Making the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment More Space Efficient in Practice

Sandeep K. Gupta *
Department of Computer Science
Rice University
Houston, TX 77005–1892

John Kececioglu †
Department of Computer Science
University of Georgia
Athens, GA 30602–7404

Alejandro A. Schäffer ‡
Department of Computer Science
Rice University
Houston, TX 77005–1892

Address for correspondence: Alejandro A. Schäffer Department of Computer Science, Rice University 6100 Main Street, Houston, TX 77005–1892.
Phone: (713) 527-8101 x3813
FAX: (713) 285-5930

*Present address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; skgupta@pdos.lcs.mit.edu

†Some of the work of this author was carried out at the Department of Computer Science of the University of California at Davis; kece@cs.uga.edu

‡schaffer@cs.rice.edu

Abstract

The MSA program is one of the few existing programs that attempts to find optimal alignments of multiple protein or DNA sequences. The MSA program was written and distributed in 1989. It implements a branch-and-bound technique together with a variant of Dijkstra's shortest paths algorithm to prune the basic dynamic programming graph. We have made substantial improvements in the time and space usage of MSA. The improvements make feasible a variety of problem instances that were not feasible previously. On some runs we achieve an order of magnitude reduction in space usage and a significant multiplicative factor improvement in running time. To explain how the improvements work, it is necessary to give a much more detailed description of MSA than has been previously published. In practice, MSA rarely produces a provably optimal alignment; the reasons for lack of optimality are discussed at length in the body of the paper.

1 Introduction

Alignment of multiple DNA and protein sequences is a fundamental problem in molecular biology. A variety of combinatorial definitions of the alignment problem are used in practice. Finding an optimal multiple sequence alignment seems to take time and space exponential in the number of sequences for all definitions, and is provably hard for a few [13, 4, 18]. Nevertheless it is often useful to align a small (but larger than 2) number of sequences optimally.

One of the existing programs that attempts to construct an optimal alignment for some definition is `MSA` [5, 12]. `MSA` (version 1.0) was completed and distributed in 1989; most of the implementation of the resource intensive part was performed by the second author of this paper. The paper [12] gives a short overview of the program, but no detailed description of the implementation has ever been published. The description of the shortest paths computation in this paper is derived from notes circulated by the second author [10].

Some other programs that attempt to compute a global multiple sequence alignment include: `AMULT` [3, 2], `DFALIGN` [8], `MULTAL` [16, 17], `TULLA` [15], `CLUSTAL V` [9], and `MWT` [11]. Of these methods, `MWT` is the only one other than `MSA` that attempts to compute an optimal alignment based on some well-defined score. In practice, `MSA` rarely produces a provably optimal alignment; the reasons for lack of optimality are discussed at length at the end of Section 2. The methods other than `MSA` and `MWT` tend to use much less space because they do not claim to search for an optimal alignment. Two surveys of these programs and other issues in multiple sequence alignment are [6, 14]. In the second survey, `MSA` was not tested extensively, precisely because it exhausted the space on the computer that the authors of [14] used.

The stimulus for this paper is a successful attempt to reduce the space usage of `MSA` by significant multiplicative factors. On some runs we achieve an order of magnitude reduction in space usage and a significant multiplicative improvement in running time. The amount of improvement varies widely with the data set. With current computer technology the space usage of the original `MSA` is a more severe constraint than its time usage. The space reduction we have achieved makes many runs feasible that were not feasible with the original `MSA`. Also, many runs that were marginally feasible would previously exceed the real memory size and thrash extensively while allocating virtual memory. *Thrashing* occurs when the amount of space that a process requires approaches or exceeds the available real memory and the operating system must repeatedly *swap* memory items between the real memory and the virtual memory on disk. Many of these thrashing runs can now fit in main memory, which can lead to a huge double-digit multiplicative reduction in wall-clock time.

Our approach to improving the `MSA` program uses algorithmic techniques that are well-known in sectors of the computer science research community, but not well-known in the biology research community. The resource intensive part of `MSA` is an implementation of a complex variant of Dijkstra's single-source shortest-paths algorithm [7]. The version of multiple sequence alignment that `MSA` solves can be formulated as a single-source, single-destination shortest paths problem on a mesh-shaped dynamic programming graph. The dimension of the mesh is the number of sequences, and the number of vertices is the product of the sequence lengths.

To save space (even in the original version) the dynamic programming graph is generated "on-the-fly", so that only subgraphs are stored in memory at any given time. The essential idea in our space usage improvements is to prove nontrivial algorithmic invariants about when edges and vertices of the graph need to be created and can be safely deleted. To further improve the running time we carefully recoded a few subroutines that consumed the bulk of the running time, as demonstrated by runtime profiling of the code.

`MSA` will never be able to compete with heuristic alignment algorithms that can align dozens of sequences of triple-digit lengths, but the new version of `MSA` is significantly more powerful than the first version. To obtain the source code, contact the third author by electronic mail at (schaffer@cs.rice.edu).

The remainder of this paper is organized as follows. Section 2 defines the version of multiple sequence alignment that MSA solves and the basic mathematical method it uses. Section 3 describes in much more detail how `MSA` implements that method for substitution costs that ignore gap penalties. Section 4 explains how gap penalties are incorporated. Section 5 describes our space and time improvements. Section 6 compares the two versions of `MSA` with measurements on real data sets of protein sequences. We conclude with a short Discussion.

2 Definitions and the Carrillo-Lipman Heuristic

In this section we define the version of multiple sequence alignment solved by `MSA` and review the heuristic of Carrillo and Lipman used in `MSA`. Let $S_1, \ldots, S_K$, be the input sequences and assume that K is at least 2. Let Σ be the input alphabet; we assume that Σ does not contain the character '`-`', so that the dash can be used to denote a gap in the output. Most of the inputs to `MSA` consist of protein sequences, but the only place in which this assumption shows up is that the program contains default character substitution costs for proteins.

A multiple sequence alignment A is a rectangular character array on the alphabet $\Sigma' := \Sigma \cup \{\texttt{-}\}$ that satisfies the following conditions:

1. There are exactly K rows.
2. Ignoring the gap characters, row I is precisely the string S_I.

Each multiple sequence alignment induces a pairwise alignment $A_{i,j}$ on the pair of sequences S_i, S_j. This alignment can be obtained simply by copying rows i and j of A, with the proviso that columns in which both rows have a '`-`' are ignored.

```
--VLSPADKTNVKAAWGKVGAHAGEYGAEALE-
-VHLTPEEKSAVTALWGKVNVD--EVGGEALGR
--GLSDGEWQLVLNVWGKVEADIPGHGQEVLI-
MKFFAVLALCIVGAIASPLTADEASLVQSS---
```

Figure 1: A sample alignment of prefixes of four globin sequences

Figure 1 shows an example multiple alignment computed by `MSA` using the default parameter settings. It has four rows, one per sequence. In this case each input sequence has 30 characters, and the output alignment has 33 columns. In general, the input sequences can be of varying lengths. The four sequences in this figure are the first 30 characters of each of the four globin sequences HAHU, HBHU, MYHU, IGLOB from the globin data set in [14]. The default setting in `MSA` does not penalize gaps at the end of the sequence, which explains why the alignment it computed puts most of the gaps at the end.

In the simplest cost model there is a cost function sub : $\Sigma' \times \Sigma' \rightarrow \mathbf{N}$, such that $\mathrm{sub}(a, b)$ is the cost of substituting a b in the second sequence for an a in the first sequence; also $\mathrm{sub}(-, b)$ is the cost for columns where the first sequence has a gap and the second has a b, and $\mathrm{sub}(a, -)$ is the cost for columns where the first sequence has an a and the second has a $-$. The function sub should be symmetric. The cost of a pairwise alignment $A_{i,j}$ is

$$c(A_{i,j}) := \sum_{1 \leq k \leq w} \mathrm{sub}(A[i][k], A[j][k]),$$

where w is the width of the rectangular array A. A crucial point is that the cost of the pairwise alignment induced by A will always be at least as large as the cost of an optimal alignment of (just) the two sequences S_i, S_j. `MSA` attempts to minimize pairwise alignment costs. This is consistent with the use of Dijkstra's algorithm in `MSA`, but some other papers on multiple sequence alignment would say that higher values of sub are preferred.

By default, `MSA` supports a more sophisticated cost model where gaps that span several consecutive columns cost less than the sum of *sub* applied individually to each column. In particular, computing the substitution cost of a column depends on whether the previous column has a $-$ or not. With an optional flag, `MSA` supports a variation on gap costs where gaps at either end of the string cost nothing; gaps at either end are sometimes called *terminal gaps*. We use the notation $c(A_{i,j})$ for all three of the cost models, and use the context to say which cost model we refer to, if necessary.

The basic *sum of pairs* multiple sequence alignment problem is to minimize the pairwise sum:

$$\sum_{i<j} c(A_{i,j}).$$

MSA can solve the sum of pairs multiple sequence alignment, if the user supplies the appropriate flag. However, by default, MSA solves a variation where each pairwise alignment cost is multiplied by a weight, denoted by $\text{scale}(S_i, S_j)$, and the weights are chosen via an evolutionary tree of the input sequences. MSA uses a heuristic described in [1] to compute the pairwise weights. We have not modified the code that computes the pairwise weights because it is not space- or time-intensive.

Although, MSA can solve problems to optimality in principle, it often does not; furthermore, in most cases where it happens to arrive at an optimal solution, it does not actually explore enough of the solution space to *guarantee* that the output alignment is optimal. There are two distinct reasons for the lack of optimality in some cases, and lack of proof of optimality in others. We elaborate on the first reason now and on the second reason at the end of the section.

The first reason for lack of optimality is that MSA searches only among alignments whose cost is $\leq U$, for some value of U. The main part of MSA assumes that a finite value for U is known. If U is too small, the program dos not find any feasible solution. For this reason, it may take several runs, with increasing values of U to even get an alignment, optimal or not. The reason not to use an arbitrarily large value of U initially is that a larger U requires more space and time for program execution.

The value of U used in MSA is computed in a somewhat convoluted manner. Define a lower bound L as the weighted sum

$$L := \sum_{i<j} d(S_i, S_j) \cdot \text{scale}(S_i, S_j)$$

of costs of optimal pairwise alignments, where $w_{i,j}$ is the weight assigned to the pair of sequences S_i, S_j. This value of L is a lower bound on the cost of a multiple sequence alignment because it assumes that each pair of sequences is aligned optimally, independent of the other sequences. In MSA, the difference $U - L$ is stored in the variable DELTA, which we denote here by δ. From here, there are three ways to get to U. In each case a difference δ is obtained and U is computed as $L + \delta$. One option is for the user to specify δ.

Second, if the user does not provide δ, a preprocessor program computes a heuristic multiple sequence alignment from a star-tree of pairwise alignments in which places where all sequences agree are forced into alignment, and an incremental heuristic is used in the intervals between forced columns. This heuristic is not time- or space- intensive. The heuristic multiple alignment induces a pairwise cost; the difference between that cost and the optimal pairwise cost $c_{i,j}$ is denoted by $\epsilon_{i,j}$. Then δ is computed by:

$$\delta := \sum_{i<j} (\min(\texttt{maxepsilon}, \epsilon_{i,j}) * w_{i,j}).$$

The number `maxepsilon` is set by default to 50, but can be changed by changing one constant in the code.

The third option to get δ is that the user may suppress the preprocessor program by supplying a file with values for $\epsilon_{i,j}$, in which case these values are used in the same way as the computed ϵ values in the second option. Again, this method may yield an upper bound U that is too low.

Carrillo and Lipman [5] realized that not all alignments A need to be considered. Define $d(S_1, S_2, \ldots, S_K) := \min_A c(A)$. Let L be the sum of pairs lower bound defined as above, and let U be an upper bound on $d(S_1, \ldots, S_K)$, and let $\mathcal{A}$ be a multiple sequence alignment of minimal cost. Then

$$\begin{aligned} U - L &\geq \sum_{i<j} [c(\mathcal{A}_{i,j}) - d(S_i, S_j)] \\ &\geq c(\mathcal{A}_{p,q}) - d(S_p, S_q) \end{aligned}$$

for all distinct p and q, since for all alignments A, $c(A_{i,j}) \geq d(S_i, S_j)$. Rearranging, we derive the Carrillo and Lipman bound

$$c(\mathcal{A}_{i,j}) \leq d(S_i, S_j) + U - L. \tag{1}$$

Note that $\Omega(K^2)$ terms are thrown away; while the bound may achieve equality, for K equally dissimilar sequences, $c(\mathcal{A}_{i,j}) - d(S_i, S_j)$ will be overestimated by a factor $O(K^2)$.

In the classic dynamic programming algorithm for multiple sequence alignment, a directed acyclic graph is constructed in which source to sink paths of weight C correspond to alignments of cost C. For multiple sequence alignment, this graph is conveniently represented with vertices embedded at integer coordinates in K-space and sequences labeling the coordinate axes. A path P from the vertex $s = \langle 0, \ldots, 0 \rangle$ that ends at vertex $t = \langle n_1, \ldots, n_k \rangle$ represents an alignment of the K prefixes $S_i[1 \ldots n_i]$ for $1 \leq i \leq K$. There is a one-to-one correspondence between edges in the path and columns in the alignment. For example, in the alignment of Figure 1 the first two edges are

$$\langle 0,0,0,0 \rangle \rightarrow \langle 0,0,0,1 \rangle \rightarrow \langle 0,1,0,2 \rangle$$

For column c, if we have a real character in row i, then the cth edge in the path advances the coordinate in dimension i by 1; if we have a dash in row i, the coordinate in dimension i stays the same.

The algorithm used to compute the shortest path from s to t is a variant of Dijkstra's algorithm [7]. In Dijkstra's algorithm, as used in the next section, each vertex v has a label $v.D$ which is the current cost of a shortest path from s to v. Each edge $e = v \rightarrow w$ has a cost, denoted by $\text{cost}(e)$. When the algorithm explores an edge $v \rightarrow w$, it compares the current $w.D$ (representing a known path to w) against $v.D + \text{cost}(e)$ (representing a path to w through v). The new label $w.D$ is the lesser of the two alternatives. The algorithm always chooses v to have minimum label D among vertices with unused outgoing edges. The vertices with unexplored outgoing edges are kept in a priority queue, so that v can be found quickly.

Inequality (1) restricts consideration to those paths whose projection onto the plane for every sequence pair S_i and S_j has weight at most $d(S_i, S_j) + U - L$. In practice, `MSA` also prunes vertices by lower-bounding costs from the current vertex to the sink in the dynamic programming graph. If the minimum cost to reach v from the source plus the (lower bound on the) cost to reach the sink from v is higher than U, then v can be pruned. For each pair of sequences, S_i, S_j, `MSA` computes the standard two dimensional dynamic programming graph $D_{i,j}$ and the cost of an optimal alignment passing through each vertex. We call vertices whose cost in $D_{i,j}$ is at most $c_{i,j} + \epsilon_{i,j}$ *admissible* with respect to the pair (i, j). Returning to the dynamic programming graph in K-space, `MSA` considers only vertices that are admissible with respect to every pair. Admissibility depends only on the pairwise alignment and not the actual costs in the K-dimensional dynamic programming graph. By manually increasing the value of $\epsilon_{i,j}$, the user can improve the chances that `MSA` considers enough of the dynamic programming graph, so that the output alignment is optimal or near-optimal. However, we emphasize that it is generally impractical to set each $\epsilon_{i,j}$ to $U - L$, which is what would be needed to guarantee optimality.

3 First Implementation with no Gap Penalties

We design an algorithm incorporating inequality (1) where pairwise alignments are scored as the sum of single-symbol insertion, deletion, and substitution costs; this algorithm is later extended for an affine gap cost function. Input consists of the sequences $S_1, S_2, \ldots, S_K$ of lengths $N_1, N_2, \ldots, N_K$, and the difference $\delta := U - L$. We will write N for $\max_i N_i$.

The algorithm is naturally organized in two stages. In the first stage, $O(K^2)$ pairwise sequence comparisons are performed to determine, for each plane defined by sequence pair S_i and S_j, the set of points $\langle p, q\rangle$ on a path from $\langle 0, 0\rangle$ to $\langle N_i, N_j\rangle$ of weight at most $d(S_i, S_j) + \epsilon_{i,j}$. Let $F_{i,j}$ be the set of such points $\langle p, q\rangle$ on the face formed by S_i and S_j. At the conclusion of this stage, the lower bound $L = \sum_{i<j} d(S_i, S_j)$ and upper bound $U = L + \delta$ can be computed. The first stage is neither time- nor space-intensive, so we give it superficial treatment.

In the second stage, an alignment corresponding to a shortest source to sink path whose projection lies wholly within the $F_{i,j}$ is computed. This weighted directed acyclic graph $G = \langle V, E\rangle$ has vertex set

$$V := [0, N_1] \times [0, N_2] \times \cdots \times [0, N_K]$$

and edge set

$$E := \left\{ v \to w \parallel w - v \in \{0, 1\}^K - \{0\}^K \right\},$$

where we use the standard componentwise subtraction of vectors. For character a, define $a^1 := a$ and $a^0 := -$. Using this notation, edge $\langle v_1, \cdots, v_K\rangle \xrightarrow{e} \langle w_1, \cdots, w_K\rangle$ has cost

$$\mathrm{cost}(e) = \sum_{i<j} \mathrm{sub}\left((S_i[w_i])^{w_i - v_i}, (S_j[w_j])^{w_j - v_j}\right).$$

Let $R_{i,j}$ be the region in K-space of points $\langle p_1, p_2, \ldots, p_K\rangle$ such that $\langle p_i, p_j\rangle \in F_{i,j}$, and define $R := \bigcap_{i<j} R_{i,j}$. Then the paths of interest lie within $G \setminus R$, the subgraph of G induced on the vertex subset R.

There are some subtleties in the algorithm because the vertices and edges are generated dynamically and because of the regular mesh-coordinate structure of the dynamic programming graph. We will say that a vertex or edge "exists" when the algorithm has allocated space for it and has not freed that space. At vertex v, we find its coordinates as point p. Then for each point q such that $q - p \in \{0,1\}^K - \{0\}^K$, we want a vertex w that is a neighbor of v and an edge $v \rightarrow w$. The vertex w may already exist, because we found a path to point q previously. Therefore, we need a data structure to keep track of which vertices exist. For this purpose, we use a trie of all points, such that p is a leaf in the trie if and only if a vertex v for point p has been created. All leaves in the trie are at depth K, representing the K coordinates, one per level.

This algorithm is formalized in Figure 2. Two fundamental data structures are used. Function PRIORITY() creates an empty priority queue; INSERT(x, k, q) inserts item x with key k into priority queue q; EXTRACT(q) returns and removes the item of minimum key in priority queue q; DECREASE(x, k, q) decreases to k the key of x in q. Function TRIE() creates an empty trie; INSTALL(x, s, t) installs item x with string s into trie t; LOOKUP(s, t) returns the item with string s in trie t, or Λ if none exists. In addition, two new data types are used. Type **point** is a K-tuple of integers; a point variable p has named fields $p.1, p.2, \cdots, p.K$, and is created by $p \leftarrow$POINT$(p_1, p_2, \cdots, p_K)$. Type **vertex** has two named fields: a distance D, and a point P; statement $v \leftarrow$VERTEX(d, p) creates a vertex v with $v.D = d$ and $v.P = p$. The priority queue of vertices is ranked by distance, while the trie of vertices is constructed over the integer sequences forming points. Shortest paths are computed using the distance field; adjacent vertices are generated and accessed using the point field.

Function TRACEBACK(v) returns an alignment by tracing backward through the graph from vertex v. (By examining the distance and point fields of preceding vertices, a shortest path can be reconstructed.)

One optimization incorporated in the algorithm of Figure 2 is that edges leaving some vertices may be ignored. When a vertex v is removed from the priority queue, $v.D$ is the length of a shortest path from s to v. (It is assumed that all insertion, deletion, and substitution costs—and hence all edge weights—are positive.) A lower bound on the length of a path from v to t is $\sum_{i<j} C_r[v.P.i, v.P.j]$, whose terms are computed in Stage 1. If this quantity plus $v.D$ exceeds U, then the length of any path

algorithm MSA(**sequence** $S_1, S_2, \cdots, S_K$ **; number** δ)
    trie T **; priority queue** Q **; vertex** s, t, v, w **; point** p, q
    (Stage 1.)
    $L \leftarrow 0$
    for $I \leftarrow 1$ **to** $K-1$ **do**
        for $J \leftarrow I+1$ **to** K **do**
            Let $a_1 a_2 \cdots a_n = S_I$ and $b_1 b_2 \cdots b_m = S_J$.
            for $i \leftarrow 1$ **to** n **do**
                for $j \leftarrow 1$ **to** m **do**
                    Compute $C_{f,I,J}[i,j] := d(a_1 \cdots a_i, b_1 \cdots b_j)$ and $C_{r,I,J}[i,j] := d(a_{i+1} \cdots a_n, b_{j+1} \cdots b_m)$.
                od
            od
            for $i \leftarrow 1$ **to** n **do**
                $F_{I,J}[i] \leftarrow \{j \parallel C_{f,I,J}[i,j] + C_{r,I,J}[i,j] \leq C_{f,I,J}[n,m] + \epsilon_{i,j}\}$
            od
            $L \leftarrow L + C_{f,I,J}[n,m]$
        od
    od
    $U \leftarrow L + \delta$
    (Stage 2.)
    $s \leftarrow$ VERTEX(0, POINT$(0, 0, \cdots, 0)$) **;** $t \leftarrow$ VERTEX(∞, POINT$(N_1, N_2, \cdots, N_K)$)
    $T \leftarrow$ TRIE() **;** INSTALL$(s, s.P, T)$ **;** INSTALL$(t, t.P, T)$
    $Q \leftarrow$ PRIORITY() **;** INSERT$(s, s.D, Q)$
    while $\neg$ EMPTY(Q) **do**
        $v \leftarrow$ EXTRACT(Q) **;** $q \leftarrow v.P$
        if $v = t$ **then**
            output TRACEBACK(t) **; return**
        fi
        if $v.D + \sum_{i<j}(\mathrm{scale}(S_i, S_j) \cdot C_{r,I,J}[q.i, q.j]) \leq U$ **then**
            for all $r \leftarrow$ POINT$(r_1, r_2, \cdots, r_K)$ **s.t.** $r - q \in \{0,1\}^K - \{0\}^K$ **and for all** $i < j$, $r_j \in F_{i,j}[r_i]$ **do**
                $w \leftarrow$ LOOKUP(r, T)
                if $w = \Lambda$ **then**
                    $w \leftarrow$ VERTEX(∞, r) **;** INSERT$(w, w.D, Q)$ **;** INSTALL$(w, w.P, T)$
                fi
                if $v.D + \mathrm{cost}(q \xrightarrow{e} r) < w.D$ **then**
                    $w.D \leftarrow v.D + \mathrm{cost}(e)$ **;** DECREASE$(w, w.D, Q)$
                fi
            od
        fi
    od
    output There does not exist a multiple sequence alignment with cost at most U.
end

Figure 2: Algorithm without gap costs.

from s to t through v exceeds U, and edges leaving v do not need to be examined.

We now give some more details about the concrete implementation of the data structures. Making the realistic assumption that edge costs are integers, we can implement a discrete priority queue with buckets for the possible path lengths, the number of buckets being bounded by U. Then INSERT() and DECREASE() are constant-time operations, and all EXTRACT() operations take $O(U + |R|)$ total time, which is in turn $O(KN + |R|)$. As a minor concern, the statement VERTEX$(\infty, \ldots)$ can be implemented as VERTEX$(U + 1, \ldots)$, since U is a bound on vertex distances.

Since the trie is constructed over integer sequences, the children of a node are conveniently indexed through an array of pointers. Consider accessing the vertex in the trie associated with point $\langle p_1, p_2, \cdots, p_K \rangle$ in R. After having traversed the trie on prefix $p_1 p_2 \cdots p_{j-1}$, the next coordinate p_j must be in the set $\bigcap_{i<j} F_{i,j}[p_i]$. Let m and n be the minimum and maximum integers in this set. Then pointers to child nodes, and an encoding of the intersection, can be represented in a vector of length $O(n - m)$. These vectors tend to be short, and once allocated, never need change in length as new vertices are installed. Generation of adjacent vertices is often short-circuited by "falling off" the trie. To minimize space consumption, the P vertex field is not represented as a K-tuple of integers; $v.P$ points to the trie leaf associated with vertex v, and coordinates are recovered from trie node labels by following father links to the root.

One other implementation issue concerns the treatment of holes in the region R. Recall $F_{I,J}[i] := \{j \| C_{f,I,J}[i,j] + C_{r,I,J}[i,j] \leq d(S_I, S_J) + \epsilon_{i,j}\}$. These sets are implemented as a vector of j values, which can require $O(N)$ space for each set. Alternately, one could approximate the set in $O(1)$ space by storing the minimum and maximum integer elements, on the assumption that the sets are dense, and ignore holes. This will significantly reduce the overhead in a lexicographic enumeration, but in our experiments the vector solution appeared to be more effective. Furthermore, the space usage is dominated by the edges, and the faces take a small fraction of the space.

4 Adding Gap Penalties

In this section we explain how to modify the shortest-paths algorithm so as to handle affine gap costs in the alignment. The pseudocode in this section is a fairly accurate description of the actual code in `MSA` version 1.0.

When costs are determined character-by-character without treating multi-character gaps specially, the shortest paths in the graph representing least-cost alignments satisfy an important property, sometimes called the "principle of optimality". Specifically any least-cost path from the source s to vertex w, ending the edge $v \to w$, must start with a least-cost subpath from s to v. This justifies computing the distance from s to w, denoted $w.D$ by

$$w.D \leftarrow \min_u u.D + \mathrm{cost}(u \rightarrow w),$$

which is what the algorithm of the previous section does implicitly.

When affine gap costs are used, the principle of optimality does not apply in the above form. It may be preferable to start with a suboptimal path from s to v which initiates a lot of multi-column gaps in the alignment before the vertex corresponding to v. The cost of the edge $v \rightarrow w$ represents the sum-of-pairs substitution costs for using one character from each sequence or a gap. We use a character for sequence j if and only if the jth coordinate of the point represented by w is 1 more than the jth coordinate of the point represented by v. If the coordinates are equal, this corresponds to a gap in the alignment. Because of the affine gap cost function, we must know whether a gap character denoted by DASH is preceded by another DASH for this sequence or not. This can be determined from the preceding edge.

The way the costs are computed in `MSA 1.0` is shown in Figure 3. Suppose that $e = u \rightarrow v$ and $f = v \rightarrow w$. Let the points in K-dimensional space represented by vertices u, v, w be p, q, r respectively. We compute the cost of edge f under the assumption that it is preceded by edge e. Since we know the points p, q, we can determine for which sequences the edge e already includes a gap. This occurs precisely for those S_I such that $q[I] = p[I]$. Otherwise, $q[I] - p[I] = 1$, indicating that we do put a real character of S_I in the column of the alignment corresponding to this edge. The difference $q[I] - p[I]$ is stored in Δ_e. Similarly we compute Δ_f for the edge f. The double loop on I, J at the bottom of the cost algorithm computes the sum of pairwise contributions from the edge f. When there are no gaps, the pairwise contribution depends only on $\mathrm{sub}(S_I[r[I]], S_J[r[J]])$. When there are gaps, the GAPPENALTY term assigns different costs depending on whether this is the beginning of a new gap or the continuation of a multicharacter gap.

Figure 4 shows the modifications to the alignment algorithm for gap costs. Stage 1 does not change significantly, so we show only the new Stage 2. It is natural to modify Stage 2 so that it is an edge-based version of Dijkstra's algorithm. Instead of computing the shortest path starting at s and ending at each vertex, we compute the shortest path starting at s and ending at each edge. To make things work completely in terms of edges, a dummy source node $NULL$ is introduced and an edge from $NULL$ to s is created to start the algorithm. Now the D distance fields are associated with edges and $e.D$ is the current minimum cost of a path starting at s and ending with edge e.

Looking at the data structures, the priority queue Q now stores edges. The trie still stores vertices to allow lookups by coordinate values. EDGELIST is the list of edges. The call EDGE(v, w, d) makes an edge from vertex v to vertex w with distance label d. INCLUDE(e, E) adds edge e to EDGELIST E. FIND(q, r, E) returns the edge with tail q and head r in EDGELIST E if there is one, or returns Λ if such an edge is not on the list.

```
algorithm COST(edge f, edge e)
    point p, q, r ; integer Δ_e[K], Δ_f[K] ; character subchar[K]
    p ←e.tail.P
    q ←f.tail.P
    r ←f.head.P
    for I ←1 to K do
        Δ_e[I] ←q[I] − p[I]
        Δ_f[I] ←r[I] − q[I]
        if Δ_f[I] = 1 then
            subchar[I] ←S[I][r[I]]
        else
            subchar[I] ←DASH()
        fi
    od
    totalcost ←0
    for I ←2 to K do
        for J ←1 to I − 1 do
            if DoNotPenalizeTerminalGaps and (q[I] = 0 or q[I] = N[I] or q[J] = 0 or q[J] = N[J]) then
                totalcost ←totalcost + scale(S_J, S_I) * sub([subchar[J]][subchar[I]])
            else
                totalcost ←totalcost + scale(S_J, S_I) * (sub([subchar[J]][subchar[I]])+
                GAPPENALTY(Δ_e[I], Δ_e[J], Δ_f[I], Δ_f[J]))
            fi
        od
    od
    return totalcost
    end
```

Figure 3: Edge cost algorithm.

```
algorithm MSA(sequence S_1, S_2, ···, S_K ; number δ)
    trie T ; priority queue Q ; vertex s, t, v, w ; point p, q ; edge list E ; edge e, f
    (Stage 2.)
    s ←VERTEX(POINT(0, 0, ···, 0)) ; t ←VERTEX(POINT(N_1, N_2, ···, N_K))
    T ←TRIE() ; INSTALL(s, s.P, T) ; INSTALL(t, t.P, T)
    e ←EDGE(NULL, s, 0)
    E ←EDGELIST() ; INCLUDE(e, E)
    Q ←PRIORITY() ; INSERT(e, e.D, Q)
    while ¬ EMPTY(Q) do
        e ←EXTRACT(Q) ; v ←e.head
        q ←v.P ; p ←e.tail.P
        if v = t then
            output TRACEBACK(t) ; return
        fi
        if e.D + Σ_{i<j} scale(S_i, S_j) * C_r[q.i, q.j]) ≤ U then
            for all r ←POINT(r_1, r_2, ···, r_K) s.t. r − q ∈ {0,1}^K − {0}^K and for all i < j r_j ∈ F_{i,j}[r_i] do
                w ←LOOKUP(r, T)
                if w = Λ then
                    w ←VERTEX(r) ; INSTALL(w, w.P, T)
                fi
                f ←FIND(q, r, E)
                if f = Γ then
                    f ←EDGE(v, w, ∞) ; INCLUDE(f, E) ; INSERT(f, f.D, Q)
                fi
                if e.D + COST(f, e) < f.D then
                    f.D ←f.D + COST(f, e) ; DECREASE(f, f.D, Q)
                fi
            od
        fi
    od
    output There does not exist a multiple sequence alignment with cost at most U.
end
```

Figure 4: Modified Stage 2 for gap costs.

Whenever we extract an edge $e = u \rightarrow v$ from the priority queue, we find the points p, q in the mesh graph that the vertices represent. We then look for all points r such that the mesh graph has an edge from the point q to the point r. We either find the vertex w representing r or make a new one and call it w. Let f be the edge $v \rightarrow w$. We compute the cost of f preceded by e and see if this gives a new and/or better way to have a path ending with edge f.

5 Improvements

In this section we describe the algorithmic changes that we implemented to improve the performance of `MSA`. In planning the improvements, we used two important profiling tools that measure the runtime behavior of C programs. The first tool is the function-level profiler `gprof`, which shows for each function: how often it is called, from where it is called, and an estimate of the total time it takes over all calls. The second tool is the basic block-level profiler `tcov`, which shows for each basic block how often it is executed. Both tools can be used to predict how much time will be saved by many types of code changes. `gprof` shows how often memory is allocated, so it is good for predicting how much space will be saved by certain changes. `tcov` shows how often each conditional test succeeds or fails, so it is especially useful in rewriting small intensely-used sections of a program.

By profiling `MSA` with `gprof` we immediately learned two important facts:

1. The running time is dominated by three functions:

    - COST computes the cost of an edge e followed by an edge w
    - ADJACENT finds the next neighbor of the tail of the edge that has just been extracted from the heap
    - EDGE searches through a small list of edges for an edge between two endpoints, and if the edge is not found creates a record for a new edge and initializes it. Our measurements showed that the vast majority of the time over all calls to EDGE was spent searching for edges that already existed.

2. The bulk of the space is taken by records that store edges.

From here we naturally set out to investigate exactly when edges are needed. Not surprisingly, we also had to study when vertices are needed. Later we recoded the cost computation to save more time.

We discovered that a major problem with edges is that they were stored in lists at the incoming vertex; i.e., the edge $v \rightarrow w$ would be stored in a list at vertex w. This choice had two virtues: it allowed for easy backtracking from source to sink to report the final alignment, and it was the easiest way to check whether an edge already existed or needed to be created. However, it also has two significant problems: it requires many list searches (which do not take O(1) time) to find records for existing edges, and it precludes several opportunities for deleting edges and vertices. To see the first problem consider what happens each time v is extracted from the priority queue. We will want to find all the edges $v \to w$, but each such edge is in an arbitrary location on a list at w, and we need to search the list to find it. The second problem is more subtle, and we will explain it later at length. We set out to correct the two major problems, while finding alternate ways to achieve the virtues.

If we instead store the list of outgoing edges $v \to w$ at v, we can scan the list in $O(1)$ time per edge and we never need to regenerate an edge. We changed the implementation, so that edges are stored in a list at the tail vertex. If the list of outgoing edges is NULL, then the vertex has no outgoing edges.

To provide for easy backtracking, we added a BACKTRACK field to each edge, which is its preceding edge in some optimal path. This makes backtracking very simple and fast, although it costs one extra pointer per edge. Each edge also has a reference count, which keeps track of how many BCKTRACK fields point to it. As we shall see, this reference count is very helpful in detecting when edges and vertices are no longer needed.

With these changes, we can now sketch the proofs of four facts about the creation of vertices and edges. A complete proof would require a line-by-line analysis of the actual code, which would horribly obfuscate the key ideas.

Fact 5.1 Once an edge e is extracted from the priority queue, $e.D$ will never again decrease.

Proof: All edge costs are non-negative. After the first edge is extracted from the priority queue, whenever an edge h is reinserted into the queue, it must have a preceding edge g such that $g.D \leq h.D$. When $e = w \to x$ is extracted from the queue, its D field is no bigger than any other D field. The only way $e.D$ could decrease is if we later extract an edge $f = v \to w$ such that f followed by e provides a better path to get to x. Furthermore, we would have to have $f.D < e.D$. However, f cannot be in the queue when e is extracted because e has a minimum D field, and an edge with a lower D field cannot be inserted into the queue later. ∎

Fact 5.2 It is correct to delete (and never recreate) an edge $e = v \to w \neq t$ that has already been extracted from the priority queue for either of the following two reasons:

1. w has no outgoing edges (can be detected in $O(1)$ time by empty outgoing list);
2. all edges of the form $w \to x$ have an edge other than e as backtrack edge (detected in O(1) time by reference count on e).

Proof: Once e is extracted from the queue, all paths that include e followed by an outgoing edge from w are considered. By the previous fact, the cost of such a path can never again decrease. If all outgoing edges of the form $w \rightarrow x$ are eliminated (reason 1), the minimum cost path to x cannot include e.

Any edges that point to e as backtrack edge must be of the form $w \rightarrow x$. The BACKTRACK pointer can be made to point to e only in the phase where e is just extracted from the heap and its succeeding edges are considered. After this time, no new BACKTRACK pointers can be assigned to e. Hence, once we reach a stage where no BACKTRACK pointers point to e (reason 2), the edge e cannot occur in an optimal path and can be deleted. ▮

Fact 5.3 If v has no outgoing edges, then all edges into v can be deleted.

Proof: A key point is that when we first extract an edge whose head is v all outgoing edges from v are created. If all edges coming out of v have been deleted, there is no way to continue a path that gets to v, so no edge coming into v can be in an optimal path. Thus the incoming edges can be deleted. ▮

The first test in the previous Fact is extremely useful in deleting edges, and helps explain why it is a good idea to store the edges at their tail.

Fact 5.4 If vertex v has no more outgoing edges or incoming edges after some edge outgoing from v is extracted, then v can never occur in a optimal path. In this case, v can be deleted, and need not be recreated.

Proof: If v is deleted, this is because it either has no incoming edges or no outgoing edges remaining, after having some. This must be because all the edges have themselves been deleted. By Facts 5.2 and 5.3, these edges cannot occur on an optimal path and do not need to be recreated. Hence v cannot occur on an optimal path and need not be recreated. ▮

The tests in the preceding facts allow us to aggressively free space used by edges and vertices. It is important to combine the tests for the two kinds of objects, even though most of the space is taken up by edges. This is because the deletion of a vertex permits the deletion of all its remaining outgoing or incoming edges, causing a cascade of deletions.

Next we turn to some improvements that reduce the time spent computing COST. The first improvement we made was to inline COST by moving its code inside the MSA function that has the basic loop for Dijkstra's algorithm. An obvious, but minor advantage of inlining is that the time needed for the actual function call is saved.

A much more significant advantage of inlining COST is that we can use invariants that apply across consecutive calls. Recalling the main loop from the previous section, suppose e has just been extracted from the queue. Let the endpoints of

e have coordinate values p and q. We then find each point r that can follow q and call COST with points p, q, r as arguments. The key points are that there will usually be multiple values of r for one p, q pair, but much of the COST computation depends only on p and q. After inlining COST, we identified those pieces of the cost computation that depended only on p, q and pulled them out of the loop on r.

The second collection of improvements involved the fundamental comparison of D labels in Dijkstra's algorithm. Recall that if we just extracted $e = p \to q$ (using coordinates) from the heap and $f = q \to r$ is a succeeding edge, we compare whether $e.D + \text{cost}(e, f) < f.D$. Our experiments with `tcov` made it clear that the test fails very often.

Our experiments suggested that we should see if the test would fail before all of the costs in cost are calculated. That is, it may be the case that $e.D + extra \geq f.D$, where $extra < \text{cost}(e, f)$. As a start, we tried with $extra = 0$, avoiding the cost computations entirely. Surprisingly this showed a measurable speedup. Then we noticed that the COST computation now inlined had a double loop on i, and j, where each loop iteration adds to the cost. A natural place to put a more refined partial test is after each iteration of the outer loop on i, using $extra$ as the pairwise sum of costs for all $j < i$.

Finally, we noticed from a `tcov` profile that there is a different way to arrange the double loop on i and j, which might allow for more early exits with a sufficiently large vale of $extra$. In the original formulation the first iteration on i did 1 value of j, the second iteration on i did 2 values of j and so on, until the last iteration did $K - 1$ values of j. We reversed the loop on i, so that the first iteration did $K - 1$ values, the second $K - 2$ and so on.

6 Results

In this section, we compare the performance of `MSA` 1.0 with our improved version on real data sets. We ran all the tests on a Sun SparcStation 10 with 128Mbytes of RAM. We used the `cc` C compiler with the -O flag for optimization. To measure time we took the sum of the user and system times reported by the `time` command. To measure space usage we ran the `top` command while `MSA` was running and noted the peak amount of space usage that it reported.

We used data sets extracted from the data sets in [14]. That paper uses 4 data sets consisting of 12 globin sequences, 12 kinase sequences, 12 protease sequences, and 12 RH sequences respectively. We extracted subsets of these 12-element sets that would give rise to `MSA` runs of moderate time and space usage. This enables us to measure the improvement in resource usage without affecting the system too much. We did do two runs in which `MSA` 1.0 far exceeded the 128Mbytes of RAM (by using virtual memory), causing thrashing; in one of those two cases we had to stop the run to permit other users to get fair access to the computer.

In all runs, we used `MSA` with the default parameter settings, except that we

# Data Set	Old Time	New Time	Old Space	New Space
Globins A (7)	265s	135s	9.1Mbytes	4.8Mbytes
Globins B (10)	385s	130s	11.3Mbytes	6.1Mbytes
Proteases A (5)	79s	37s	5.7Mbytes	3.0Mbytes
Proteases B (4)	25min	9min	81Mbytes	5.8Mbytes
Kinases A (5)	37min	8min	67Mbytes	6.8Mbytes
Kinases B (6)	stopped	118m	$>$ 300Mbytes	25Mbytes
Kinases C (4)	530s	210s	35Mbytes	4.7Mbytes
RH A (4)	394m	68m	380Mbytes	31Mbytes

Table 1: Comparison of resource usage for two versions of `MSA`. Number of sequences in each data set is in parentheses.

made δ large enough for `MSA` to find some alignment. Thus the alignments produced are not provably optimal.

Not being biologists, we refrain from doing any serious qualitative analysis of the alignments produced by `MSA`. McClure, Vasi, and Fitch proposed a scoring scheme based on the ability to properly align important small regions of the sequences called *motifs*. To define motifs they start with some predetermined alignment and designate a few special columns; generally, the characters from the different sequences in the special columns are identical or show little variation. The intuition is that since these places in the sequence change relatively little from sequence to sequence, those places are essential to the biological function of the protein sequences. An alignment matches the motif if we can find columns that look exactly like the designated columns in the predetermined alignment. We checked the alignments produced by `MSA` for all the motifs used in [14]. `MSA` essentially found all the motifs correctly, except that in two cases one sequence was shifted slightly in the region of one motif.

The data sets we used, expressed in the notation of [14] are:

- Globins A: HUMA (human hemoglobin, α chain), HBHU (human hemoglobin β chain), MYHU (human myoglobin), IGLOB (insect hemoglobin from *Chironomus thummi*), GPUGNI (nonlegume hemoglobin from swamp oak), GGZLB (bacteria, *Viteoscilla sp.*), GPYL (legume, yellow lupine).

- Globins B: HAOR (duckbill platypus, α-chain hemoglobin), HADK (duck, α-chain hemoglobin), HBOR (duckbill platypus, β-chain hemoglobin), HBDK (duck, β-chain hemoglobin), MYHU, MYOR (duckbill platypus, myoglobin), IGLOB, GPYL, GPUGNI, GGZLB.

- Proteases A: HIV-I (human immunodeficiency virus, Type I), SRV-I (simian retrovirus, type I), MoMLV (Moloney murine leukemia virus), 17.6 (Retrotransposon from *Drosophila melanogaster*), PEPH (human pepsin sequence).

- Proteases B: PEPH, MoMLV, CaMV (cauliflower mosaic virus), Copia (retrotransposon from *Drosophila melanogaster*).

- Kinases A: CD28 (*S. cerevisiae*), MLCK (rat skeletal muscle), PSKH (hela cell), CAPK (bovine cardiac muscle), WEE1 (Dual specificity kinase from *S. pombe*).

- Kinases B: CD28, MLCK, PSKH, CAPK, WEE1, CSRC (chicken oncogenic protein).

- Kinases C: CD28, CAPK, WEE1, PDGMR (mouse, PDGF receptor).

- RH A: HTLV-II (human T-cell leukemia virus), RSV (Rous sarcoma virus), MoMLV (Moloney murine leukemia virus), 17.6 (retrotransposon from *Drosophila melanogaster)*, HBV (human hepatitis B virus, ayw strain).

Table 1 shows the relative time and space usage of `MSA` 1.0 (old) and `MSA` 2.0 (new) on these data sets. In the case of Kinases C, we had to stop the old `MSA` run after it had run overnight and used more than 300Mbytes of memory. In the case of RH A, the old `MSA` run thrashed extensively, and therefore, the actual wall-clock time from start to finish was 44 hours (much larger than the cpu time usage reported in the table) compared to approximately 1 hour for the new version. We note that the running time seems to depend more on how close the sequences are genetically, than on how many sequences there are. For example, Proteases B has only 4 sequences, but takes much longer than Proteases A, which has 5 sequences.

7 Discussion

We have made improvements in the space and time usage of the `MSA` program for multiple sequence alignment. Since memory is finite on any computer, our improvements make feasible many problems that were not feasible with the first version of `MSA`. The changes to the code involved what would generally be considered low-level implementation decisions. Our work serves as yet another example of the fact that in programs that allocate a lot of memory, it pays to be careful in looking for opportunities to free memory. Also, in programs that have a tight inner loop that takes most of the time, it pays to try a variety of methods of coding that inner loop.

Acknowledgments

The first and third authors thank Dr. Sandhya Dwarkadas for useful discussions about MSA and Prof. Willy Zwaenepoel for his encouragement. The second author thanks Prof. Marcella McClure for providing her data sets from [14]. Research of the first author was supported by grants from the National Science Foundation and the Texas Advanced Technology Program. Research of the second author was supported by a Department of Energy Human Genome Distinguished Postdoctoral

Fellowship. Research of the third author was supported by a contract from IBM and a grant from the National Institutes of Health.

References

[1] S. F. Altschul, R. J. Carroll, and D. J. Lipman. Weights for data related by a tree. *J. Mol. Biol.*, 207:647–653, 1989.

[2] G. J. Barton and M. J. E. Sternberg. Evaluation and improvements in the automatic alignment of protein sequences. *J. Mol. Biol.*, 198:327–337, 1987.

[3] G. J. Barton and M. J. E. Sternberg. A strategy for the rapid multiple alignment of protein sequences. *Protein Engineering*, 1:89–94, 1987.

[4] H. Bodlaender, R. G. Downey, M. R. Fellows, and H. T. Wareham. The parameterized complexity of sequence alignment and consensus. In *Proc. of the 5th Symp. on Combinatorial Pattern Matching, Lecture Notes Comp. Sci. 807*, pages 15–30, 1994.

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

[6] S. C. Chan, A. K. C. Wong, and D. K. Y. Chiu. A survey of multiple sequence comparison methods. *Bull. Math. Biol.*, 54:563–598, 1992.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] D. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:351–360, 1987.

[9] D. G. Higgins, A. J. Bleasby, and R. Fuchs. Clustal V: improved software for multiple sequence alignment. *CABIOS*, 8:189–191, 1992.

[10] J. Kececioglu. Notes on an approach of carrillo and lipman to minimum sum of pairs multiple sequence alignment. unpublished notes, 1989.

[11] J. Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Proc. of the 4th Symp. on Combinatorial Pattern Matching, Lecture Notes Comp. Sci. 684*, pages 106–119, 1993.

[12] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proc. Natl. Acad. Sci. USA.*, 86:4412–4415, 1989.

[13] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[14] M. A. McClure, T. K. Vasi, and W. M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 11:571–592, 1994.

[15] S. Subbiah and S. C. Harrison. A method for multiple sequence alignment with gaps. *J. Mol. Biol.*, 209:539–548, 1989.

[16] W. R. Taylor. Multiple sequence alignment by a pairwise algorithm. *CABIOS*, 3:81–87, 1987.

[17] W. R. Taylor. A felxible mathod to align large numbers of biological sequences. *Journal of Molecular Evolution*, 28:161–169, 1988.

[18] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. 1993. To appear in *J. Computational Biol.*.