The Practical Use of the $\mathcal{A}^*$ Algorithm for Exact Multiple Sequence Alignment

Martin Lermen* Knut Reinert*

December 30, 1997

Abstract

Multiple alignment is an important problem in computational biology. It is well known that it can be solved exactly by a dynamic programming algorithm which in turn can be interpreted as a shortest path computation in a directed acyclic graph. The $\mathcal{A}^*$ algorithm (or goal directed unidirectional search) is a technique that speeds up the computation of a shortest path by transforming the edge lengths without losing the optimality of the shortest path. We implemented the $\mathcal{A}^*$ algorithm in a computer program similar to MSA [GKS95] and FMA [SI97b]. We incorporated in this program new bounding strategies for both, lower and upper bounds and show that the $\mathcal{A}^*$ algorithm, together with our improvements, can speed up computations considerably. Additionally we show that the $\mathcal{A}^*$ algorithm together with a standard bounding technique is superior to the well known Carillo-Lipman bounding since it excludes more nodes from consideration.

1 Introduction

One of the most prominent problems in computational molecular biology is multiple sequence alignment. It is used for extracting and representing biologically important commonalities from a set of sequences. It is easy to generalize the standard algorithm of Needleman and Wunsch ([NW70]) to more than two sequences. However the time and space complexity grows exponentially in the number of sequences. Solving the problem to optimality is therefore only tractable for small problem instances. Nevertheless exact algorithms are important, because they can be used as a last step of algorithms that use motif-search or divide-and-conquer approaches. For example Stoye et al. ([SMD97]) try in their approach to divide the sequences at appropriate "slicing" locations which are determined through a branch-and-bound procedure. The resulting subproblems are solved recursively. The recursion stops if the lengths of the sequences in a subproblem fall below a certain threshold. The subproblem is then solved to optimality. Of course this approach tries to end the recursion as soon as possible. Therefore programs are needed that can solve large instances with many sequences to optimality. We refrain from citing further seminal papers concerning pairwise and multiple alignment, because by now a general methodology has been established and the three quite recently published monographs (Gusfield [Gus97], Setubal and Meidanis [SM97], Waterman [Wat95]) give an excellent overview and motivation for the problem.

In this paper we show that the application of the so-called $\mathcal{A}^*$ algorithm, together with new strategies for computing better lower and upper bounds, considerably speeds up the computation of optimal multiple alignments. We implemented the $\mathcal{A}^*$ algorithm in a computer program (GSA) in which we incorporated these new bounding strategies and compared our program with other implementations. Additionally we show that the $\mathcal{A}^*$ algorithm together with a standard bounding technique is superior to the well known Carillo-Lipman bounding since it excludes more nodes from consideration. We conjecture that the speedup

*Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany.

imposed by our techniques will be transfered to an ongoing implementation of our algorithm that supports affine gap costs.

We now define the problem formally. Let $S_1, \ldots, S_K$, $K \geq 2$ be sequences of length $N_1, \ldots, N_K$ over an alphabet Σ which must not contain the reserved blank character '$-$' and define $\Sigma' := \Sigma \cup \{-\}$. A *multiple alignment* of these strings is a $K \times \omega$ matrix $A = (a_{ij})$ with the following properties:

1. A has exactly K rows,
2. ignoring the blank character, the i-th row is the sequence S_i,
3. there is no column consisting only of blank characters.

We denote with $\omega = \omega(A)$ the number of columns of A and with $A_{i_1,i_2,\ldots,i_k}$ the projection of A to the sequences $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$. An alignment of a subset $\{S_{i_1}, \ldots, S_{i_k}\}$ of the K strings is denoted by $A(S_{i_1}, \ldots, S_{i_k})$ (e.g. $A(S_i, S_j, S_k)$ is an alignment of the three sequences S_i, S_j and S_k).

The quality of an alignment is often measured with a function over the columns. The cost measure that is most widely used is the *(weighted) sum of pairs cost* ((W)SOP) which is defined as follows: If *sub* is a fixed symmetrical function $sub : \Sigma' \times \Sigma' \to \mathbf{N} = \{0, 1, 2, \ldots\}$ with $sub(-,-) = 0$ then we define $c(A_{i,j}) = \sum_{l=1}^{\omega} sub(a_{il}, a_{jl})$ as the cost of the projection of A to the sequences S_i and S_j. Using this definition the *sum of pairs* (SOP) cost function is defined as

$$c(A) = \sum_{1 \leq i < j \leq K} c(A_{i,j}) = \sum_{1 \leq i < j \leq K} \left(\sum_{l=1}^{\omega} sub(a_{il}, a_{jl}) \right).$$

The goal is then to compute a minimum cost SOP alignment A^*. For convenience we will only talk about the SOP cost measure, however, all our results hold for the WSOP measure as well. In the sequel we will denote optimal alignments with the superscript $*$, i.e., A^*_{ij} is the projection of an optimal alignment A^* to the sequences S_i and S_j and $A^*(S_i, S_j)$ is an optimal alignment for the sequences S_i and S_j.

Like most multiple alignment problems, the SOP alignment problem can be solved by dynamic programming and is equivalent to finding a shortest path from a designated source to a designated sink in a K-dimensional acyclic mesh-shaped graph, the so-called *dynamic programming graph*. The set of paths from the source to the sink codes all possible alignments. Each (directed) edge of the dynamic programming graph represents a possible column. The weight of such an edge is the SOP cost of the column it represents. Dynamic programming yields an algorithm with time complexity $O(K^2 2^K N)$ and space complexity $O(N)$, where $N = \prod_i N_i$ which is feasible only for very small problem instances. While the SOP alignment problem is NP-complete, Kececioglu et al. presented in [GKS95] a branch-and-bound algorithm whose implementation (c.f. [KAL94]) – 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 their approach, a heuristic alignment of the K sequences yields an upper bound for the branch-and-bound procedure. Lower bounds are calculated by adding up the cost of all optimal pairwise alignments over suffixes of the sequences.

The $\mathcal{A}^*$ algorithm also computes a shortest path in the dynamic programming graph with redefined edge weights which improves the speed of the computation considerably. Shibuya et al. presented in [SI97b] an implementation of the $\mathcal{A}^*$ algorithm – called FMA in the sequel – which they used for a parametric analysis for multiple sequence alignment.

In Section 2 we will review the techniques of Kececioglu et al., Carillo and Lipman and the $\mathcal{A}^*$ algorithm in more detail and finally prove that the $\mathcal{A}^*$ algorithm bounds the number of explored vertices in the dynamic programming graph more efficiently than the Carillo-Lipman technique. In Section 3 we explain three ideas, each of which gives great performance improvements in the implementation. The first idea is not only to use projections to two but also to three sequences to obtain a better lower bound. The second idea is to periodically recompute a better upper bound during the execution of the algorithm. The third idea consists of enumerating the neighbors of a node q in a clever way, so that only small incremental changes are necessary when moving from edge to edge. Finally we give some computational results in Section 4 to show how our enhancements compare to the two implementations of Kececioglu et al. and Shibuya et al.

2 Shortest path computation and the $\mathcal{A}^*$ algorithm

An alignment of the K sequences can be interpreted as a path in a K-dimensional grid graph with node set:

$$V = \{v = (v[1], v[2], \ldots, v[K]) : v[i] \in \{1, \ldots, N_i\}\}$$

and edges

$$E = \{(p, q) : p, q \in V,\ p \neq q \text{ and } q - p \in \{0, 1\}^K\}$$

Let us denote the set of all paths from a node p to a node q by $p \to q$.

$$p \to q := \left\{(p = v_0,\ v_1,\ \ldots\ ,\ v_\omega = q) : (v_i, v_{i+1}) \in E,\ 0 \leq i < \omega\right\}$$

Likewise we write $p \to q \to r$ for the set of paths from p to r passing through node q. A path π of length ω from $s = (0, \ldots, 0)$ to $t = (N_1, \ldots, N_K)$ corresponds to the alignment described by the following matrix:

$$a_{ij} = \begin{cases} - & \text{if } v_j[i] - v_{j-1}[i] = 0 \\ S_i[v_j[i]] & \text{if } v_j[i] - v_{j-1}[i] = 1 \end{cases} \qquad \text{for } 1 \leq i \leq K,\ 1 \leq j \leq \omega$$

Thus, the cost of an alignment can be interpreted as the sum of edge costs

$$c(\pi) := \sum_{i=0}^{\omega-1} c(v_i, v_{i+1})$$

with

$$c(v_i, v_{i+1}) := \sum_{1 \leq k < l \leq K} sub(a_{kx}, a_{lx})$$

where x denotes the number of the column corresponding to the edge (v_i, v_{i+1}). Note that we overload the notation of c because an edge in the dynamic programming graph corresponds exactly to a column of a multiple alignment. We denote the shortest path in $p \to q$ by $p \to^* q$ and its length by $c(p \to^* q)$. A node r in the grid naturally divides each sequence S_i in a prefix α_i^r and a suffix σ_i^r.

Of course it is not feasible to compute a shortest path in the full grid graph, since its size is $O(N = \prod_i N_i)$. Using Dijkstra's algorithm for computing shortest paths in graphs with nonnegative edge costs, Kececioglu et al. described in [GKS95] an algorithm which reduces the number of nodes that have to be visited by a bounding procedure. In their approach, a heuristic alignment A^{heur} of the K sequences yields an upper bound $U = c(A^{heur})$ for the cost of the optimal alignment A^*. Lower bounds $L(r \to t)$ are calculated by adding up the cost of all optimal pairwise alignments over suffixes of the sequences, i.e., for each node r in the grid graph we have:

$$L(r \to t) := \sum_{1 \leq i < j \leq K} c(A^*(\sigma_i^r, \sigma_j^r)) \tag{1}$$

The algorithm uses a priority queue Q, where it stores the values of the best known paths for prefixes of the sequences as keys. In each step the node q with minimum key k is extracted from Q and then *expanded*, which means that all neighbors r of q are inserted in Q with the key $k + c(q, r)$. Dijkstra's algorithm ensures that the key k of the node q with the minimal key is always the cost of the shortest path from s to q, i.e., $k = c(s \to^* q)$. In the expansion of a node q one does not need to insert a neighbor r if $c(s \to^* q) + c(q, r) + L(r \to t) > U$. That means if the sum of the length of the optimal path from s to q plus the length of the edge (q, r) plus the lower bound $L(r \to t)$ is already greater than an upper bound U, then no optimal alignment A^* can go through r. Later we will see that this simple bounding strategy applied to a dynamic programming graph with changed edge weights always yields better results than the well known Carillo-Lipman bounding.

Carillo and Lipman employ a different idea to reduce the number of vertices in the dynamic programming graph. The following property holds for any optimal multiple alignment A^* (c.f. [CL88]):

Theorem 2.1 (Carillo, Lipman) *Let A^* be an optimal alignment of the K strings $S_1, \ldots, S_K$, $L := L(s \to t)$ be the lower bound defined in Equation 1 and $U = c(A^{heur})$ be an upper bound for $c(A^*)$. Then the following inequality holds for every projection on a pair S_i, S_j of sequences:*

$$c(A^*_{i,j}) \leq c(A^*(S_i, S_j)) + U - L$$

Proof:

$$\begin{aligned} U - L &\geq \sum_{1 \leq k < l \leq K} (c(A^*_{k,l}) - c(A^*(S_k, S_l))) \\ &\geq c(A^*_{i,j}) - c(A^*(S_i, S_j)),\ \forall\ 1 \leq i < j \leq K \\ \Rightarrow c(A^*_{i,j}) &\leq c(A^*(S_i, S_j)) + U - L \end{aligned}$$

∎

Due to Theorem (2.1) an optimal alignment path cannot pass through a node r if for any pair i, j holds:

$$CL_{i,j}(r) := c(A^*(\alpha^r_i, \alpha^r_j)) + c(A^*(\sigma^r_i, \sigma^r_j)) - c(A^*(S_i, S_j)) + L(s \to t) > U.$$

We call a node r *CL-valid* if $CL_{i,j}(r) \leq U$ for all pairs i, j. Otherwise we call it *CL-invalid*. A representation of the set of CL-valid nodes can be precomputed, so that we can efficiently decide whether a given node q is CL-valid or not.

The $\mathcal{A}^*$-algorithm speeds up computations by directing the search of a shortest source-to-sink path more towards the sink node t. It redefines the cost of all edges in E as follows:

$$c'(q, r) := c(q, r) - l(q \to t) + l(r \to t),$$

where $l(u \to v)$ is a lower bound for the cost $c(u \to^* v)$ of a shortest path from u to v. If $l()$ fulfills the *consistency* condition

$$c(q, r) + l(r \to t) \geq l(q \to t),\ \forall (q, r) \in E,$$

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 $L(q \to t)$ as the lower bound in the redefinition of the edge weights, because L fulfills the consistency condition:

$$\begin{aligned} c(q, r) + L(r \to t) &= \sum_{1 \leq i < j \leq K} (c(A^*(\sigma^r_i, \sigma^r_j)) + c(q, r) \\ &\geq \sum_{1 \leq i < j \leq K} (c(A^*(\sigma^q_i, \sigma^q_j)) \\ &= L(q \to t) \end{aligned}$$

The redefinition of the edge weights directs the search in the grid more towards the sink node t. Therefore this technique is also called *Goal Directed Unidirectional Search (GDUS)* (c.f. [Len90]). We now want to apply the simple bounding procedure described above. With the new edge weights a shortest path from s to q has the length $c'(s \to^* q) = c(s \to^* q) + L(q \to t) - L(s \to t)$. Since the value $L(s \to t)$ is a precomputed constant we discard it and insert a neighboring node r with the key $\textsc{Prio}_q(r)$ into the priority queue Q, where

$$\textsc{Prio}_q(r) := c(s \to^* q) + c(q, r) + L(r \to t). \tag{2}$$

Applying the bounding procedure, we do not always have to insert a node r into the priority queue Q, namely if $\textsc{Prio}_q(r) > U$ where U is an upper bound for $c(s \to^* t)$. The argumentation is as before, because $\textsc{Prio}_q(r)$ is a lower bound for a path from s to t passing through q and r. This does not exclude that r might be inserted later if it is inspected from another neighbor q'. However, if r is never inserted into Q, i.e., if $\textsc{Prio}(r) := \min_{(q,r) \in E}\ \textsc{Prio}_q(r) > U$, then we call r *U-invalid*, otherwise we call it *U-valid*. We will now show that the redefining of the edge costs together with the simple bounding strategy always yields better results than the bounding achieved by the Carillo-Lipman technique.

Theorem 2.2 *CL-invalidity implies U-invalidity.*

Proof:

$$\begin{aligned}
\mathrm{CL}_{i,j}(q) &= c(A^*(\alpha_i^q,\alpha_j^q)) + c(A^*(\sigma_i^q,\sigma_j^q)) - c(A^*(S_i,S_j)) + L(s \to t) \\
&= c(A^*(\alpha_i^q,\alpha_j^q)) + c(A^*(\sigma_i^q,\sigma_j^q)) - c(A^*(S_i,S_j)) + \sum_{1\le k<l\le K} c(A^*(S_k,S_l)) \\
&= c(A^*(\alpha_i^q,\alpha_j^q)) + c(A^*(\sigma_i^q,\sigma_j^q)) + \sum_{\substack{1\le k<l\le K \\ (k,l)\neq(i,j)}} c(A^*(S_k,S_l)) \\
&\le c(A^*(\alpha_i^q,\alpha_j^q)) + c(A^*(\sigma_i^q,\sigma_j^q)) + \sum_{\substack{1\le k<l\le K \\ (k,l)\neq(i,j)}} \Big(c(A^*(\alpha_k^q,\alpha_l^q)) + c(A^*(\sigma_k^q,\sigma_l^q))\Big) \\
&= \sum_{1\le k<l\le K} c(A^*(\alpha_k^q,\alpha_l^q)) + \sum_{1\le k<l\le K} c(A^*(\sigma_k^q,\sigma_l^q)) \\
&= \sum_{1\le k<l\le K} c(A^*(\alpha_k^q,\alpha_l^q)) + L(q \to t) \\
&\le c(s \to^* q) + L(q \to t) \\
&=: \textsc{Prio}(q)
\end{aligned}$$

If $\mathrm{CL}_{i,j}(q) > U$, then $\textsc{Prio}(q) > U$, so q is always U-invalid if it is CL-invalid. ∎

The above proof was first published in the master thesis of the first author ([Ler97]), however, the authors acknowledge that the above theorem has been shown independently by Horton and Lawler in [Hor97].

The U-bounding reduces the number of relevant nodes substantially. However, we still explore enough of the grid to guarantee that the computed alignment is optimal. Now we give up that guarantee in order to obtain an even better bounding. This is done similarly to the program of Kececioglu et al. whereas Shibuya et al. do not employ this technique. Let A^{heur} be a heuristically computed alignment yielding an upper bound U. We define

$$eps_{i,j} := c(A^{heur}_{i,j}) - c(A^*(S_i,S_j))$$

and

$$EPS_{i,j} := \min(\max(eps_{i,j}, \text{MIN_EPS}), \text{MAX_EPS})$$

where MIN_EPS, MAX_EPS are nonnegative constants with MIN_EPS≤MAX_EPS.

We make the (not always true) assumption, that any alignment A^π for a path $\pi \in s \to r \to t$ cannot be optimal, if for a pair (i,j)

$$c(A^*(\alpha_i^r,\alpha_j^r)) + c(A^*(\sigma_i^r,\sigma_j^r)) > c(A^*(S_i,S_j)) + EPS_{i,j} \tag{3}$$

and call r *face-invalid* in this case, otherwise *face-valid*. The following choice produces good results:

$$\text{MIN_EPS} := \frac{2 \cdot \sum_{1\le i<j\le K}(eps_{i,j})}{K(K-1)}$$

Information on face-invalid nodes can be efficiently precomputed and stored using so-called faces. For each pair of dimensions (i,j) a face merely consists of two arrays of the size $O(max(N_i,N_j))$. The values MIN_EPS and MAX_EPS are crucial parameters. If MIN_EPS is set too low, it can be that we do not find an optimal alignment, since nodes on an optimal path in $s \to t$ might then be ignored. If we set it to high, removing the effect of face-bounding, the time and space consumption rises. If we set the MAX_EPS parameter too low it is often the case that we do not find an alignment at all. We will discuss this problem in more detail in Section 4.

3 Improvements

In this section we explain three ideas, each of which results in great performance improvements in the implementation. The first idea is not to use only projections to two but also to three sequences to obtain better lower bounds. The second idea is to periodically recompute a tighter upper bound during the execution of the algorithm and finally we enumerate the neighbors of a node q in a clever way, so that only small incremental changes are needed when moving from edge to edge.

3.1 Triple alignments

Already Carillo and Lipman noted that their idea of reducing the volume of the search-space by using lower-dimensional optimal alignments can be extended to higher dimensions. Unfortunately the number of optimal d-dimensional alignments is $O(\binom{K}{d})$, and what is even worse, the space consumption is $O(\binom{K}{d} \cdot N)$, where $N = \prod_{i=i_1}^{i_d} N_i$. In this section we show how to carefully select a reasonable number of triples of strings, the alignments of which yield good lower bounds compared to the bounds that are achieved by computing pairwise alignments.

We replace three optimal pair alignments with one optimal *triple alignment*. This is allowed, since the projection of an optimal multiple alignment to three strings can never yield a better alignment than the optimal alignment of these three strings. On the other hand, the cost for the optimal triple alignment is never smaller than the sum of the three optimal pairwise alignment costs.

We want to find a set of triples (more precisely a selection of sets of three elements from the set of indices)

$$\mathcal{T} = \Big\{ \{i,j,k\} : i,j,k \in \{1,2,\ldots,K\} \text{ and } \; i \neq j \neq k \Big\}$$

with the additional property

$$|\sigma \cap \tau| < 2 \text{ for all } \sigma, \tau \in \mathcal{T}, \sigma \neq \tau.$$

That means, we want to find a set of triples with no common pair. For a given set $\mathcal{T}$ we also define

$$\mathcal{P} := \{(i,j),(j,k),(i,k) : \forall \{i,j,k\} \in \mathcal{T}\}$$

For any optimal alignment $A^{r,*}$ going through node r we define:

$$\begin{aligned} L_3(r \to t) \quad &:= \quad \sum_{\{i,j,k\} \in \mathcal{T}} c(A^*(\sigma_i^r, \sigma_j^r, \sigma_k^r)) + \sum_{(i,j) \notin \mathcal{P}} c(A^*(\sigma_i^r, \sigma_j^r)) \\ &\geq \quad \sum_{1 \leq i < j \leq K} c(A^*(\sigma_i^r, \sigma_j^r)) = L(r \to t) \end{aligned}$$

Computing $L_3(r \to t)$ usually yields a much tighter lower bound than $L(r \to t)$. The new lower bound L_3 can be shown to fulfill the consistency condition in the same way as L in Equation 2. Note that U-bounding is also improved, since the the value of $\textsc{Prio}(q)$ increases if L_3 is used. How much the bounds are improved depends on the choice of the triples. We adopted the following heuristic which yields good results: for each triple $\{i,j,k\}$ compute

$$D(i,j,k) := c(A^*(S_i,S_j,S_k)) - c(A^*(S_i,S_j)) - c(A^*(S_j,S_k)) - c(A^*(S_i,S_k)).$$

We select a triple $\{i,j,k\}$ if

1. $D(i,j,k) > 0$ and
2. there is no triple $\{i',j',k'\}$ with $|\{i,j,k\} \cap \{i',j',k'\}| \geq 2$ and $D(i',j',k') \geq D(i,j,k)$.

The difficulty in using triple alignments for better lower bounds is that we have to precompute and store a three-dimensional grid with the values $c(A^*(\sigma_i^r, \sigma_j^r, \sigma_k^r))$ (distance grid) for every used triple, which is quite space consuming. Such a distance grid for $\{i,j,k\}$ can be

computed with the Dijkstra-algorithm. In fact, we only need those values $c(A^*(\sigma_i^r, \sigma_j^r, \sigma_k^r))$ which we want to compute $L_3(r \to t)$ for, namely for the nodes r that are face-valid for the K-dimensional problem. Therefore we can cut off large areas of the three-dimensional distance grids if we use face-bounding with respect to the $EPS(i,j)$ of the K-dimensional problem.

3.2 Dynamic Upper Bound

In the preceding section we described how the volume of the search space in the multi-dimensional grid can be reduced by ignoring U-invalid nodes. We can do even better if it is possible to improve the upper bound during the progress of the algorithm.

For each node q we know its optimal distance $c(s \to^* q)$ from the source s as soon as it is removed from the priority queue in Dijkstra's algorithm. The closer we get to the sink t, the better the chance that

$$U' := c(s \to^* q) + c(A^{q,heur})$$

defines a better upper bound than U. In this term, $A^{q,heur}$ is a heuristically computed alignment of the suffixes $\sigma_1^q, \ldots, \sigma_K^q$.

If $U' < U$, then more nodes can be ignored because of U-invalidity (more precisely: U'-invalidity) than it was possible before. Therefore we compute at "promising" *trial nodes* q, just removed from the priority queue, a heuristic alignment of the suffixes and try to improve the upper bound *(dynamic upper bound)*. We have to select those trial nodes carefully because computing the heuristic multiple alignment is time-consuming, especially in the beginning, when the path from q to t is still long. Two heuristics were tested in order to select the trial nodes, each one with acceptable results. In both cases the first trial node is $u_0 = s$.

Heuristic 1
The node q becomes a new trial node u_{i+1}, if

$$c(s \to^* q) > c(s \to^* u_i) + \frac{c(A^{u_0,heur}) - L(s \to t)}{const_1}$$

where u_i was the previous trial node and $const_1 \geq 1$ is a constant, indicating the maximum total number of trial nodes. Heuristic 1 regards a trial node q as good, if the algorithm has made enough progress with respect to the cost of the optimal prefix path of the previous trial node.

Heuristic 2
The node q becomes a new trial node u_{i+1}, if

$$StepCount(s \to q) = StepCount(s \to u_i) + const_2$$

where u_i was the previous trial node, $const_2 \geq 1$ is a constant, and $StepCount(s \to r)$ is the number of edges on an optimal path from s to r. Heuristic 2 regards a trial node q as good, if the algorithm has made enough progress with respect to the number of steps the algorithm has made since visiting the previous trial node.

In our implementation we preferred heuristic 1 because it does not require additional memory space. After successfully improving the upper bound from U to U' it is possible to search the grid for U-valid nodes, which have now become U'-invalid. All edges to these nodes that have been visited up to that stage of the algorithm can be ignored and the nodes can be deleted from the grid *(garbage collection)*.

3.3 Gray-Code

For every node q that is extracted from the priority queue the $\mathcal{A}^*$ algorithm examines all outgoing edges to neighboring nodes r. The difference vector $r - q$ is an element of $\{0,1\}^n - \{(0,\ldots,0)\}$. All neighbors $r^1, \ldots, r^{2^K-1}$ have to be checked if they are within the bounds of the grid. If so, r^i must then be checked for face-validity. Representing the node r^i with its coordinates in a trie usually takes time $O(K)$. This time can be substantially reduced if the neighbors are enumerated in *Gray-Code succession*. Gray-Code is an enumeration of

all elements of $\{0,1\}^n - \{(0,\ldots,0)\}$, where subsequent nodes only differ in one dimension. If we reuse the representation of r^i in order to get r^{i+1} by changing only one integer, only constant time is needed for each neighbor. For notational convenience we define r^0 as q. Let $\ldots i_{[3]} i_{[2]} i_{[1]}$ be the binary representation of i. Define

$$b^i := \begin{cases} 0 & \text{for } i = 0 \\ j & \text{else, where } j \text{ minimal with } i_{[j]}=1 \end{cases}$$

and

$$\begin{aligned} d^0 &:= 0 \\ d^i &:= (d^{i-1}) \text{ XOR } 2^{b^i-1} \text{ for } 1 \le i \le 2^K - 1 \end{aligned}$$

The neighbors are given as

$$\begin{aligned} r^i &:= q + d^i \\ &= (q[1] + d^i_{[K-1]},\, q[2] + d^i_{[K-2]},\, \ldots,\, q[K] + d^i_{[0]}) \end{aligned}$$

The Gray-Code technique is used in the algorithm in two further ways. Firstly it can be used to check in two steps if the neighbor r^i of a node q is within the grid bounds. We mark all indices i with $q[i] + 1 > N_i$ in a bit field a of length K by setting the bit $a_{[i]}$. Before r^1 is checked, all bits of a should be set to 0. A neighbor r^i for $i > 0$ is within the bounds if

$$0 \le r^i[K - b^i - 1] \le N_{K-b^i-1} \text{ and } (d^i \text{ AND } a) = 0.$$

If $r^i[K-b^i-1] > N_{K-b^i-1}$ then this is marked by setting the bit $a_{[i]}$ for subsequent neighbors.

Secondly the computation of edge costs can be speed up. As described above, every edge (q, r^i) uniquely defines a column of an alignment. We store this column in an array $char^i$ of length K over the alphabet Σ':

$$char^i[j] = \begin{cases} - & \text{if } r^i[j] - q[j] = 0 \\ S_j[r[j]] & \text{if } r^i[j] - q[j] = 1 \end{cases} \quad \text{for } 1 \le i \le K$$

If $char^0$ is initialized with blank characters '$-$', then $char^i$ can be obtained from $char^{i-1}$ by modification of the single character $char[K - b^i - 1]$. Additionally one could compute the cost of an edge in time $O(K)$. One only needs to subtract the pairwise costs between the x-th ($x = K - b^i - 1$) character in $char^{i-1}$ and all others and add the pairwise costs between the x-th character in $char^i$ and all others to the previous cost value $c(q, r^{i-1})$.

$$c(q, r^i) = c(q, r^{i-1}) - \sum_{1 \le k \ne x \le K} \Big(sub(char^{i-1}[k], char^{i-1}[x]) - sub(char^i[k], char^i[x]) \Big) \quad (4)$$

It is easy to check that for $K \ge 5$ the right hand side of Equation 4 has less terms than the right hand side of the usual computation of the edge costs which is:

$$c(q, r^i) = \sum_{1 \le k < l \le K} sub(char^i[k], char^i[l]) \quad (5)$$

However, in practice it turned out to be faster to use Equation 5 for edge cost computation. This is due to two reasons:

1. The cost for evaluating Equation 5 are in practice seldom $O(K^2)$ since one can check for U-invalidity each time after adding K terms, say. Very often only a few checks are needed to prove that a node is U-invalid.
2. On the other hand, if one uses Equation 4 to compute the edge costs, it is necessary to compute the costs to all neighboring nodes, even if they are face-invalid. This imposes a considerable overhead compared to the other method which does not consider face-invalid nodes.

4 Computational results

We implemented the described algorithm in C++ using the library of efficient data types and algorithms LEDA (c.f. [MN95]). Although this imposes a time and space overhead by a factor of 2 to 3 compared to ad hoc implementations it makes the software easy to read, to maintain, and to extend. Based on our implementation (GSA=goal directed sequence alignment) there is an ongoing implementation of a version supporting affine gap costs which is intended to replace MSA as a last step of the divide-and-conquer approach of Stoye et al. ([SMD97]). In their approach they try to divide the K sequences at appropriate "slicing" locations which are determined through a branch-and-bound procedure. The resulting subproblems are solved recursively. The recursion stops if the lengths of the sequences in a subproblem fall below a certain threshold. The subproblem is then solved to optimality, currently using MSA. Of course this approach tries to end the recursion as soon as possible. Therefore programs are needed that can solve large instances with many sequences to optimality.

We divide this section into two parts. In the first part we discuss the effect of the algorithmic techniques introduced before (dynamic upper bound, GDUS, triple alignments) on the time and space consumption of the algorithm. In the second part we compare our implementation with two other packages for optimal sequence alignment, namely the widely known program MSA in its latest version 2.1 [KAL94], and FMA, a very recent implementation by Shibuya and Imai [SI97a], which also uses an $\mathcal{A}^*$ strategy. In order not to have an advantage against FMA or MSA, we use their two default cost matrices which are called `dayhoff` (MSA) and `PAM250` (FMA).

All examples were run on an UltraSparc Station 2/200 with 1024 MB memory. The program together with all examples and converting tools can be obtained via anonymous ftp from `ftp://ftp.mpi-sb.mpg.de/pub/outgoing/reinert/GSA.tgz`. The LEDA Software library can be downloaded from `http://www.mpi-sb.mpg.de/LEDA/leda.html`. We tested combinations of the above mentioned flags on the following three examples. The first is an benchmark example of five protein fragments that are relatively easy to align:

Example 1:

```
1: ASVLTQPPSVSGAPGQRVTISCTGSSSNIGAGHNVKWYQQLPGTAPKLLIFHNNARFSVSKSGT
2: QSVLTQPPSASGTPGQRVTISCSGTSSNIGSSTVNWYQQLPGMAPKLLIYRDAMRPSGVPDRFS
3: EVQLVQSGGGVVQPGRSLRLSCSSSGFIFSSYAMYWVRQAPGKGLEWVAIIWDDGSDQHYADSV
4: AVQLEQSGPGLVRPSQTLSLTCTVSGTSFDDYYWTWVRQPPGRGLEWIGYVFYTGTTLLDPSLR
5: PSVFLFPPKPKDTLMISRTPEVTCVVVDVSHEDPQVKFNWYVDGVQVHNAKTKPREQQYNSTYR
```

```
1: SATLAITGLQAEDEADYYCQSYDRSLRVFGGGTKLTVLR
2: GSKSGASASLAIGGLQSEDETDYYCAAWDVSLNAYVFGTGTKVTVLGQ
3: KGRFTISRNDSKNTLFLQMDSLRPEDTGVYFCARDGGHGFCSSASCFGPDYWGQGTPVTVSS
4: GRVTMLVNTSKNQFSLRLSSVTAADTAVYYCARNLIAGGIDVWGQGSLVTVSS
5: VVSVLTVLHQNWLDGKEYKCKVSNKALPAPIEKTISKAKG
```

The next two examples are taken from McClure's globin dataset. In the second example there are 11 fragments from this dataset.

Example 2: 11 fragments of McClure's globin dataset

```
1 : VLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
2 : MLTDAEKKEVTALWGKAAGHGEEYGAEALERLFQAFPTTKTYFSHFDLSHGSAQIKAHGK
3 : VLSAADKTNVKGVFSKIGGHAEEYGAETLERMFIAYPQTKTYFPHFDLSHGSAQIKAHGK
4 : VHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKV
5 : VHLSGGEKSAVTNLWGKVNINELGGEALGRLLVVYPWTQRFFEAFGDLSSAGAVMGNPKV
6 : VHWTAEEKQLITGLWGKVNVADCGAEALARLLIVYPWTQRFFASFGNLSSPTAILGNPMV
7 : GLSDGEWQLVLNVWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKFKHLKSEDEMKASED
8 : GLSDGEWQLVLKVWGKVEGDLPGHGQEVLIRLFKTHPETLEKFDKFKGLKTEDEMKASAD
9 : MKFFAVLALCIVGAIASPLTADEASLVQSSWKAVSHNEVEILAAVFAAYPDIQNKFSQFA
10: GVLTDVQVALVKSSFEEFNANIPKNTHRFFTLVLEIAPGAKDLFSFLKGSSEVPQNNPDL
11: MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVRPLFDMGRQESLEQPKALAMT
```

The third example is a set from 5 complete sequences from the globin dataset.

Example 3: 5 sequences of McClure's globin dataset

```
1: VLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVAD
2: VHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHG
3: MKFFAVLALCIVGAIASPLTADEASLVQSSWKAVSHNEVEILAAVFAAYPDIQNKFSQFAGKDL
4: GVLTDVQVALVKSSFEEFNANIPKNTHRFFTLVLEIAPGAKDLFSFLKGSSEVPQNNPDLQAHA
5: MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVRPLFDMGRQESLEQPKALAMTVLAA

1: ALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKF
2: KKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQA
3: ASIKDTGAFATHATRIVSFLSEVIALSGNTSNAAAVNSLVSKLGDDHKARGVSAAQFGEFRTAL
4: GKVFKLTYEAAIQLEVNGAVASDATLKSLGSVHVSKGVVDAHFPVVKEAILKTIKEVVGDKWSE
5: AQNIENLPAILPAVKKIAVKHCQAGVAAAHYPIVGQELLGAIKEVLGDAATDDILDAWGKAYGV

1: LASVSTVLTSKYR
2: AYQKVVAGVANALAHKYH
3: VAYLQANVSWGDNVAAAWNKALDNTFAIVVPRL
4: ELNTAWTIAYDELAIIIKKEMKDAA
5: IADVFIQVEADLYAQAVE
```

4.1 Different GSA runs

We demonstrate the effects of the above mentioned algorithmic techniques on the time and space consumption of GSA. The following flags for GSA were used:

- `-`
  Without any flag, GSA uses the GDUS strategy. It does not use the dynamic upper bound but the triple alignments (see Section 3.1) to achieve better lower bounds. The value MIN_EPS is computed as explained in Section 2. By default the unit cost edit distance is used to compute the cost of an edge.
- `-2`
  With this flag, GSA only uses pairwise projections for computing lower bounds.
- `-u`
  With this flag, GSA uses the dynamic upper bound strategy described in Section 3.2
- `-a`
  This flag prevents the insertion of U-invalid nodes into the trie. This slightly slows down the computation but uses slightly less space.
- `-g`
  This flag disables the GDUS strategy.
- `-e<x>`
  This flag overrides the computation of the MIN_EPS values described in Section 2 and sets the value to x.
- `-c<cost matrix>`
  This flag overrides the unit cost edit distance and rather computes the cost of an edge using the specified cost matrix.

The time and space consumption for the different flags is given in the following two tables using the PAM250 cost matrix (`-cpam250`). The first table shows the result with the GDUS strategy:

	Example 1		Example 2		Example 3	
flags	time (sec)	space (MB)	time (sec)	space (MB)	time (sec)	space (MB)
-	21	17	5.8	6.2	57	47
-a	24	17	6	6	62	47
-2	33	14	4.6	5.6	233	87
-a2	35	14	5.6	5.5	256	87
-u	37	14	20	7.2	80	34
-au	38	14	22	7.1	87	33

The second table shows the results without the GDUS strategy (flag: `-g`).

	Example 1		Example 2		Example 3	
flags	time (sec)	space (MB)	time (sec)	space (MB)	time (sec)	space (MB)
-g	52	9.1	13.7	6.3	887	23
-ag	58	8.6	16	6.2	998	23
-2g	54	7.2	11.4	5.6	1036	20
-a2g	58	7.2	13.9	5.5	1081	19
-ug	58	7.7	23	7	895	25
-aug	56	7.5	26	7	543	21

The above table justifies the following observations which are strengthened by further examples:

1. The GDUS strategy considerably speeds up the computations but it uses more space. This effect gets more dramatic with bigger problem instances. Therefore the user has to decide whether time or space is the limiting factor and use the `-g` flag accordingly.
2. The dynamic upper bound slows down the computations but reduces space consumption for bigger problem instances. The bigger the problem the more one can neglect the increase in running time compared to the decrease in space consumption. It is always wise to use the `-u` flag for big problem instances.
3. The same argument holds for the improved lower bound for triple alignments. For short examples the relative increase in space and time consumption through the computation of triple alignments is high. However, this effect diminishes with bigger problem instances (e.g. Example 3 with the `PAM250` cost matrix) where again space is the limiting factor. Therefore it is advisable to use triple alignments for lower bound computation on big problem instances, instead of switching them off with `-2`.
4. The flag `-a` usually gives a small increase in running time and a small decrease in space consumption. This effect varies but in any case the memory allocation is less. We therefore recommend to use the `-a` flag.

4.2 Comparison of GSA with MSA and FMA

In order to allow a comparison we had to adapt some definitions in the MSA code. First we removed the precompiler definition `#define MINE 5` in the file `ecalc.c` which sets the value for MIN_EPS in MSA. We replaced it by an integer variable `MINE` which can be set by a new command line switch `-x value` and is initialized to 5. Then we replaced the precompiler definition `#define MAXE 50` by `#define MAXE 9999` in the file `ecalc.c`. This actually seems to be a good idea in general, because the value 50 very often prevents MSA from finding any alignment, whereas the high value very often finds an alignment. We also changed the default definition of gap costs in `main.c` from 8 to 0, since the current implementation of GSA only supports linear gap costs. Finally we changed the definition of `#define NUMBER 10` in the file `defs.h` to `#define NUMBER 12` in order to be able to compute alignments of up to 12 sequences.

It should be explicitly noted that MSA supports affine gap costs, a feature which is switched off here in order to yield the same alignments. Nevertheless MSA uses this more time and space consuming algorithm. Until GSA supports affine gap costs, there can be no final judgment about the quality of the two programs. Nevertheless we hope that our comparison illustrates the advantage of the $\mathcal{A}^*$ algorithm together with our improvements over the standard bounding techniques.

The code of the program FMA was not changed, because it also uses linear gap costs. Unfortunately FMA does not use face-bounding, so that it naturally cannot compute larger examples to optimality. The programs were invoked (at least) with the following flags in order to compute the same alignments:

```
fma -g -12 -12 -f <string-file> -c dat/dayhoff.score  (dayhoff matrix)
fma -f <string-file> -c dat/PAM250-score              (PAM250 matrix)
msa -g -b <string-file>                               (dayhoff matrix)
msa -g -b <string-file> -c pam250.dat                 (PAM250 matrix)
```

```
gsa -cpam250  <string-file>                    (dayhoff matrix)
gsa -cdayhoff <string-file>                    (PAM250 matrix)
```

We regard the computed alignment value as optimal if we compute it with sufficiently large MIN_EPS or, since this is not always possible for large examples, if all three programs yield the same value. The program FMA is invoked only once on each example, since we cannot influence its face bounding. The programs MSA and GSA are called four times. Once with their default values for MIN_EPS (5 for MSA and calculated like in Section 2 for GSA), once with the lowest MIN_EPS that still yields the optimal alignment, once with the lowest value of MIN_EPS that yields an alignment at all and once with the value of MIN_EPS set to a very large value so that no face-bounding is done. If a program uses more space than 512 MB or computes longer than one hour we mark this fact by the table entry `+`. The column "flags" contains additional flags. Note that the MIN_EPS values for GSA and MSA are not correlated, although we display the values for MIN_EPS in the same column. For a start we compare our first benchmark problem computed with the PAM250 cost matrix. GSA computed the following upper and lower bounds:

lower bound $L(s \to t)$	37028
lower bound $L_3(s \to t)$:	37318
upper bound U_0:	38014
optimal cost:	37740

The following table shows the results with the PAM250 cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	426	158	37440
GSA	-a	99	21	17	37440
GSA	-ae46	46	6.8	6.7	37440
GSA	-ae1	1	2.8	4.4	37956
GSA	-aue99999	99999	226	135	37440
MSA	-	5	3.4	2.5	37790
MSA	-x46	46	78	3.9	37440
MSA	-x1	1	1.8	1	37816
MSA	-x99999	99999	+	+	+

With the dayhoff matrix GSA computes the following bounds:

lower bound $L(s \to t)$	17751
lower bound $L_3(s \to t)$:	17907
upper bound U_0:	18253
optimal cost:	18040

The next table shows the results of the programs run with the dayhoff cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	1954	440	18040
GSA	-a	51	80	56	18040
GSA	-ae10	10	12	8.8	18040
GSA	-ae1	1	4.3	5.8	18202
GSA	-ae99999	99999	233	198	18040
MSA	-	5	81	5.5	18070
MSA	-x13	13	555	11	18040
MSA	-x1	1	37	3.6	18091
MSA	-x99999	99999	+	+	+

With both cost matrices FMA needs more time and space than GSA run with flag `-e99999`. MSA invoked with `-x99999` could not find an alignment within the specified space and time bounds. Called with the standard option GSA produces always the optimal result within 21 respectively 80 seconds. With its default values MSA solves the problem with the PAM250 cost matrix quicker, however it does not compute the optimal solution due to the small

MIN_EPS value. If one increases this value until the optimal solution is found then MSA needs considerably more time than GSA (78 sec. compared to 6.8 sec. with the PAM250 and 555 sec. compared to 12 sec. with the dayhoff matrix).

In the second example we demonstrate that GSA is able to align a lot of sequences of reasonable length in a short time to optimality. This fact makes it particulary useful for the divide-and-conquer approach of Stoye et al. GSA computed the following upper and lower bounds with the PAM250 cost matrix:

lower bound $L(s \to t)$	105136
lower bound $L_3(s \to t)$:	105670
upper bound U_0:	106782
optimal cost:	105990

The following table shows the results with the PAM250 cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	+	+	+
GSA	-a	30	5.8	6.2	105990
GSA	-ae22	22	4.9	5.3	105990
GSA	-ae1	1	5	5.1	106732
GSA	-aue99999	99999	340	142	105990
MSA	-	5	0.9	1	105990
MSA	-x1	1	0.8	1	105990
MSA	-x1	1	0.8	1	105990
MSA	-x99999	99999	+	+	+

With the dayhoff matrix GSA computes the following bounds:

lower bound $L(s \to t)$	51338
lower bound $L_3(s \to t)$:	51693
upper bound U_0:	52200
optimal cost:	51998

The next table shows the results of the programs run with the dayhoff cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	+	+	+
GSA	-au	16	+	+	+
GSA	-ae7	7	40	8.8	51988
GSA	-ae1	1	8.5	6.3	52088
GSA	-aue99999	99999	+	+	+
MSA	-	5	44	3	52008
MSA	-x10	10	+	+	+
MSA	-x1	1	16	2.8	52008
MSA	-x99999	99999	+	+	+

The optimal alignment with the PAM250 cost matrix computed by GSA is:

```
1 :  V-L-SPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGK-
2 :  M-L-TDAEKKEVTALWGKAAGHGEEYGAEALERLFQAFPTTKTYFSHFDLSHGSAQIKAHGK-
3 :  V-L-SAADKTNVKGVFSKIGGHAEEYGAETLERMFIAYPQTKTYFPHFDLSHGSAQIKAHGK-
4 :  VHL-TPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKV
5 :  VHL-SGGEKSAVTNLWGKV--NINELGGEALGRLLVVYPWTQRFFEAFGDLSSAGAVMGNPKV
6 :  VHW-TAEEKQLITGLWGKV--NVADCGAEALARLLIVYPWTQRFFASFGNLSSPTAILGNPMV
7 :  G-L-SDGEWQLVLNVWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKFKHLKSEDEMKASED-
8 :  G-L-SDGEWQLVLKVWGKVEGDLPGHGQEVLIRLFKTHPETLEKFDKFKGLKTEDEMKASAD-
9 :  MKFFAVLALCIVGAIASPLTADEASLVQSSW-KA-VSHNEVEILAAVFAAYPDIQNKFSQFA-
10:  GVL-TDVQVALVKSSFEEFNANIPKNTHRFFTLVLEIAPGAKDLFS-F-LKGSSEVPQNNPDL
11:  M-L-DQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVRPLFD-MGRQESLEQPKALAMT
```

GSA is the only program, that can compute a guaranteed optimal alignment with the `PAM250` cost matrix. If one subtracts the 120MB used for the triple alignments the space consumption

of the main algorithm is quite moderate thanks to our improved lower and upper bounds. MSA and FMA in turn need excessive space and time. In the case of the `PAM250` cost matrix MSA very quickly finds an optimal alignment, even with a MIN_EPS value of 1. This shows that the heuristic alignment is very close to the optimal alignment. However with its default `dayhoff` cost matrix MSA is not able to find an optimal alignment within the space and time bounds, whereas GSA finds an probably optimal alignment in 40 sec.

In the third example we take 5 full length globin seuqences from McClure's dataset. GSA computed the following upper and lower bounds with the PAM250 cost matrix:

lower bound $L(s \to t)$	49860
lower bound $L_3(s \to t)$:	50260
upper bound U_0:	51320
optimal cost:	50694

The following table shows the results with the PAM250 cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	+	+	+
GSA	-a	147	57	47	50694
GSA	-ae124	124	53	42	50694
GSA	-ae1	1	32	22	51194
GSA	-aue99999	99999	+	+	+
MSA	-	5	25	3.6	50756
MSA	-x90	90	+	+	+
MSA	-x1	1	17	3.0	50762
MSA	-x99999	99999	+	+	+

With the dayhoff matrix GSA computes the following bounds:

lower bound $L(s \to t)$	24012
lower bound $L_3(s \to t)$:	24194
upper bound U_0:	24575
optimal cost:	24445

The next table shows the results of the programs run with the dayhoff cost matrix:

program	flags	min eps.	time (sec)	space (MB)	alignment cost
FMA	-	-	+	+	+
GSA	-a	57	735	235	24445
GSA	-ae6	6	139	53	24445
GSA	-ae1	1	66	38	24486
GSA	-aue99999	99999	+	+	+
MSA	-	5	757	12	24447
MSA	-x10	10	2144	20	24445
MSA	-x1	1	352	5.7	24450
MSA	-x99999	99999	+	+	+

The optimal alignment with the PAM250 cost matrix computed by GSA is:

```
1 :  V--LSPAD--K-TNVKAAW--GK--V--GA-HAGEYGA-EALERMFLSFPTTKTYFPHF-D---LS
2 :  VH-LTPEE--K-SAVTALW--GK--V--NV-DEVG-GE-A-LGRLLVVYPWTQRFFESFGDLSTPD
3 :  MKFFAVLALCIVGAIASPLTADEASLVQSSWKAVSHNEVEILAAVFAAYPDIQNKFSQFAG-KDLA
4 :  GV-LTDVQ--V-ALVKSSF--EE--F--NA-NIPKNTH-RFFTLVLEIAPGAKDLFSFLKG--SSE
5 :  M--LDQQT--I-NIIKATV--PV--L--KE-HGVTITT-TFYKNLFAKHPEVRPLFDMGRQ-ESLE

1 :  --HGSAQVKGHGKKVAD-ALTNAVA-HVD-DMPNALSAL-SDL-HAHKLR-VDPVNFKLLSHCLLV
2 :  AVMGNPKVKAHGKKVLG-AFSDGLA-HLD-NLKGTFATL-SEL-HCDKLH-VDPENFRLLGNVLVC
3 :  SIKDTGAFATHATRIVS-FLSEVIALSGNTSNAAAVNSLVSKLGDDHKARGVSAAQFGEFRTALVA
4 :  VPQNNPDLQAHAGKVFKLTYEAAIQLEVN-GAVASDATL-KSLGSVHVSKGVVDAHFPVVKEAILK
5 :  QPKALAMTVLAAAQNIE-NLPAILP-AVK-KIAVK-HCQ-AGVAAAHYPI-VGQELLGAIKEVLGD

1 :  TLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR---
```

```
2 :  VLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH---
3 :  YLQANV--SWGDNVAAAWNKALDNTFAIVVPRL----
4 :  TIKEVVGDKWSEELNTAWTIAYDELAIIIKKEMKDAA
5 :  AATDDILDAWGKAYGVIADVFIQVEADLYAQAVE---
```

For the longer sequences none of the programs can guarantee optimality. GSA is the only program that can compute the "optimal" value within the space and time bounds with both cost matrices. In fact it needs only 53 sec. with the `PAM250` and 139 sec. with the `dayhoff` cost matrix, whereas MSA needs 2144 sec. with the `dayhoff` matrix.

In all our examples the strategy of GSA for computing the MIN_EPS values is always such that the optimal alignment is computed. Additionally the strategy achieved in all but one cases the optimal alignment in reasonable time so that the user can trust this heuristically computed value.

In conclusion one can say, that the $\mathcal{A}^*$ algorithm together with the proposed improvements considerably speeds up the computation for multiple sequence alignment. It certainly will do this also with affine gap costs. What needs to be shown is whether our improvements then still dominate the overhead imposed by C++ and LEDA.

5 Conclusion

In this paper we showed that the $\mathcal{A}^*$ algorithm with standard bounding techniques is superior to the well known Carillo-Lipman bound, because it excludes at least as many nodes in the dynamic programming graph from consideration. Further on we improved this algorithm in form of better lower and upper bounds. We implemented the algorithm in a C++ class using the software library LEDA. This makes the algorithm easy to read and to maintain. We show that this implementation outperforms similar programs due to our algorithmic improvements and conjecture that an ongoing implementation of GSA with affine gap costs will be a useful tool for multiple sequence alignment.

References

[CL88] H. Carrillo and David J. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, 1988.

[GKS95] S.K. Gupta, J.D. Kececioglu, and A.A. Schaeffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Comput. Biol.*, 2:459–472, 1995.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences : Computer Science and Computational Biology.* Cambridge University Press, New York, NY, first edition, 1997.

[Hor97] Paul Horton. *String algorithms and machine learning applications for computational biology.* PhD dissertation, UC Berkeley, Department of Computer Science, December 1997.

[KAL94] John D. Kececioglu, Steven Altschul, and David J. Lipman. Msa 2.1 : A program for computing multiple alignments. source codes (http://www.ibc.wustl.edu/ibc/msa.html), 1994.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley-Teubner, Chichester, first edition, 1990.

[Ler97] Martin Lermen. Multiple sequence alignment. Master's thesis, Universität des Saarlandes, Im Stadtwald, 66123 Saarbrücken, 1997.

[MN95] Kurt Mehlhorn and Stefan Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[MVF94] Marcella McClure, Taha K. Vasi, and Walter M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 4(11):571–592, 1994.

[NW70] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[SI97a] Tetsuo Shibuya and Takahiro Ikeda. Flexible multiple alignment program - version 0.34 alpha, suboptimal and parametrix analysis. Obtained by shibuya@is.s.u-tokyo.ac.jp, 1997.

[SI97b] Tetsuo Shibuya and Hiroshi Imai. New flexible approaches for multiple sequence alignment. In *Proceedings of the First Annual International Conference on Computational Molecular Biology (RECOMB97)*, pages 409–420, 1997.

[SM97] Joao Carlos Setubal and Joao Meidanis. *Introduction to computational molecular biology.* PWS Publishing Company, Boston, 1997.

[SMD97] J. Stoye, V. Moulton, and A. W. M. Dress. DCA: An efficient implementation of the divide-and-conquer approach to simultaneous multiple sequence alignment. *CABIOS*, 13(6), 1997. In press.

[Wat95] Michael S. Waterman. *Introduction to computational biology : maps, sequences, and genomes.* Chapman & Hall, London, 1995.