the sequences are of a length short enough to be tractable for the exact alignment procedure. To this end, DCA has a parameter Z, the *stop length*, such that the recursion stops if the maximal length of a sequence in a block drops below Z.

Certainly the choice of the cut positions is critical for the success of the DCA procedure, and inadequate cut positions in an early division step can deteriorate the whole alignment. It has been shown that the heuristics of *minimal additional costs* yields very good, in many cases optimal cut positions. Minimal additional cost cut positions are defined by means of forward/backward matrices (which are well known from the study of local sequence similarities and suboptimal alignments) for all pairs of sequences. These matrices allow for any pair of cut positions to quickly assess the overhead of cutting and combining the best possible alignment of the resulting prefixes and suffixes, as opposed to an optimal alignment of the complete (un-cut) sequences, called the additional

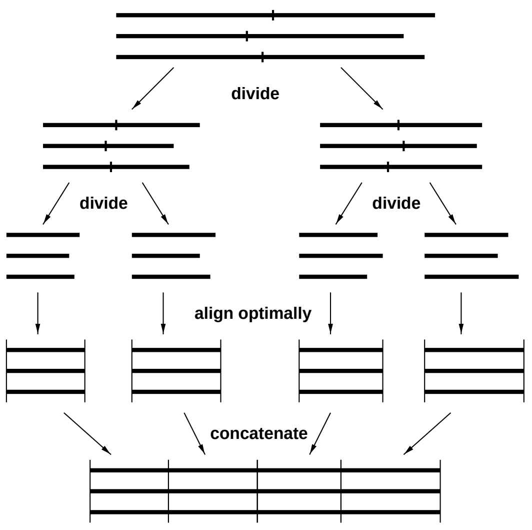

Fig. 2. The basic Divide-and-Conquer Alignment method.

cost imposed by the cut positions. The K cut positions are then selected, so that the sum of all pairwise additional costs is minimized. For more details on the definition and computation of cut positions, a number of variations, and efficient speed-up techniques, see (Stoye, 1998) and references therein.

Iterative improvement of the upper bound

DCA called with a small value of Z allows to quickly compute an upper bound U for the $\mathcal{A}^*$-based optimal alignment procedure. However, not only the $\mathcal{A}^*$ algorithm can use DCA, but both programs can benefit from each other: DCA adheres a time versus quality tradeoff; the larger one chooses the parameter Z, the (provably) better is the alignment one gets, while the computation time increases due to the larger optimal alignments to be computed. This motivates an iterative combination of both DCA and the optimal alignment procedure: Successively, we call DCA with increasing values of Z where, at each step, we can use the values of the corresponding partial alignments from the previous step to compute an upper bound for the computation of an optimal alignment using the $\mathcal{A}^*$ algorithm. Moreover, we can stop at any point of this procedure and have a heuristic alignment. The longer we wait, the better is the alignment—up to optimal. To our knowledge this is the first iterative alignment algorithm that provably converges to an optimal alignment.

Note that for iteratively computing better DCA alignments with larger Z-values one only has to compute the cut positions once for the smallest values of Z. Larger Z-values are obtained by 'ignoring' intermediate cut positions. However, one has to be careful when this way 'fusing' two short alignments, because with quasi-natural gap costs the alignment score is not additive. The problem becomes apparent if one focuses on a gap that runs through a cut point. In the final alignment, of course, the gap initiation cost will be imposed only once on this gap. During the construction, however, when the parts on the left and right hand side of the cut position are aligned independently, it is not clear, if such a gap will be continued on the other side of the cut position. Hence, in order to use a valid upper bound, the gap initiation cost must be added on either side of the cut. This way, the sum of the costs for the small alignments will be higher than the cost of the alignment obtained by concatenating all small alignments.

Implementation

We have implemented the algorithms described in the previous sections as a C++ library of classes for the alignment of sequences called *OMA* (which is short for *Optimal Multiple Alignment*), built upon the *Library of Efficient Data stuctures and Algorithms* (LEDA; Mehlhorn and Näher (1999)). Note that using the library approach and the use of C++ classes, one has to expect a fairly large constant factor in running time and memory usage. Nevertheless, our main emphasis was to create an open library that easily can be modified and/or extended. We have compiled an executable called *oma* which is freely available from the address http://bibiserv.techfak.uni-bielefeld.de/oma/. It is the basis for the computations of the Results section.

To speed up the computation or to improve the result, we have included in *oma* some additional techniques from the literature which we briefly review next. Most of these features are optional and can be switched off by the user.

Face bounding If one does not insist on a provably optimal alignment, one can use a simple heuristic to speed up the procedure considerably. This so-called *face bounding* (Gupta *et al.*, 1995; Lermen and Reinert, to appear) is employed by defining for each pair of sequences (k, k') a non-negative constant $EPS_{k,k'}$, and then making the (not always true) assumption that any alignment cannot be optimal that passes a node v in the grid graph with $c(A^{\mathrm{opt}}(\alpha_k^v, \alpha_{k'}^v)) + c(A^{\mathrm{opt}}(\sigma_k^v, \sigma_{k'}^v)) > c(A^{\mathrm{opt}}(S_k, S_{k'})) + EPS_{k,k'}$. This way the exploration of the grid graph is artificially bounded to a banded region around the best paths, similar to two-dimensional banded alignments (Myers, 1986). The impact of this effect can be controlled by a parameter, and it can be switched off completely so that an optimal alignment is guaranteed, at the expense of considerably higher resource requirements.

Gray code enumeration Both Gupta *et al.* (1995) and Lermen and Reinert (to appear) point out that the iteration over all outgoing edges of a node can be efficiently perfomed by enumerating the neighbours of a node in

Gray code succession. The $2^K - 1$ edges outgoing from a given node are numbered in binary, using K bits. When evaluating the node, the Gray code enumeration guarantees that from one outgoing edge to the other, only one bit in the binary representation changes and the overall enumeration takes $O(K)$ time. This allows an efficient computation in many internal loops.

Re-alignment at cut positions It has been observed (Stoye *et al.*, 1997) that alignments created by DCA sometimes contain obvious errors in the neighbourhood of cut positions. To correct for these errors, DCA allows the user to specify a *window size* $W \geq 0$. After the final step of the divide-and-conquer procedure, the concatenation of the short alignments, a window of size W is placed across each cut position, and inside this window the sequences are re-aligned optimally. Re-alignment usually leads to small local improvements of the alignment. This feature is included in *oma* as well, but, of course, it makes sense only for iterations where Z is larger than the window size.

Sequence weighting To avoid overweighting redundant information that can arise, e.g. from some identical or highly similar sequences in the sequence set, *oma*, like DCA, aims at optimizing a *weighted* sum-of-pairs score of the form $c(A) := \sum_{k<k'} w_{k,k'} c(A_{k,k'})$. The weight factors $w_{k,k'}$ are computed from the pairwise distances between the sequences. Higher weights are given to the more similar pairs, as having them aligned optimally should be more important than aligning two fairly unrelated sequences optimally at the expense of worsening a good alignment of closely related sequences. The strength of weighting can be controlled by a parameter λ, the *weight intensity*, which can be adjusted by the user to any value between $\lambda = 0$ (no weighting) and $\lambda = 1$ (maximum weighting). Alternatively, arbitrary user-defined weights can be provided.

Parallelization We have considered parallelizing the algorithms included in *OMA*. Obviously, the subproblems obtained after cutting the sequences can be handled independently, and hence can be computed in parallel on different processors with distributed memory. During the program development we have kept this in mind, and we have compiled a multi-threaded version of *oma*. However, this does not help in the final optimal alignment step of the complete sequences. Introducing parallelization here would mean to parallelize the priority queue using e.g. methods described in Brodal (1999), which needed shared memory. This, however, is out of the scope of the current implementation.

Results

We have run *oma* on a number of alignment problems from the Benchmark Alignments Database (BAliBASE; Thompson *et al.* (1999a)). All runs were performed on a Sun Ultra Enterprise 450 with 400 MHz processors. Jobs were limited to 2 GB of memory and 12 h of CPU time. We used a distance version of Dayhoff's PAM 250 matrix with quasi-natural gap costs as the alignment cost function, sequence weighting was switched off and end-gaps were penalized like internal gaps. The general result is that we can align a typical set of 4 to 6 protein sequences to optimality within 10 s up to a few minutes. Some more difficult examples, however, require excessive computation time and memory. If we stop the computation after 1 min, we get a (sub-optimal) alignment in all test cases from the *reference1* subset of BAliBASE. A few detailed results follow.

Figure 3 shows for increasing values of Z the behaviour of *oma* on the test set 1cpt from *reference1* of BAliBASE, containing four cytochrome p450 sequences. The sequence lengths range from 378 to 434 amino acids, and the average sequence identity is 20%. One can see the monotonically decreasing alignment cost, and how the cost of the heuristic alignment upper-bounds the score of the *oma* alignment. As noted above, we have two upper bound values: the lower one, which is just the alignment cost of the previous iteration, and the larger one, which is the lower one increased by multiple end gap costs whenever a gap runs through a cut point. A rather close-to-optimal alignment is obtained already after a few iterations. However, the many very short alignments computed in the beginning take longer time than the fewer (but still relatively short) alignments around $Z = 32$. Even though the upper bound is already very close to the optimal alignment score, the last step ($Z = 512$) which yields the optimal alignment takes by far the longest time to compute.

For comparison reasons, we have also run this example without the $\mathcal{A}^*$ strategy. Here the computation for $Z = 512$ could not be performed within the 2 GB of available memory. The number of edges explored in the search phase of the last alignment step which could be run, for $Z = 256$, increases from 1.4×10^6 (with $\mathcal{A}^*$) to 2.2×10^6 (without $\mathcal{A}^*$). The computation time increases from 180 s to 549 s.

Table 1 shows some more results on short (top), medium length (middle), and long (bottom) sequences. Each block is divided in distantly related, closer, and closely related sequences (see the column *avg.id.*). The alignment cost and, in parentheses, the running time and memory usage of *oma* is shown at two different Z-values. The first Z-value (called $Z_{\max}$) was chosen such that the program did not use more than 2 GB of memory and did not run longer than 12 h, whereas the second Z-value is half

Table 1. SP alignment cost, computation time and memory usage of *oma*, MSA and DCA for selected test sets from *reference1* of BAliBASE. The best score in each line is printed in bold face

Test set	K	Length	avg.id.	$Z_{\max}$	*oma* at $Z_{\max}$		*oma* at $Z_{\max}/2$		MSA		DCA	
1ubi	4	76–94	18	128	**8631**	(37.9 s, 21 MB)	**8631**	(6.0 s, 2 MB)	8639	(0.6 s, 3 MB)	8685	(0.4 s, 3 MB)
1wit	5	89–106	17	128	**16517**	(2.0 min, 36 MB)	16523	(4.0 s, 2 MB)	16533	(2.1 s, 3 MB)	16610	(0.4 s, 3 MB)
3cyr	4	95–109	31	128	**9888**	(4.3 s, 2 MB)	**9888**	(4.6 s, 2 MB)	**9888**	(0.4 s, 3 MB)	**9888**	(0.3 s, 3 MB)
1pfc	5	108–117	28	128	**17708**	(20.0 s, 2 MB)	**17708**	(5.7 s, 2 MB)	17710	(1.0 s, 3 MB)	17771	(0.4 s, 3 MB)
1fmb	4	98–104	49	128	**8804**	(3.3 s, 2 MB)	**8804**	(3.0 s, 2 MB)	**8804**	(0.2 s, 2 MB)	**8804**	(0.5 s, 2 MB)
1fkj	5	98–110	44	128	**15809**	(4.3 s, 2 MB)	15815	(3.6 s, 2 MB)	15815	(0.3 s, 3 MB)	15815	(0.6 s, 2 MB)
3grs	4	201–237	14	128	**23478**	(22.0 s, 2 MB)	23491	(9.9 s, 2 MB)	23489	(2.2 min, 14 MB)	23590	(0.8 s, 2 MB)
1sbp	5	224–263	19	128	**43115**	(62.9 min, 668 MB)	43188	(2.1 min, 70 MB)	–	(> 12 h)	43581	(1.1 s, 2 MB)
1ad2	4	203–213	30	256	**19714**	(14.7 s, 2 MB)	**19714**	(9.6 s, 2 MB)	19726	(0.6 s, 3 MB)	19716	(0.5 s, 3 MB)
2cba	5	237–259	26	128	**40281**	(15.4 min, 183 MB)	40295	(17.9 s, 2 MB)	**40281**	(63.0 min, 69 MB)	40496	(0.9 s, 3 MB)
1zin	4	206–216	42	256	**19110**	(8.0 s, 2 MB)	**19110**	(6.8 s, 2 MB)	**19110**	(0.5 s, 2 MB)	**19110**	(0.7 s, 2 MB)
1amk	5	242–254	49	128	**36659**	(11.0 s, 2 MB)	**36659**	(10.8 s, 2 MB)	**36659**	(0.9 s, 2 MB)	**36659**	(0.9 s, 2 MB)
2myr	4	340–474	16	128	**43541**	(2.1 min, 53 MB)	43629	(21.8 s, 2 MB)	–	(> 12 h)	43834	(1.6 s, 2 MB)
1pamA	5	435–572	18	64	**86357**	(7.6 min, 62 MB)	86482	(27.2 s, 2 MB)	–	(> 2 GB)	86923	(2.7 s, 2 MB)
1ac5	4	421–483	29	128	43341	(34.5 s, 16 MB)	43380	(18.3 s, 2 MB)	**43325**	(22.2 min, 32 MB)	43513	(1.5 s, 2 MB)
2ack	5	452–482	28	128	**77139**	(24.9 min, 234 MB)	77161	(67.5 s, 31 MB)	–	(> 12 h)	77422	(2.2 s, 2 MB)
1ad3	4	424–447	47	256	39218	(15.8 s, 2 MB)	39218	(13.2 s, 2 MB)	**39209**	(1.7 s, 2 MB)	39225	(1.0 s, 2 MB)
1rthA	5	526–541	42	256	**80352**	(36.3 s, 15 MB)	**80352**	(26.8 s, 2 MB)	80358	(4.7 s, 2 MB)	80449	(2.1 s, 2 MB)

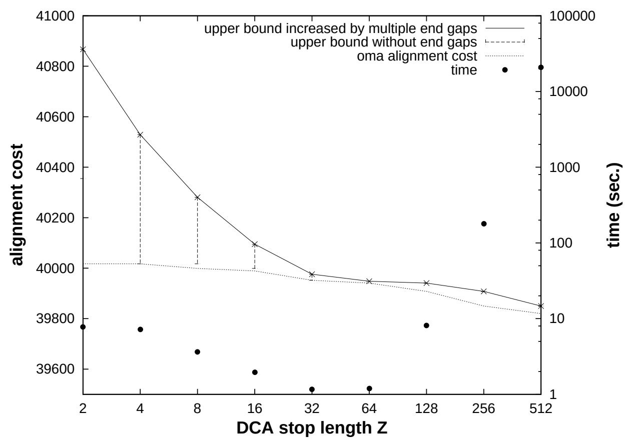

Fig. 3. The successive improvement of the alignment cost.

of the first one. This shows the tremendous decrease in runnning time by only going back one step in the iteration. For comparison, we have computed alignments with the programs MSA and DCA. These results are shown in Table 1 as well. Only in few cases the *oma* alignment has a higher cost than the MSA alignment, while in several cases MSA computes worse alignments or is unable to compute any alignment with the given resources. DCA usually is the fastest method, but most of the alignments are worse than those computed by *oma* or MSA. For the complete results on all test sets from *reference1* of BAliBASE, see http://bibiserv.techfak.uni-bielefeld.de/oma/.

Note that the running time and memory usage of *oma* not only depends on the number and length of the input sequences but—like for many multiple alignment programs—it also depends on the similarity of the sequences (although there are counter examples). This is due to a more effective reduction of the search space in branch-and-bound type algorithms for easy problem instances.

We have also investigated the biological quality of the (sub-) optimal alignments computed by *oma*. To this end we have used the test program *bali_score* that comes with *BAliBASE*. The authors of BAliBASE have defined core blocks of their alignments, and *bali_score* computes a percentage of correctly aligned residue pairs within these core regions (see Thompson *et al.* (1999b) for a detailed description of the evaluation procedure). Averaged over all test sets from *reference1* of BAliBASE, we obtain the following results: For the alignments of the group with $<$25% average identity, *oma* correctly aligns 60% of the residues in the core blocks. In the group between 20 and 40% identity, 92% of the residues in the core blocks are successfully aligned, and in the group above 35%, 94% of the residues in core blocks are aligned correctly. These values are almost exactly the values achieved by the best performing alignment programs in the study of Thompson *et al.* (1999b).

In a final experiment, we have investigated the maximal number of sequences that we can align optimally with our method. Therefore,we have selected a large family

Table 2. Computation time and memory usage of *oma* for different numbers K of cytochrome C sequences

K	Time [s]	Memory [MB]
1	2.0	6.7
2	2.3	6.7
3	2.8	6.7
4	3.7	7.7
5	5.2	8.8
6	5.3	9.1
7	7.1	10.5
8	9.2	12.2
9	15.2	14.6
10	18.4	19.1
11	79.5	31.4
12	197.8	45.8
13	883.0	84.2
14	4305.5	149.5
15	22197.6	298.8
16	72856.9	1051.1

of very similar protein sequences, cytochrome C. Due to the high similarity of cytochrome C, the correct multiple alignment is not a great challenge. It is easily obtained by hand, and most automatic methods produce quite successfully almost the same alignment. However, for theoretical reasons we wanted to find out where *oma* reaches its limits. The results are presented in Table 2. One observes rather moderate resource requirements for up to 12 sequences. Above this value, the exponential increase becomes very apparent. Note that these values only hold for the very similar sequences used in this experiment. In a realistic setting, one has to expect much higher resource requirements, see the results presented above.

Conclusion

We have presented a new iterative alignment algorithm that combines an improved algorithm for the optimal alignment of multiple biological sequences based on the $\mathcal{A}^*$ algorithm with the recursive Divide-and-Conquer Alignment method. Although still too expensive for larger test sets, we believe that with this approach we are close to how far one can get with simultaneous, optimal multiple sequence alignment.

Acknowledgements

We would like to thank Andreas Dress and Robert Giegerich for helpful conversations as well as for continuous support of this project. We would also like to thank the Max-Planck-Institut (MPI) für Informatik in Saarbrücken for generously making their computer facilities available to us.

References

Altschul,S.F. (1989) Gap costs for multiple sequence alignment. *J. Theor. Biol.*, **138**, 297–309.

Brodal,G.S. (1999) Priority queues on parallel machines. *Parallel Comput.*, **25**(8), 987–1011.

Carrillo,H. and Lipman,D. (1988) The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, **48**(5), 1073–1082.

Gupta,S.K., Kececioglu,J.D. and Schäffer,A.A. (1995) Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Comp. Biol.*, **2**(3), 459–472.

Horton,P. (1997) String Algorithms and Machine Learning Applications for Computational Biology, *PhD dissertation*, University of California, Berkeley, CA.

Kececioglu,J.D. and Zhang,W. (1998) Aligning alignments. In Farach,M. (ed.), *Proceedings of CPM 1998* Lecture Notes in Computer Science 1448, Springer Verlag, Berlin, pp. 189–208.

Lengauer,T. (1990) *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley-Teubner, Chichester.

Lermen,M. and Reinert,K. (to appear) The practical use of the $\mathcal{A}^*$ algorithm for exact multiple sequence alignment. *J. Comp. Biol.*, Accepted for publication. (See also *Technical Report 97-1-028*, MPI fürInformatik, Saarbrücken, Germany, 1997.)

Mehlhorn,K. and Näher,S. (1999) *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK.

Myers,E.W. (1986) An $O(ND)$ difference algorithm and its variations. *Algorithmica*, **1**, 251–266.

Shibuya,T. and Imai,H. (1997) New flexible approaches for multiple sequence alignment. *J. Comp. Biol.*, **4**(3), 385–413.

Stoye,J. (1998) Multiple sequence alignment with the divide-and-conquer method. *Gene*, **211**, GC45–GC56.

Stoye,J., Moulton,V. and Dress,A.W.M. (1997) DCA: an efficient implementation of the divide-and-conquer approach to simultaneous multiple sequence alignment. *CABIOS*, **13**(6), 625–626.

Thompson,J.D., Plewniak,F. and Poch,O. (1999a) BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, **15**(1), 87–88. See http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE/.

Thompson,J.D., Plewniak,F. and Poch,O. (1999b) A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Res.*, **27**(13), 2682–2690.

Tönges,U., Perrey,S.W., Stoye,J. and Dress,A.W.M. (1996) A general method for fast multiple sequence alignment. *Gene*, **172**, GC33–GC41.

Wang,L. and Jiang,T. (1994) On the complexity of multiple sequence alignment. *J. Comp. Biol.*, **1**(4), 337–348.