Combining Divide-and-Conquer, the $\mathcal{A}^*$-Algorithm, and Successive Realignment Approaches to Speed up Multiple Sequence Alignment

— Extended Abstract —

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

Abstract

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 object oriented implementation of the algorithm and present results showing the effectiveness of the procedure.

1 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 [1, 3, 5]. Since the problem of computing optimal multiple alignments according to the SP score is NP complete [13], 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) [10, 12]. In fact we will show how to combine an efficient procedure for computing optimal multiple alignments with this method such that the

[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

[3]Universität Bielefeld, Research Center for Interdisciplinary Studies on Structure Formation (FSPM), Postfach 100131, 33501 Bielefeld, Germany

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

[*]Corresponding author. E-mail: j.stoye@dkfz-heidelberg.de, phone +49-6221-422719, fax +49-6221-422849

resulting procedure produces increasingly better alignments that converge to an optimal one.

Formally, an alignment of sequences $S_1, \ldots, S_K$, $K \geq 2$, over an alphabet Σ is a $K \times \omega$ matrix $A = (a_{ij})$, $a_{i,j} \in \Sigma \cup \{-\}$, such that ignoring the blank character $-$, the ith row reproduces sequence S_i, and there is no column consisting only of blanks. By $A_{i_1,i_2,\ldots,i_k}$ we denote the projection of A to the sequences $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$.

For a pairwise projection of A to S_i and S_j let $c(A_{i,j})$ 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 $c(A) = \sum_{i<j} c(A_{i,j})$ is called the *sum of pairs (SP) cost* of the alignment A.

The gap penalty usually depends on the length of the gap. For pairwise alignments, the class of affine gap cost functions is widely acknowledged to yield good results. Although it is also possible to use affine gap costs for multiple alignments, Altschul [1] 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.

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.* [3] presented a branch-and-bound algorithm whose implementation – called MSA in the sequel – 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 [4,7,9]). 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 [2] applied in MSA, and (2) combining this algorithm with the DCA approach in order to develop an iterative procedure for computing multiple alignments with a nice time-versus-quality tradeoff.

In Section 2 we will review the techniques of Gupta *et al.* and the $\mathcal{A}^*$ algorithm. Section 3 describes the DCA method and how it is combined with the optimal alignment procedure. In Section 4 we give details of our implementation of this algorithm. In Section 5 finally we present results showing the practicality of the method.

2 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 path starting in d and ending with an edge $e = (u, v)$ corresponds to one possible alignment of the prefixes $S_{i_1}, \ldots, S_{i_K}$ induced by $v = (i_1, \ldots, i_K)$. (The dummy node and edge ensure that 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 corresponding suffixes of the sequences.

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 $c(f|e) = c(d \to^* e) + c(f) + \mathit{gappenalty}(e, f)$, and the cost of the shortest path ending with f is the minimum of $c(f|e)$ over all edges e incident to f.

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.* [3] 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 $L(w)$ 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.

The $\mathcal{A}^*$ algorithm employs basically the same bounding procedure with redefined edge costs. Thereby it speeds up computations by directing the search of a shortest path more towards the sink node t. (That is why this technique is also called *Goal Directed Unidirectional Search* (GDUS) [6]). 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.

It is worthwile noting that it can be shown [4, 7] that the above redefinition of edge costs implies the well-known Carrillo-Lipman bound [2]. 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.

3 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) [10, 12]. 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.

3.1 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.

In DCA the sequences are cut at certain positions near to their center, in the sequel called *cut positions*. 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 to the respective sequences 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 is stopped when the sequence length drops below this value.

Of course, 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. However, it has been shown that the heuristics of *minimal additional costs* using a variation of forward/backward matrices (which are well known from the study of local sequence similarities and suboptimal alignments) yields very good, in many cases optimal cut positions. For more details on the definition and computation of cut positions, a number of variations, and efficient speed-up techniques, see [10].

3.2 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, the following observation gets us even further. 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 the 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. Hence, one has to correct for this when using the sum of the scores of two adjacent alignments as an upper bound for the fused alignment in the next iteration (as shown in Figure 1).

4 Implementation

We have implemented the algorithms described in this paper 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) [8]. Our main emphasis was to create an open library that easily can be modified and/or extended. The parts of the library build a hierarchical, modular system, so that one can rearrange blocks or replace predefined modules by ones own classes. While the implemented algorithms reach state-of-the-art efficiency, one has to expect, however, that the library approach and the use of C++ classes will result in a fairly large constant factor in running time and memory performance.

The three layers of OMA's hierarchy of abstraction levels are:

- the *ProgramLayer* which implements the general structure of an alignment program with input/output and the main procedure;
- the *AlgorithmLayer* which implements the algorithms on a relatively abstract level (in our case the optimal $\mathcal{A}^*$ alignment procedure, DCA, and iteration);
- the *ConcreteLayer* which implements the data stuctures used by the other layers (sequences, alignment, distance matrices, faces, a trie, etc.).

This modular, hierarchical structure allows for the high flexibility we intended with our implementation. The general control structure is quickly defined on the *ProgramLayer* (in most cases, the supplied library will do), and the algorithms can then easily be built on the *AlgorithmLayer* using predefined OMA classes, without having to modify the more complicated classes of the *ConcreteLayer*.

Our predefined library comes with a complete set of classes for the *AlgorithmLayer* which implement the methods described in this paper. These classes are built together in a program called *oma* which is freely available from the address `http://www.poempel.com/towi/Beruf/oma.html`. It is the basis for the computations of the following section.

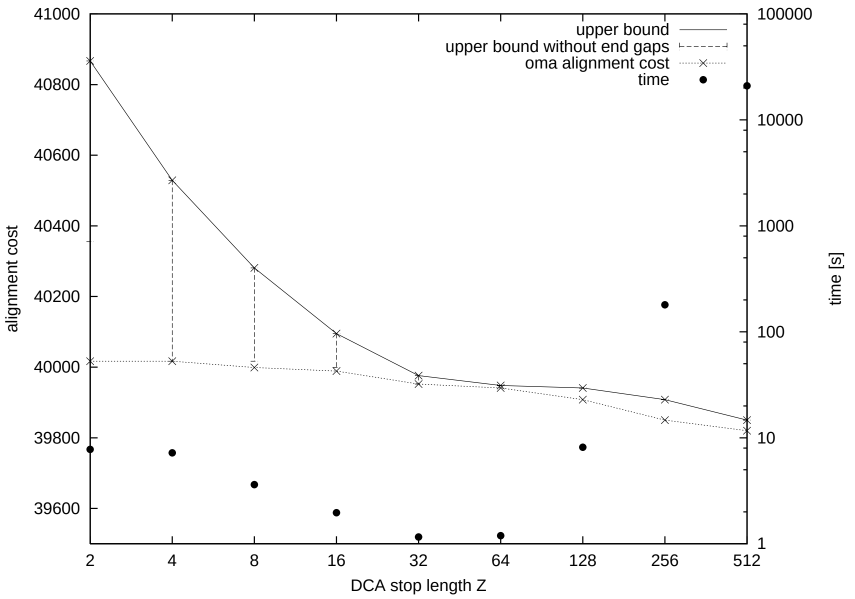

Figure 1: The successive improvement of the alignment cost

5 Results

We have run *oma* on a number of alignment problems from the Benchmark Alignments Database[1] (BAliBASE) [11]. All runs were performed on a Sun Enterprise 10000 with 2GB of addressable memory. We used Dayhoff's PAM 250 matrix with quasi-natural gap costs as the alignment cost function. The general result is that we can align a typical set of 4 to 6 protein sequences to optimality within 10 seconds up to a few minutes. Some more difficult examples, however, require excessive computation time and memory. If we stop the computation after 1 minute, we get a (sub-optimal) alignment in all test cases from the *reference1* subset of BAliBASE. A few detailed results follow.

Figure 1 shows for increasing values of Z the behavior 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 (increased by a correction for multiply counted end gaps at fusion points, see Section 3.2) upper-bounds the score of the *oma* alignment. A rather close-to-optimal alignment score is obtained already after a few iterations. However, the comparatively many alignments to be computed in the beginning also 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

[1] `http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE/`

	K	*length*	*avg. id.*	max. Z	cost	time (sec.)	memory (MB)
1ubi	4	76- 94	18	64	8626	58.7	29
1wit	5	89-106	17	64	16517	152.1	32
3cyr	4	95-109	31	128	9888	11.5	8
1pfc	5	108-117	28	64	17708	28.2	10
1fmb	4	98-104	49	64	8804	5.8	8
1fkj	5	98-110	44	64	15809	6.6	9
3grs	4	201-237	14	128	23478	97.7	30
1sbp	5	224-263	19	64	43149	>9000	?
1ad2	4	203-213	30	128	19714	20.0	10
2cba	5	237-259	26	128	40281	2342.4	215
1zin	4	206-216	42	128	19220	11.0	10
1amk	5	242-254	49	128	36659	19.1	9
2myr	4	340-474	16	128	43532	1920	411
1pamA	5	435-572	18	32	86482	98	16
1ac5	4	421-483	29	256	43325	2595	629
2ack	5	452-482	28	128	77137	2046	229
1ad3	4	424-447	47	256	39209	31	24
1rthA	5	526-541	42	512	80350	4737	684

Table 1: Best SP alignment costs obtained with the maximal value of Z that was computable on the used computer, computation time, and memory usage of *oma* for different test sets from *reference1* of BAliBASE

the 2GB of available memory. The number of edges explored in the search phase of the last alignment step which could be run, $Z = 256$, increases from 1.4×10^6 (with $\mathcal{A}^*$) to 2.2×10^6 (without $\mathcal{A}^*$). The computation time increases from 180 seconds to 549 seconds.

Table 1 shows some more results on short (top), medium length (middle), and long (bottom) sequences.[2] Each block is divided in distantly related, closer, and closely related sequences (see the column *avg. id.*). Note that the running time not only depends on the number and length of the input sequences but – like for most multiple alignment programs – it also highly depends in the similarity of the sequences.

Acknowledgments

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 compute facilities available to us.

References

[1] S. F. Altschul. Gap costs for multiple sequence alignment. *J. Theor. Biol.*, 138:297–309, 1989.

[2] For the complete results on all test sets from *reference1* of BAliBASE, and for a comparison with the program MSA, see `http://www.poempel.com/towi/Beruf/oma.html`.

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

[3] S. K. Gupta, J. D. Kececioglu, and A. A. Schäffer. 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, 1995.

[4] P. Horton. *String Algorithms and Machine Learning Applications for Computational Biology.* Ph. D. dissertation, University of California, Berkeley, CA, 1997.

[5] J. D. Kececioglu and W. Zhang. Aligning alignments. In M. Farach, editor, *Proceedings of CPM 1998*, number 1448 in LNCS, pages 189–208, Berlin, 1998. Springer Verlag.

[6] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley-Teubner, Chichester, 1990.

[7] M. Lermen and K. Reinert. The practical use of the $\mathcal{A}^*$ algorithm for exact multiple sequence alignment. *J. Comp. Biol.*, 1999. Accepted for publication. (See also Technical Report 97-1-028, MPI für Informatik, Saarbrücken, Germany, 1997.)

[8] K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, to appear 1999.

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

[10] J. Stoye. Multiple sequence alignment with the divide-and-conquer method. *Gene*, 211:GC45–GC56, 1998.

[11] J. D. Thompson, F. Plewniak, and O. Poch. BAliBASE: A benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, 15(1):87–88, 1999.

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

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