Sequence Comparison with Mixed Convex and Concave Costs

David Eppstein

Computer Science Department
Columbia University
New York, NY 10027

February 20, 1989

Running Head: "Sequence Comparison with Mixed Costs"

Abstract: Recently a number of algorithms have been developed for solving the minimum-weight edit sequence problem with non-linear costs for multiple insertions and deletions. We extend these algorithms to cost functions that are neither convex nor concave, but a mixture of both. We also apply this technique to related dynamic programming algorithms.

Introduction

Recently a number of algorithms have been developed for solving the minimum-weight edit sequence problem with non-linear costs for multiple insertions and deletions. We define the *modified edit distance* problem as follows. Given two strings over alphabet Σ, $x = x_1 \cdots x_m$ and $y = y_1 \cdots y_n$, the *edit distance* of x and y is the minimal cost of an edit sequence that changes x into y. This sequence contains operations deleting characters from x, inserting characters into y, and substituting characters in x for different characters in y. Each operation has an associated cost, and the cost of a sequence is the total cost of all its operations. Notice that a sequence of consecutive deletes corresponds to a substring of x that is missing from y; we call such a deleted substring a *gap* in x. Similarly we call a sequence of consecutive inserts a gap in y. In the most well-known version of the edit distance problem, the cost of a gap is proportional to the number of symbols deleted from (or inserted to) it, so we can just as well consider each gap to be composed of single character deletions or insertions. For this case the minimum cost edit distance can be computed by a well known dynamic program in time $O(mn)$.

In many applications we would like the cost of such a gap to be nonlinear. In particular the cost of deleting $x_{l+1} \cdots x_k$ might be taken to be

$$w(l, k) = f_1(x_l, x_{l+1}) + f_2(x_k, x_{k+1}) + g(k - l). \tag{1}$$

The cost consists of charges for breaking the sequence at x_{l+1} and x_k, plus an additional cost that depends on the length of the gap. The *modified edit distance* is defined to be the minimum cost of an edit sequence which changes x into y, where the costs of gaps in x are as in equation 1, and similarly the costs of gaps in y are derived from an analogous weight function w'. This sequence alignment problem arises in the context of sequence comparison in molecular biology [12], geology [13], and in speech recognition [11].

To compute the modified edit distance, we consider a dynamic programming equation of the form

$$C[i, j] = \min\{C[i - 1, j - 1] + s(x_i, y_j), F[i, j], G[i, j]\} \tag{2}$$

$$F[i, j] = \min_{0 \le \ell < j} C[i, \ell] + w(\ell, j) \tag{3}$$

$$G[i, j] = \min_{0 \le \ell < i} C[\ell, j] + w'(\ell, i) \tag{4}$$

with initial conditions $C[i, 0] = w'(0, i)$ for $1 \le i \le m$ and $C[0, j] = w(0, j)$ for $1 \le j \le n$. Thus $C[i, j]$ computes the minimum cost of an edit sequence between initial substrings of the two input strings, with substring lengths i and j. Such an edit sequence can either end in a substitution or exact match (the $C[i - 1, j - 1]$ term), an insertion ($F[i, j]$), or a deletion ($G[i, j]$). There are two gap cost functions, w and w', for inserts and deletes respectively.

The obvious dynamic program to solve this recurrence takes time $O(mn \cdot \max(m,n))$. However note that each row of F, and each column of G, can be solved as a separate case of the following recurrence:

$$E[j] = \min_{0 \le i < j} D[i] + w(i,j). \tag{5}$$

The function $w(i,j)$ here may be either the original function w for recurrence 3, or w' for recurrence 4. The values of D are here taken from the appropriate positions of C, and the computed values of E give the values in F or G; for instance for row r of recurrence 3 we would have $D[i] = C[r,i]$ and $F[r,j] = E[j]$. In general we may allow D to be any values easily computed once the corresponding values of E are known. Also, the initial value of $D[0]$ is known a priori.

In fact, for cost functions satisfying equation 1, we may simplify recurrence 5 to

$$E[j] = \min_{0 \le i < j} D[i] + g(j-i). \tag{6}$$

For example, if we are computing row r of recurrence 3, then in this simplified recurrence $D[i] = C[r,i] + f_1(x_i, x_{i+1})$, and $F[r,j] = E[j] + f_2(x_j, x_{j+1})$. In other words, we have abstracted the portions of $w(i,j)$ depending only on the endpoints i and j into arrays D and E, and out of the cost function $w(i,j)$ which now is simply a function $g(j-i)$ of the length of the gap.

The obvious dynamic programming algorithm for recurrence 5 (and recurrence 6) takes time $O(n^2)$; if we speed up this computation we will achieve a corresponding speedup in the computation of the modified edit distance.

Two dual cases of recurrence 5 have previously been studied. In the *concave* case, the gap length cost function w satisfies the *quadrangle inequality*:

$$w(i,j) + w(i',j') \le w(i',j) + w(i,j'), \tag{7}$$

whenever $i \le i' \le j \le j'$. In the *convex* case, the weight function satisfies the inverse quadrangle inequality, found by replacing $\le$ by $\ge$ in equation 7. In recurrence 6, cost function g is concave or convex in the usual sense exactly when $w(i,j) = g(j-i)$ is concave or convex with the above definition.

For both the convex and the concave cases, good algorithms have recently been developed. Hirschberg and Larmore [6] assumed a restricted quadrangle inequality with $i \le i' < j \le j'$ in inequality 7 that does not imply the inverse triangle inequality. They solved the "least weight subsequence" problem, with $D[j] = E[j]$, in time $O(n \log n)$ and in some special cases in linear time. They used this result to derive improved algorithms for several problems. Their main application is an $O(n \log n)$ algorithm for breaking a paragraph into lines with a concave penalty function. This problem had been considered by Knuth and Plass [8] with general penalty functions. Galil and Giancarlo [5] discovered algorithms for both the convex and concave cases which take time $O(n \log n)$, or linear time for some special cases. Miller and Myers [10] independently discovered a similar algorithm for the convex case. Aggarwal et al. [1] had previously given an algorithm which solves an offline version of the concave case, in which D does not depend on E, in time $O(n)$; Wilber [15] extended this work to an ingenious $O(n)$ algorithm for the online concave case; however as we shall see in the next section Wilber's algorithm has shortcomings that make it inapplicable

to the sequence alignment problem. Klawe and Kleitman [7] extended the algorithm of Aggarwal et al. to solve the convex case in time $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

It is not possible, without further assumption, to solve recurrence 5 more efficiently than the obvious $O(n^2)$ algorithm, because we must look at each of the possible values of $w(i,j)$. However, this argument does not apply to recurrence 6, because $g(j-i)$ only has linearly many values to be examined. Therefore we would like to speed up the computation of recurrence 6, without assuming anything about the convexity or concavity of g. It is not now known whether such a solution is possible. In this paper we provide a partial solution, by broadening the class of cost functions g for which an efficient solution is possible. In particular we allow g to be *piecewise concave and convex*, with s alternations. More precisely, we assume that we can find indices $c_1, c_2, \ldots c_{s-1}$ such that g is concave in the range $[1, c_1]$, convex in the range $[c_1, c_2]$, and so on. For such functions we solve recurrence 5 in time $O(ns\,\alpha(n/s))$, and therefore we also solve the modified edit sequence problem in time $O(n^2 s\,\alpha(n/s))$. Note that these times are never worse than the times of $O(n^2)$ and $O(n^3)$ for the naive dynamic programming solutions; when s is small our times will be much better than the naive times. Our algorithms use as subroutines the previous solutions to the concave and convex cases of recurrence 5; if these solutions are improved it is likely that our algorithms would also be sped up.

Interleaving Concave Computations

For the algorithms in this paper, as for the reduction given in the introduction from sequence alignment to recurrence 5, we interleave the solutions of a number of separate convex or concave copies of recurrence 5. Because of this, we require an additional property of any solutions we use: each value $E[j]$ must be known before the computation of $E[j+1]$ begins. Instead, Wilber's $O(n)$ time algorithm for the concave case of recurrence 5 guesses a block of values of $E[j]$ at once, then computes the corresponding values of $D[j]$ and verifies from them that the guessed values were correct. If they were incorrect, they and the values of $D[j]$ need to be recomputed, and the work done computing and verifying the incorrect values can be amortized against progress made. But if Wilber's algorithm is interleaved with other computations, and an incorrect guess is made, those other computations based on the incorrect guess will also need to be redone, and this extra work can no longer be amortized away. Therefore, we now present a modification to Wilber's algorithm that allows us to use it in interleaved computations.

First let us sketch Wilber's algorithm as he originally presented it. The algorithm depends on the following three facts. First, let $i_1 < i_2 < j_1 < j_2$, and let $P[j] = \min_{i_1 \le i \le i_2} D[i] + w(i,j)$ for $j_1 \le j \le j_2$ and for some concave function w. Then the values of $P[j]$ can be computed in time $O((i_2 - i_1) + (j_2 - j_1))$, using the algorithm of Aggarwal et al. [1]. Second, in recurrence 5, let $i(j)$ be the value of i such that $D[i] + w(i,j)$ supplies the minimum value for $E[j]$ and again let w be concave. Then for $j' > j$, $i(j') \ge i(j)$. And third, if we extend w to be equal to $+\infty$ for (i,j) with $i \ge j$, it remains concave.

Wilber's algorithm proceeds as follows. Assume that we know the values of $D[j]$, $E[j]$, and $i(j)$ for $j \le k$. Let $D[j] = d(E[j])$ be the function taking values of $E[j]$ computed in the recurrence to the corresponding values of $D[j]$ used for later values of the recurrence. Let $p = \min(2k - i(k) + 1, n)$. Define a *stage* to be the following sequence of steps, which Wilber repeats until all values are

computed:

(1) Compute $P[j] = \min_{i(k) \le i \le k} D[i] + w(i,j)$ for $k < j \le p$ using the algorithm of Aggarwal et al. $P[j]$ may be thought of as a guess at the eventual value of $E[j]$.

(2) Compute $Q[j] = d(P[j])$ for $k < j \le p$. I.e. we perform the computation that would be used to compute the values of $D[j]$, if $E[j]$ were in fact equal to $P[j]$.

(3) For each j with $k < j \le p$, compute $R[j] = \min_{k<i<j} Q[i] + w(i,j)$ using the algorithm of Aggarwal et al. $R[j]$ substitues the values of $Q[j]$ into the recurrence to verify that the guessed values of $P[j]$ were correct.

(4) Let h be the least j such that $R[j] < P[j]$, or p if no such index exists. Then $E[j] = P[j]$ for $k < j \le h$. If $h = p$, we know that all guesses were correct. Otherwise, we know that they were correct only through h; then $E[h+1] = R[h+1]$ and $i(h+1) > k$. In either case, start a new stage with k updated to reflect the newly verified values of $E[j]$ and $D[j]$.

A proof that the above algorithm in fact correctly computes recurrence 5 is given in Wilber's paper. We now give the time analysis for this algorithm; a similar analysis will be needed for our modification to the algorithm. The total time for each stage is $O((p-k)+(k-i(k)) = O(k-i(k))$. If $h = n$ we are done, and this final stage will have taken time $O(n)$. If $h = p \neq n$ then we will have computed $p - k = 2k - i(k) + 1 - k = k - i(k) + 1$ new values, so the time taken is matched by the increase in k. And if $h \neq p$, then $i(h+1) > k$ and $i(h+1) - i(k) > k - i(k)$, so the time taken is matched by the increase in $i(k)$. Neither k nor $i(k)$ decrease, and both are at most n, so the algorithm takes linear time.

However as we have seen this may not hold when we interleave its execution with other computation. In particular, the analysis above depends on step 2 of each stage, the computation of $Q[j] = d(P[j])$, taking only constant time for each value computed; but if we interleave several computations this step may take much longer. The problem is that $d(x)$ may not be a simple function depending only on x, but instead it may use the value of x as part of one of the other interleaved dynamic programs; and if we supply $P[j]$ as the value of x instead of the correct value of $E[j]$, this may cause incorrect work to be done in the interleaved programs. This incorrect computation will need to be undone, if the value of $P[j]$ turns out not to equal $E[j]$, and therefore the time taken to perform it can not be amortized against the total correct work performed. Instead we now describe a way of performing each stage in such a way that we only need to compute $d(E[j])$, for actual values of $E[j]$.

We introduce a new variable, c, corresponding to the role of $i(k)$ in Wilber's algorithm, and an array $A[j]$ which stores the already-computed "influence" of $D[i]$, for $i < c$, on future values of $E[j]$. That is, for all j from 1 to n,

$$A[j] = \min_{0 \le i < c} D[i] + w(i,j). \tag{8}$$

Actually, equation 8 will not hold as written above; instead we guarantee the slightly weaker condition that, if index i supplies the minimum of $D[i] + w(i,j)$ in the computation of $E[j]$, then either $i \ge c$ or $E[j] = A[j]$.

Initially $c = 0$ and all values of A are $+\infty$; clearly equation 8 holds for these initial conditions. As in Wilber's algorithm, let k be the greatest index such that $D[k]$ is known; initially $k = 0$. Finally let $p = 2k - c + 1$; c is always at most k so $p > k$. We proceed as follows.

(1) Compute $P[j] = \min(A[j], \min_{c \le i \le k} D[i] + w(i,j))$ for $k < j \le p$ using the algorithm of Aggarwal et al. As in Wilber's algorithm, we compute here our guess at the values of $E[j]$. Wilber's analysis applies here to show that the algorithm of Aggarwal et al. can be used, taking time $O((k-c)+(p-k))$.

(2) For each i with $k < i < p$, compute

$$B[i] = \max_{i<j\le p} P[j] - w(i,j)$$

using the algorithm of Aggarwal et al. Here we differ from Wilber's algorithm; instead of plugging our guesses into the function $d(x)$, we compute the bounds $B[i]$ directly from the guesses.

(3) While $k < p$, increase k by 1, let $E[k] = P[k]$, and compute $D[k] = d(E[k])$. If $k = p$, start the next stage at step 1. If not and $D[k] < B[k]$, stop and go to step 4. Otherwise, continue the loop in this step.

(4) We have found k to be the least index with $D[k] < B[k]$. For $k < j \le p$, let $A[j] = P[j]$. Set $c = k$, and start a new stage at step 1.

The algorithm can be visualized as in figure 1. The figure depicts a matrix, with columns numbered by j and rows numbered by i. The value at position (i,j) of the matrix is $D[i] + w(i,j)$. Positions below the diagonal are not used by the algorithm, and no value is defined there. Then the goal of the computation is to compute the minimum value in each column. As in Wilber's algorithm, rows are indexed starting from 0 but column numbers can start from 1, since $D[0]$ is defined and used in the minimization but $E[0]$ is not defined. The values in any row of a matrix are not known until the minimum in the corresponding column has been computed.

At each stage, the minima in all columns up to and including column k have been computed, and so the values in all rows up to k are computable. The contribution of the values in rows above (but not including) row c to the minimization for each column j has been computed into $A[j]$. Step 1 extends this computation of the contribution to include area (1) of the figure, i.e. rows c through k and columns $k+1$ through p. The remaining steps test the values so computed, to see whether they are the actual minima in their columns. If so, k can be advanced to p. Otherwise, one of the columns in the range from $k+1$ through p has a minimum in a row from k to p, and by concavity none of the values in area (2) of the figure will be the minimum in their columns. So in this case, we have computed the influence between rows c and k, and we can advance c.

More formally, we have the following lemmas.

Lemma 1. If, in the computation of a stage, for some i it is the case that $B[i] \le D[i]$, and assuming the values of D computed in all previous stages were correct, then for all j, $i < j \le p$, $D[i] + w(i,j) \ge P[j]$.

Proof: $P[j] - w(i,j) \le B[i]$ by the computation of B. So if $B[i] \le D[i]$, then clearly the desired inequality holds. •

Lemma 2. If, in the computation of a stage, we encounter a row i with $B[i] > D[i]$, and assuming the values of D computed in all previous stages were correct, then there exists a column j with $i < j \le p$, such that $D[i] + w(i,j) < P[j]$.

Proof: Let j be the column supplying the maximum value of $B[i]$, i.e. $B[i] = P[j] - w(i,j)$. Then $P[j] > D[i] + w(i,j)$. •

Lemma 3. For any j, $A[j] \geq \min_{0 \leq i < j} D[i] + w(i,j)$.

Proof: $A[j]$ is always taken to be a minimum over some such terms, so it can never be smaller than the minimum over all such terms. •

Next we show that $A[j]$ encodes the minimization over rows above row c, so the total minimum is the better of $A[j]$ and the minimimum over later rows.

Lemma 4. Each stage computes the correct values of D and E. Further, after the computation of a given stage, for each index j,

$$\min_{0 \leq i < j} D[i] + w(i,j) = \min(A[j], \min_{c \leq i < j} D[i] + w(i,j)), \tag{9}$$

Proof: We prove the lemma by an induction on the number of stages; thus we can assume that it held prior to the start of the stage. By k and c here we mean the values held at the end of the stage; let k' denote the value held by k at the start of the stage, and similarly let c' denote the value of c at the start of the stage.

We first prove the assertion that E and D are computed correctly. In a given stage, we compute these values for indices j with $k' < j \leq k$. In particular, $E[j] = P[j]$ and $D[j] = d(E[j])$ for those indices. Recall that $P[j]$ was computed as

$$P[j] = \min(A[j], \min_{c \leq i \leq k'} D[i] + w(i,j)).$$

Further, for $i < k$, $D[i] \geq B[i]$, or else we would have stopped the loop in step 3 earlier. Therefore by lemma 1, for each row i with $k' < i < j$, $P[j] \leq D[i] + w(i,j)$, so these additional rows can not affect the minimization for $E[j]$, and by the induction hypothesis of equation 9 $E[j]$ is in fact computed correctly.

Now we show that equation 9 also holds. Clearly if the stage terminates with $k = p$, it remains true, because c and A remain unchanged. Otherwise, $c = k$ is the least row such that $B[c] > D[c]$. By lemma 2, there exists a column j with $c < j \leq p$, such that

$$D[c] + w(c,j) < P[j] \leq \min_{0 \leq i \leq c'} D[i] + w(i,j).$$

By lemma 1, for $c' < i < c$, $P[j] \leq D[i] + w(i,j)$, so

$$D[c] + w(c,j) < \min_{0 \leq i < c} D[i] + w(i,j).$$

But then by concavity, for every $j' > j \geq p$,

$$D[c] + w(c,j') < \min_{0 \leq i < c} D[i] + w(i,j').$$

So

$$\begin{aligned} \min_{0 \leq i < j'} D[i] + w(i,j') &= \min_{c \leq i < j'} D[i] + w(i,j') \\ &= \min(A[j'], \min_{c \leq i < j'} D[i] + w(i,j')), \end{aligned}$$

where the last part of the above equation holds because of lemma 3. For $k < j' \leq p$, the values of $D[i] + w(i, j')$ for $c' \leq i \leq k'$ have already been computed in step 1 and incorporated into $A[j']$ in step 4. And since for each $i < c$, $D[i] \geq B[i]$, we know by lemma 1 that $D[i]+w(i, j') \geq P[j'] = A[j']$ so these rows can not affect the minimum. Therefore the equation is true for all columns. •

It remains to show that the bounds B computed in step 2 can be computed using the algorithm of Aggarwal et al. Recall that B is defined by the recurrence $B[i] = \max_{i<j\leq p} P[j] - w(i, j)$. To hide the dependence of j on i, define $f(i, j)$ to be $P[j] - w(i, j)$ if $i < j$, or $-\infty$ otherwise. Then $B[i] = \max_{k+1<j\leq p} f(i, j)$. The problem is to find the maxima on the rows of the matrix implicitly determined by the function $f(i, j)$. The algorithm of Aggarwal et al. [1] can do this in time $O(p-k)$. It uses as an assumption that, for any four positions $i < i'$ and $j < j'$, if $f(i, j') \geq f(i, j)$, then $f(i', j') \geq f(i', j)$; i.e. in any submatrix, as we progress down the rows of the submatrices, the row maxima move monotonically to the right. Define a matrix of values having this property to be *totally monotonic*. It is the assumption of monotonicity that we must prove, in order to justify the use of the algorithm of Aggarwal et al.

Lemma 5. Let $f(i, j)$ be defined as above. Then the matrix of values of $f(i, j)$ for $k < i < p$ and $k + 1 < j \leq p$ is totally monotone.

Proof: First, if $(i' \geq j)$, then $f(i', j') \geq f(i', j) = -\infty$ and the conclusion holds. So we may assume that $i < i' < j < j'$. But then

$$\begin{aligned} f(i, j) + f(i', j') &= P[j] + P[j'] - w(i, j) - w(i', j') \\ &\geq P[j] + P[j'] - w(i', j) - w(i, j') \\ &= f(i', j) + f(i, j') \end{aligned}$$

by the quadrangle inequality. So if we assume $f(i, j') \geq f(i, j)$, then for the above inequality to hold $f(i', j') \geq f(i', j)$ and the matrix is monotone. •

A similar proof of the monotonicity of $D[i] + w(i, j)$ (with a definition of monotonicity for column minima instead of row maxima) holds for the use of the algorithm of Aggarwal et al. in computing the values of F in step 1. However that proof was given by Wilber for the analogous step in his algorithm, so we omit it here.

Now that we have determined the correctness of the algorithm, let us determine the amount of time it takes to compute each stage. Let k and c denote the values of the variables at the end of the stage, with k' and c' holding the values before the stage. The time for the stage is then $O((p - k') + (k' - c')) = O(k' - c')$. If, after the stage, $k = n$, we are done and the stage took time $O(n)$. If the stage finished without finding any $D[i] < B[i]$, then $k = p$ so $k - k' = 2k' - c' + 1 - k' = k' - c' + 1$ and the time spent is balanced by the increase in k. Otherwise, $c - c' = k - c' > k' - c'$ and the time spent is balanced by the increase in c. Both k and c are monotonically increasing and both are bounded by n. Thus as before the new algorithm takes linear time. But note that now we only calculate the values of $D[j]$ corresponding to actual computed values of $E[j]$; thus the algorithm can be safely interleaved with other computations, without loss of time.

Thus we have seen that recurrence 5, for concave cost functions, can be computed in linear time, even when the computation must be interleaved with other similar computations. We will use

this improvement to Wilber's algorithm as a subroutine in the next section. As another application, we can use it to solve the modified edit distance problem with concave costs in time $O(n^2)$.

Piecewise Convex and Concave Functions

Let us consider again recurrence 6:

$$E[j] = \min_{0 \le i < j} D[i] + g(j - i).$$

We assume that there exist indices $c_1, c_2, \ldots c_{s-1}$ such that g is concave in the range $[0, c_1]$, convex in the range $[c_1, c_2]$, and so on. By examining adjacent triples of the values of g, we can easily divide the numbers from 0 to n into such ranges in linear time; therefore from now on we assume that $c_1, c_2, \ldots c_{s-1}$ are given. Also define $c_0 = 0$ and $c_s = n$.

We now form s functions g_1, g_2, $\ldots g_{s+1}$ as follows. Let $g_p(x) = g(x)$ if $c_{p-1} \le x \le c_p$; for other values of x let $g_p(x) = +\infty$. Then

$$E[j] = \min_{1 \le p \le s} E_p[j], \tag{10}$$

where

$$E_p[j] = \min_{0 \le i < j} D[i] + g_p(j - i). \tag{11}$$

Our algorithm proceeds by solving recurrence 11 independently for each g_p, and then using equation 10 to find $E[j]$ as the minimum of the s results obtained. We use as subroutines the algorithms mentioned in the introduction for solving recurrence 5 when g is convex or concave.

It turns out that concave segments (i.e. g_p for odd p) remain concave on the entire range $[1, n]$, and therefore we could apply the algorithm of the previous section directly to them. The solution for convex segments is more complicated, because in this case the added infinities do interfere with convexity. Further, even if convexity held and we applied Klawe and Kleitman's algorithm in the straightforward way, we would only achieve a time of $O(n\alpha(n))$ for each segment; the bound we wish to achieve is $O(n\alpha(n/s))$. By a more complicated process we may solve both the concave and convex segments in such a way that the amortized time per segment is bounded by the formula above; however any individual segment p may have a solution time that is higher or lower than that bound, depending on its width $a_p = c_p - c_{p-1}$.

Now fix some segment p, either convex or concave. Then $E_p[j]$ depends on those values $D[i]$ such that $c_{p-1} \le j - i \le c_p$. Thus if we consider a matrix of pairs (i, j), the pairs such that $E_p(j)$ depends on $D[i]$ form a diagonal strip of width a_p. We solve the recurrence for g_p by dividing this strip into right triangles, of width and height a_p, pointing alternately to the upper right and lower left, and having as their hypotenuses alternately the diagonal borders of the strip determined by c_{p-1} and c_p (figure 2). We solve the recurrence independently on each triangle, and piece together the solutions to obtain the solution of g_p for the entire strip.

In particular, let the *upper triangle* U_t be the pairs (i, j) for which $j < c_p + (t-1)a_p$, $i \ge (t-1)a_p$, and $c_{p-1} \le j - i$. Similarly let the *lower triangle* L_t be the pairs (i, j) for which $j \ge c_p + (t-1)a_p$, $i < ta_p$, and $j - i \le c_p$. Then for any fixed j, all pairs (i, j) such that $E_p[j]$ depends on $D[i]$ may

be found in the union of the upper and lower triangles containing j. More formally,

$$\begin{aligned} E_p[j] &= \min_{0 \le i < j} D[i] + g_p(j-i) \\ &= \min_{c_{p-1} \le j-i \le c_p} D[i] + g_p(j-i) \\ &= \min\{X_p[j], Y_p[j]\}, \end{aligned}$$

where

$$\begin{aligned} X_p[j] &= \min_{j - c_{p-1} \ge i \ge \lceil (j-c_p+1)/a_p \rceil a_p} D[i] + g_p(j-i) \\ &= \min_{(i,j) \in U_{\lceil (j-c_{p-1}+1)/a_p \rceil}} D[i] + g_p(j-i) \end{aligned} \tag{12}$$

and

$$\begin{aligned} Y_p[j] &= \min_{\lceil (j-c_p+1)/a_p \rceil a_p > i \ge j - c_p} D[i] + g_p(j-i) \\ &= \min_{(i,j) \in L_{\lceil (j-c_p+1)/a_p \rceil}} D[i] + g_p(j-i). \end{aligned} \tag{13}$$

Thus we can compute the values of E_p by solving the values of X_p and Y_p within each upper and lower triangle. The computation within upper triangle U_t, corresponding to equation 12, can be expressed as follows:

$$X_p[j] = \min_{(t-1)a_p \le i \le j - c_{p-1}} D[i] + g_p(j-i), \tag{14}$$

which is exactly the same form as recurrence 5, except on a problem of size a_p instead of n. Further, all values of $g_p(j-i)$ in the recurrence are in the range $[c_{p-1} \le j - i \le c_p]$, so g_p is consistently either convex or concave in this range. Therefore, we can solve recurrence 14 in time $O(a_p \alpha(a_p))$ by using the Klawe and Kleitman's algorithm or our modified version of Wilber's algorithm.

The computation of Y_p in lower triangle L_t, which may be written

$$Y_p[j] = \min_{j - c_p \le i < t a_p} D[i] + g_p(j-i), \tag{15}$$

is however not of the appropriate form. In particular, for the least value of j in the triangle, $Y_p[j]$ depends on all of the values of $D[i]$ for i in the triangle; succeeding values of j use successively fewer values of $D[i]$. However, observe that this pattern of usage implies that all values of $D[i]$ will be known before any value of $Y_p[j]$ from the triangle need be computed. Therefore, this is an offline problem. Because of this offline nature of the problem, we need not compute the values of Y_p in order by j; in fact we will compute them in reverse order, to transform the corresponding search matrix to upper triangular form. Actually this step is not necessary, as the relevant algorithms can be applied directly to the lower triangles, but it simplifies the presentation.

Let $j' = c_p + t a_p - j$, and let $i' = t a_p - i$. Then equation 15 can be rewritten

$$Y_p[c_p + t a_p - j'] = \min_{1 \le i' \le j'} D[t a_p - i'] + g_p(c_p + i' - j'), \tag{16}$$

which is now the same form as that of recurrence 6. Finally note that if $g_p(j-i)$ is a convex function of $j - i$, then $g_p(c_p + i' - j')$ is also a convex function of $j' - i'$; and similarly if g_p is concave it remains so under the change of variables. Thus by reversing the order of evaluation we

have transformed equation 15 into a form that can be solved by Klawe and Kleitman's or Wilber's algorithms. Again the time taken for the solution is $O(a_p\alpha(a_p))$.

Each segment is composed of at most $O(n/a_p)$ upper and lower triangles; since the time for solving each triangle is $O(a_p\alpha(a_p))$, the total time to compute E_p for segment p is $O(n\alpha(a_p))$. The time for computing all segments is $\sum_{i=1}^{s} O(n\alpha(a_i))$, which, because $\sum a_p = n$ and by the convexity of the inverse Ackermann function, we can simplify to $O(ns\,\alpha(n/s))$. The time for combining the values from all segments is $O(ns)$. Therefore the total time for the algorithm is $O(ns\,\alpha(n/s))$.

Computation of RNA Structure

A dynamic program similar to that used above for sequence comparison has also been used for predicting the secondary structure of RNA [11, 4]. The recurrence is as follows:

$$E[i,j] = \min_{\substack{0\le i'<i \\ 0\le j'<j}} D[i',j'] + w(i'+j', i+j). \tag{17}$$

As before, $D[i,j]$ may be easily computed from $E[i,j]$. Also as before, we assume that w is a function only of the difference between the two diagonals: $w(i'+j', i+j) = g((i+j)-(i'+j'))$.

The naive dynamic programming solution takes time $O(n^4)$, and a simple observation of Waterman and Smith reduces this to $O(n^3)$ [14]. Eppstein, Galil and Giancarlo [4] showed that, if w is either convex or concave, this time can be further reduced to $O(n^2 \log^2 n)$. Aggarwal and Park [2] used different methods to reduce this time to $O(n^2 \log n)$. We now show that these results can be extended to piecewise convex and concave functions. If the number of segments in w is s, recurrence 17 can be solved in time $O(n^2 s \log n\,\alpha(n/s))$. For small s, this bound is much better than Waterman and Smith's time of $O(n^3)$. Our algorithm follows in outline that of Aggarwal and Park.

We solve recurrence 17 using divide-and-conquer techniques. Along with $E[i,j]$ we maintain another array $W[i,j]$, initially $+\infty$ at all cells. At all times $W[i,j]$ will be the minimum over some points (i',j') with $i' < i$ and $j' < j$ of $D[i',j'] + w(i'+j', i+j)$. At each recursive level we will divide the pairs of indices (i,j) into two sets; if (i',j') is in the first set and (i,j) is in the second, with $i' < i$ and $j' < j$, then after that level the minimization for $W[i,j]$ will include the value for (i',j'). The divide and conquer will ensure that all pairs (i',j') with $i' < i$ and $j' < j$ will eventually be included in $W[i,j]$, at which time we can simply take $E[i,j] = W[i,j]$.

Each level of the recursion proceeds in three phases. First, we compute recursively $E[i,j]$ for $0 \le j \le n/2$. Second, we compute $W[i,j]$ for $n/2 < j \le n$, using the formula

$$W[i,j] = \min_{\substack{0\le i'<i \\ 0\le j'\le n/2}} D[i',j'] + w(i'+j', i+j). \tag{18}$$

In the lower levels of the recursion, we take $W[i,j]$ as the minimum of its previous value and the above quantity. Finally, we compute recursively $E[i,j]$ for $n/2 < j \le n$ and combine the recursive computation with the values of $W[i,j]$ computed above. In each recursive call, we switch the roles of i and j so that each index will be halved at alternate levels of the recursion; thus the dynamic programming matrix seen at each level of the recursion remains square.

Lemma 6. The above algorithm sketch correctly computes $E[i,j]$ for each (i,j).

Proof: By induction on the number of levels in the recursion. $E[i,j]$ is computed correctly in the first recursive call by induction. In the second half of the matrix, half of the possible values which we are minimizing over are supplied by $W[i,j]$, and the other half are supplied by the recursion, so $E[i,j]$ is again computed correctly. •

Thus all that remains is to show how to solve recurrence 18. Fix i, and let $W_i[j] = W[i,j]$. Then the recurrence can be rewritten

$$\begin{aligned} W_i[j] &= \min_{\substack{0\le i'<i \\ 0\le j'\le n/2}} D[i',j'] + w(i'+j', i+j) \\ &= \min_{d} \min_{\substack{0\le i'<i \\ i'+j'=d \\ 0\le j'\le n/2}} D[i',j'] + w(i'+j', i+j) \\ &= \min_{d} Z[i,d] + w(d, i+j), \end{aligned} \tag{19}$$

where

$$Z[i,d] = \min_{\substack{0\le i'<i \\ i'+j'=d \\ 0\le j'\le n/2}} D[i',j']. \tag{20}$$

For a fixed diagonal d, equation 20 can be solved in time $O(n)$ using a prefix computation [9]. In particular $Z[i,d] = \min\{Z[i-1,d], D[i-1,d-i+1]\}$, so successive values of Z can be computed in constant time per value. Therefore all values of Z for the top level of the recursion can be computed in total time $O(n^2)$.

The remaining computation is recurrence 19. This follows a similar form to that of recurrence 5, and can be solved by the same methods. In fact the problem is offline (the values of Z on the right side of the equation do not depend on the values of W_i on the left) and so for convex or concave w, the recurrence can be solved in linear time by the algorithm of Aggarwal et al. [1]. This observation is the heart of the $O(n^2 \log n)$ algorithm of Aggarwal and Park [2] for the convex and concave cases of the RNA structure computation. For our case, w is neither convex nor concave, but mixed. As in the previous section, recurrence 19 can be solved by dividing the matrix of pairs (d,j) into diagonal strips, and the strips into triangles. This leads to a time of $O(ns\,\alpha(n/s))$ for solving each instance of recurrence 19, and a total time for all such recurrences of $O(n^2 s\,\alpha(n/s))$.

Thus we have seen that the time spent performing the computations at the outer level of our recursive algorithm is $O(n^2 s\,\alpha(n/s))$. We may compute the total time for the algorithm by expanding two recursive levels at once, one halving j and the next halving i, so that we return to the same square shape of the dynamic programming matrix at lower levels of the recursion. Let $T(n)$ be the complexity of solving the problem for an $n \times n$ dynamic programming matrix. This gives the equation

$$T(n) = 4T(n/2) + O(n^2 s\,\alpha(n/s)) = O(n^2 s\,\log n\,\alpha(n/s)).$$

Conclusions

Previous solutions to the dynamic programming equation $E[j] = \min_{0\le i<j} D[i] + w(i,j)$ assume that the cost function w is either convex or concave; that is, it satisfies either the quadrangle

inequality or its inverse. We have shown how to modify the linear-time algorithm for the concave case so that it can be interleaved with other computations, giving a time of $O(n^2)$ for the concave edit distance problem. We then showed how to adapt solutions to the convex and concave cases, to provide an efficient solution to the recurrence when w is neither convex nor concave, but can be divided into segments in each of which w is convex or concave. The resulting algorithm has applications both to approximate sequence comparison and to the computation of RNA structure.

Acknowledgements

I would like to thank my advisor, Zvi Galil, and my co-authors Pino Italiano and Raffaele Giancarlo for encouraging me to publish these results, and for many helpful comments. I would also like to thank an anonymous referee for his careful reading of the paper. This work was supported in part by NSF grants DCR-85-11713, CCR-86-05353, and CCR-88-14977.

References

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber, Geometric Applications of a Matrix-Searching Algorithm, Algorithmica 2, 1987, pp. 209–233.

[2] Alok Aggarwal and James Park, Searching in Multidimensional Monotone Matrices, 29th ACM Symp. Foundations of Computer Science, 1988, pp. 497–512.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[4] David Eppstein, Zvi Galil, and Raffaele Giancarlo, Speeding Up Dynamic Programming, 29th ACM Symp. Foundations of Computer Science, 1988, pp. 488–496.

[5] Zvi Galil and Raffaele Giancarlo, Speeding Up Dynamic Programming with Application to Molecular Biology, Theor. Comput. Sci., to appear.

[6] D.S. Hirschberg and L.L. Larmore, The Least Weight Subsequence Problem, SIAM J. Comput. 16, 1987, pp. 628–638.

[7] Maria M. Klawe and D. Kleitman, An Almost Linear Algorithm for Generalized Matrix Searching, preprint, 1987.

[8] Donald E. Knuth and Michael F. Plass, Breaking Paragraphs into Lines, Software Practice and Experience 11, 1981, pp. 1119–1184.

[9] Richard E. Ladner and Michael J. Fischer, Parallel Prefix Computation, J. ACM 27(4), 1980, pp. 831–838.

[10] Webb Miller and Eugene W. Myers, Sequence Comparison with Concave Weighting Functions, Bull. Math. Biol. 50(2), 1988, pp. 97–120.

[11] David Sankoff and Joseph B. Kruskal, editors, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983.

[12] Michael S. Waterman, General Methods of Sequence Comparison, Bull. Math. Biol. 46, 1984, pp. 473–501.

[13] Michael S. Waterman and Temple F. Smith, New Stratigraphic Correlation Techniques, J. Geol. 88, 1980, pp. 451–457.

[14] Michael S. Waterman and Temple F. Smith, Rapid Dynamic Programming Algorithms for RNA Secondary Structure, in Advances in Applied Mathematics 7, 1986, pp. 455–464.

[15] Robert Wilber, The Concave Least Weight Subsequence Problem Revisited, J. Algorithms 9(3), 1988, pp. 418–425.