Sparse Dynamic Programming II: Convex and Concave Cost Functions

David Eppstein
Zvi Galil
Raffaele Giancarlo
Giuseppe F. Italiano

CUCS-472-89

Sparse Dynamic Programming II: Convex and Concave Cost Functions

David Eppstein [1] **Zvi Galil** [1,2]
Raffaele Giancarlo [1,3] **Giuseppe F. Italiano** [1,4]

[1] Computer Science Department, Columbia University, New York, NY 10027
[2] Computer Science Department, Tel Aviv University, Tel Aviv, Israel
[3] Department of Mathematics, University of Palermo, Palermo, Italy
[4] Department of Computer Science, University of Rome, Rome, Italy

Abstract: We consider dynamic programming solutions to two recurrences, used to compute a sequence alignment from a set of matching fragments between two strings, and to predict RNA secondary structure. These recurrences are defined over a number of points that is quadratic in the input size; however only a sparse set matters for the result. We give efficient algorithms for solving these problems, when the cost of a gap in the alignment or a loop in the secondary structure is taken as a convex or concave function of the gap or loop length. Our algorithms reduce the best known time by a factor almost linear in the density of the problems: when the problems are sparse this results in a substantial speed-up.

Introduction

We are concerned here with two problems in sequence analysis, both solvable by dynamic programming. The first problem is alignment of sequences, and the second is prediction of RNA secondary structure. In both cases a cost function is involved: for sequence alignment this gives the cost of inserting or deleting a consecutive group of symbols, and for RNA structure this gives the cost of forming a loop of a given length. Also in both cases, the dynamic program has some sparsity in its structure, which we would like to exploit in the design of efficient algorithms for the problems. In a companion paper [8] we showed how to do this for cost functions that are linear in the length of the insertion or deletion, or in the length of the RNA loop. Here we extend these methods to cost functions that may be either convex or concave. Many of the cost functions that are likely to be used satisfy an additional property, which we will define following Hirschberg and Larmore [10] and Eppstein, Galil, and Giancarlo [7, 9]. For such functions, our algorithms can be made even more efficient.

Our algorithm for computing alignments from a sparse set of fragments runs in time $O(n + m + M \log M)$ for concave cost functions, and $O(n + m + M \log M \, \alpha(M))$ for convex cost functions. Here n and m are the lengths of the two input strings, M is the number of fragments found, and $\alpha(x)$ is the inverse Ackermann function, a very slowly growing function. The log function here, and throughout the paper, is assumed to be $\log x = \log_2(2 + x)$; i.e. when x is small the logarithm does not become negative. For simple convex cost functions the time can be further reduced to match the concave time bound. These bounds improve the previous best known time of $O(n + m + M^2)$ [30

Our algorithm for the prediction of RNA secondary structure with convex or concave cost functions for single loops runs in time $O(n + M \log M \log \min(M, n^2/M))$. Here n is the length of the input sequence, and M is the number of possible base pairs under consideration. When the cost function is simple, our bounds can be improved to $O(n + M \log M \log\log \min(M, n^2/M))$. The previous best known bound was $O(n^2 \log n)$ [3]; our bounds improve this by taking advantage of the sparsity of the problem.

Our algorithms are based on a common unifying framework, in which we find for each point of the sparse problem a geometric region of the dynamic programming matrix in which that point can influence the values of other points. We then resolve conflicts between different points by applying several algorithmic techniques in a variety of novel ways. In particular, previous algorithms for many of the problems we study have used either data structures [7, 9, 10] or matrix searching [1, 2, 3, 14, 28]. By combining both techniques, we achieve better bounds than either technique alone would give.

First let us define convexity and concavity as we will be using it. Each of our cost functions will be a two-parameter function $w(i, j)$, where i and j are both integer indices into the input sequences. We say that w is *concave* when, for all $i < i' < j < j'$, the *quadrangle inequality*

$$w(i, j') + w(i', j) \geq w(i, j) + w(i', j') \qquad (1)$$

is satisfied. We say that w is *convex* when the reverse inequality, which we call the *inverse quadrangle inequality*, is satisfied. For most applications, the cost function will actually depend only on the difference between its two parameters; in other words, $w(i, j) = g(j - i)$ for some function g. In this case, w will be convex or concave by the above definition exactly when g is convex or concave by the usual one-parameter definition.

The quadrangle inequality was introduced by Monge [19], and revived by Hoffman [11], in connection with a planar transportation problem. Later, F. Yao [31] used the inequality to solve a dynamic programming problem related to the construction of optimal binary search trees. Recently, the quadrangle inequality has seen use in a number of other dynamic programming algorithms for sequence analysis [3, 6, 7, 9, 14, 28].

A Dynamic Minimization Problem

We now describe a data structure to solve a minimization with dynamically changing input values. We will later use this data structure in our solution of the sparse sequence alignment problem. The data structure may also have independent interest of its own. We consider the following equation

$$E[i] = \min_j D[j] + w(i, j). \qquad (2)$$

Each of the indices i and j are taken from the set of integers from 1 through some bound n. The minimization for each $E[i]$ depends on all values of $D[j]$, not just those for which $j < i$. The cost function $w(i, j)$ is assumed to be either convex or concave. The values of $D[j]$ will initially be set to $+\infty$. At any time step, one of the following two operations may be performed:

(1) Compute the value of $E[i]$, for some index i, as determined by equation 2 from the present values of $D[j]$.

(2) Decrease the value of $D[j]$, for some index j, to a new value that must be smaller than the previous value but may otherwise be arbitrary.

We will give a data structure for this problem that will take $O(\log n)$ amortized time per operation. For simple cost functions, this time can be reduced to $O(\log\log n)$ amortized time per operation.

Equation 2 generalizes a number of problems that have appeared in other algorithms for sequence analysis, computational geometry, and other problems.

- Knuth and Plass [16] used a recurrence of the form $D[i] = \min_{j<i} D[j] + w(i,j)$ to break paragraphs into evenly spaced lines in the TEX program. They used the naive $O(n^2)$ dynamic program to solve the recurrence. Hirschberg and Larmore [10] gave algorithms for solving the recurrence in time $O(n \log n)$, assuming a weaker form of the quadrangle inequality than that used here. With the quadrangle inequality as we use it, the problem becomes trivial to solve. This recurrence can be seen as an example of equation 2, in which we consider the index i to range successively over the integers from 1 to n; for each i, we first calculate $E[i]$, and then include it in the recurrence by reducing $D[i]$ from $+\infty$ to the newly calculated value of $E[i]$.
- Aggarwal et al. [2] considered the problem of finding the minimum value of each row of an implicitly defined matrix, satisfying certain constraints. An important special case of their problem is the static version of equation 2, in which all values of $D[j]$ are specified before any value of $E[i]$ is computed. They gave a linear time algorithm for the matrix searching problem, and thus also this special case.
- Galil and Giancarlo [9] considered a generalization of the problem of Knuth and Plass, in which $D[i]$ may be computed in some simple but arbitrary way from the corresponding value of $E[i]$. This generalization can be applied for sequence alignment problems with non-linear gap costs. They gave an $O(n \log n)$ algorithm for this problem, when $w(i,j)$ is either convex or concave; a version of their algorithm takes linear time for simple functions.
- Wilber [28] extended the matrix searching techniques of Aggarwal et al. [2] to matrices in which the entries in each row depend dynamically on previously solved row minima. He used this to achieve a linear time solution to the concave case of Galil and Giancarlo's problem. However Wilber's algorithm is not suitable for the application to sequence alignment, or to other problems in which many instances of the problem are computed simultaneously. Eppstein [6] modified Wilber's algorithm to avoid these difficulties, while still taking only linear time.
- Aggarwal and Klawe [1] extended the matrix searching techniques of Aggarwal et al. [2] to staircase matrices, a class of matrices that includes as a special case triangular matrices, and used this result to solve some further computational geometry problems. Their algorithm for solving such matrices takes time $O(n \log\log n)$. Klawe and Kleitman [14] improved this result to $O(n\alpha(n))$, where α is the inverse Ackerman function. They further allowed the rows of the matrix to depend dynamically on previously computed row minima as in Wilber's algorithm This resulted in an improvement of the convex case of Galil and Giancarlo's problem to time $O(n\alpha(n))$.
- Eppstein, Galil and Giancarlo [7] gave an algorithm for computation of RNA structure, in which an important subproblem can be viewed as the computation of equation 2, when the values of $D[j]$ may be reduced in any order, but in which the values of $E[i]$ are computed

only in sequential order. They gave algorithms for this subproblem which take amortized time $O(\log n)$ per operation: for simple convex problems their algorithms take time $O(\log\log n)$ per operation. Aggarwal and Park [3] later improved their algorithm, by using a different method of computation based on the matrix searching techniques of Aggarwal et al. [2].

Our algorithm for the general dynamic equation above is similar to those of Galil and Giancarlo [9] and Eppstein, Galil and Giancarlo [7]. However we will later see how these techniques can be combined with matrix searching algorithms to provide further improvements.

We first show that we need only consider concave cost functions; the convex case will turn out to be essentially the same.

Lemma 1. If $w(i,j)$ is convex, then $w'(i,j) = w(i, n-j+1)$ is concave.

Proof: Let $f(j) = n - j + 1$. Then f maps the interval $1 \ldots n$ into itself. If $j < j'$, then clearly $f(j') < f(j)$. Therefore, if the inverse quadrangle inequality holds for $w(i,j)$, the inequality formed by reversing the order of j and j' holds for $w'(i,j) = w(i, f(j))$. But this is the same as the quadrangle inequality for $w'(i,j)$. •

Corollary 1. The dynamic minimization problem defined by equation 2, for convex weight functions $w(i,j)$, can be solved as a concave problem by reversing the order of the second index j.

From now on in this section we will assume without loss of generality that $w(i,j)$ is concave. Our algorithm is based on the following fundamental fact:

Lemma 2. For any i, j, and j', with $j < j'$, if $D[j] + w(i,j) \geq D[j'] + w(i,j')$, then for all $i' > i$, $D[j] + w(i',j) \geq D[j'] + w(i,j')$. Conversely, if $D[j] + w(i,j) \leq D[j'] + w(i,j')$, then for all $i' < i$, $D[j] + w(i',j) \leq D[j'] + w(i',j')$.

Proof: By the quadrangle inequality, $w(i,j') + w(i',j) \geq w(i,j) + w(i',j')$. Subtracting $w(i,j') + w(i',j') + D[j'] - D[j]$ from both sides and rearranging gives

$$(D[j] + w(i,j)) - (D[j'] + w(i,j')) \leq (D[j] + w(i',j)) - (D[j'] + w(i',j')).$$

But by assumption $(D[j] + w(i,j)) - (D[j'] + w(i,j'))$ is positive, and therefore $(D[j] + w(i',j)) - (D[j'] + w(i',j'))$ must also be positive and the first statement holds. The proof of the converse statement is similar. •

For specificity, let us break ties in favor of the smaller index. That is, we say that $D[j]$ is better than $D[j']$ at i if either $D[j] + w(i,j) < D[j'] + w(i,j')$, or $j < j'$.

Corollary 2. At any given time, the values of $D[j]$ supplying the minima for the positions of $E[i]$, with ties broken as above, partition the possible indices i into a sequence of intervals. If $j < j'$, if i is in the interval in which $D[j]$ is best, and i' is in the interval in which $D[j']$ is best, then $i < i'$.

Thus our algorithm need simply maintain the interval in which each value $D[j]$ is best, and a search structure of the interval boundaries, in which the interval containing a given point i can be looked up. Such a search structure can be maintained at a cost of $O(\log n)$ per modification or search, using any form of balanced binary trees [4, 15, 23]. If we use the *flat tree* data structure of van Emde Boas [24], this time can be reduced to $O(\log\log n)$. Thus it remains to show how to decrease a given value of $D[j]$, while maintaining the partition above and performing only $O(1)$ search tree operations.

In fact we may need to perform more than $O(1)$ search tree operations when we reduce a value of $D[j]$, because many other values of $D[j']$ may have their corresponding intervals reduced to nothing and thus will need to be removed from the search tree. We avoid that difficulty by, whenever we insert a value of $D[j]$ in the search tree, charging the operation with the time required to later delete it. In this way, each reduction will perform $O(1)$ non-charged search tree operations, and will be charged for $O(1)$ further operations which may occur in the future. The total is $O(1)$ operations per reduction, but the time bounds become amortized over the lifetime of the data structure rather than worst case per operation.

We call an index j into the array $D[j]$ *live* if, for some $E[i]$, $D[j]$ supplies the minimum in equation 2. As well as finding the interval containing a given index i into array E, we also need to search for the first live index before a given index j into array D. This can be done by maintaining another search tree or flat tree containing the live indices.

Let $R[j]$ be the rightmost (greatest) index in the interval corresponding to index j, and similarly let $L[j]$ be the leftmost index. For brevity, let $C(i,j)$ stand for $D[j] + w(i,j)$.

As in the algorithm of Galil and Giancarlo [9], we need a subroutine *border*(j,j'). This will always be called with $j < j'$; it returns the greatest index i such that $C(i,j) \leq C(i,j')$. If no such index exists, it returns 0. For arbitrary cost functions, lemma 2 can be used to derive a binary search routine that finds *border*(j,j') in time $O(\log n)$. For many functions, *border*(j,j') can be calculated directly as the root of a functional equation; we say that such a function has the *closest zero property*. Hirschberg and Larmore [10], and later Eppstein, Galil and Giancarlo [7, 9] used this property to derive more efficient algorithms for the problems they solved. Most simple functions that are likely to be seen in practice, such as logarithms and square roots, have the closest zero property.

The steps performed to reduce the value of $D[j]$ are as follows:

```
begin
    repeat
        find $j' < j$ as large as possible with $j'$ live;
        if no such $j'$ exists then $L[j] \leftarrow 1$; break;
        else if $C(L[j'],j) < C(L[j'],j)$ then begin
            $L[j] \leftarrow L[j']$;
            make $j'$ no longer live;
        end;
    end;
    if $j'$ still exists then begin
        $L[j] \leftarrow border(j',j)$;
        $R[j'] \leftarrow L[j] - 1$;
    end;

    repeat
        find $j' > j$ as small as possible with $j'$ live;
        if no such $j'$ exists then $R[j] \leftarrow n$; break;
        else if $C(R[j'],j) < C(R[j'],j)$ then begin
            $R[j] \leftarrow R[j']$;
            make $j'$ no longer live;
        end;
        end;
        if $j'$ still exists then begin
            $R[j'] \leftarrow border(j, j')$;
            $L[j] \leftarrow R[j'] + 1$;
        end;
        if $L[j] \leq R[j]$ and $j$ is not in the search structure then
            add $j$ to the search structure;
    end
```

Clearly a change in $D[j]$ can only affect the borders between it and its neighbors in the interval partition, and not any of the borders between unchanged values. Each iteration of the two loops above removes a point j' from the set of live points, exactly when the decrease in $D[j]$ expands the corresponding interval to cover the remaining interval of j'; that is, when j' no longer supplies the minimum at any point. The remaining steps fix the borders of the intervals between j and any remaining neighbors. It can be seen, using lemma 2, that the resulting partition is exactly that described by corollary 2. Thus the algorithm correctly solves the dynamic minimization problem we are interested in.

Theorem 1. The data structure above can be implemented to take $O(\log n)$ time, or $O(\log\log n)$ for functions with the closest zero property. The latter version also requires a setup time of $O(n)$.

Proof: The time for each reduction can be split into the time per iteration of the loops, and the remaining time. The loop time for an iteration deleting point j', as we have said, will be charged when we insert j' rather than when we delete it. This time is one search tree operation per iteration. Thus the time possibly charged to j will be one search tree operation. The remaining time consists of at most 4 search tree operations, to remove the old interval boundaries from the search tree and insert the new ones, and possibly to add j to the list of live indices. We also make two calls to the *border* subroutine. The total amortized time per operation is $O(\log n)$, or for simple functions $O(\log\log n)$. •

Sparse RNA Structure

The following recurrence has been used to predict RNA structure [7, 22, 25, 26]:

$$D[i,j] = \min\{D[i-1,j-1] + b(i,j), H[i,j], V[i,j], E[i,j]\}, \quad (3)$$

where

$$V[i,j] = \min_{0<k<i} D[k,j-1] + w'(k,i) \quad (4)$$

$$H[i,j] = \min_{0<l<j} D[i-1,l] + w'(l,j) \quad (5)$$

$$E[i,j] = \min_{\substack{1\leq i'<i-1 \\ 1\leq j'<j-1}} D[i',j'] + w(i'+j', i+j). \quad (6)$$

The function w corresponds to the energy cost of a free loop between the two base pairs, and w' corresponds to the cost of a bulge. Both w and w' typically combine terms for the loop length and for the binding energy of bases i and j. The function $b(i,j)$ contains only the base pair binding energy term, and corresponds to the energy gain of a stacked pair (see [22] for definitions of these terms). The companion paper [8] describes why the number M of base pairs (i,j) such that $D[i,j] < +\infty$ may be taken to be significantly less than n^2, and uses this fact to improve the time for solving these recurrences when w and w' are linear: here we instead allow them to be convex or concave.

In fact to compute the best structure for an RNA sequence, rather than simply the best score for the structure, we need to also maintain for each pair (i,j) a pointer to the pair (i',j') supplying the minimum. It is not difficult to modify our algorithm to maintain such pointers; we omit the details.

First note that the computation of $V[i,j]$ within a fixed column j does not depend on that in other columns, except indirectly via the values of $D[i,j]$. We may perform this computation using the algorithm of Galil and Giancarlo [9]; if the number of points i in column j such that $D[i,j] \neq +\infty$ is denoted by p_j, then the time for computing all values of $V[i,j]$ for a fixed j will be $O((p_j + p_{j-1}) \log M)$. The total time for these computations in all columns will then be $O(M \log M)$. We could achieve even better bounds using the more complicated algorithms of Klawe and Kleitman [14] or Eppstein [6], but this would not affect our total time bound.

The computation of $H[i,j]$ is similar. Therefore the remaining difficulty is the computation of $E[i,j]$, as defined by recurrence 6. For simplicity of exposition we relax the condition in the recurrence that $i' < i - 1$ and instead allow $i' < i$; similarly with j and j'. However the algorithm we describe works essentially unchanged for the actual conditions on i' and j'.

The obvious dynamic programming algorithm solves recurrence 6 for sequences of length n in time $O(n^4)$ [22]; this can be improved to $O(n^3)$ [26]. When w is a linear function of the distance between back diagonals $(i' + j') - (i + j)$, another easy dynamic program solves the problem in time $O(n^2)$ [13]. In the companion paper we reduce this time bound to $O(n + M \log\log\min(M, nm/M))$ [8].

Here we consider instead the case that the cost function is either convex or concave. Eppstein et al. [7] found a $O(n^2 \log^2 n)$ algorithm for such costs; this was later improved to $O(n^2 \log n)$ [3]. We would like to again use the sparsity of the possible base pairs to further reduce the time for the problem with convex or concave costs. We assume that the possible base pairs have already been enumerated; the companion paper [8] explains how this may be done.

Each point (possible base pair) may be considered as having a *range of influence* consisting of the region of the dynamic programming matrix below and to the right of it. Thus the range of each point is a quarterplane with vertical and horizontal boundaries. We first effectively remove the horizontal boundaries, leaving half-planar ranges, at a logarithmic cost in execution time. This is done as follows.

We solve the problem by a divide and conquer recursion on the rows of the dynamic programming matrix. For each level of the recursion, having t points in the subproblem for that level, we choose a row r such that the numbers of points above r and below r are each at most $t/2$. Such a row must always exist, and it can easily be found in linear time. Thus we can partition the points of the problem into three sets: those above r, those on r, and those below r. In fact it would be possible to partition the points into only two sets, by including the first half of the points on r among the points below r, and including the second half of the points on r among the points above r. However the correctness of the algorithm is easier to see with the three-part division; and since the best way of computing the two-part division seems to be by first computing the three-part division, we might as well just use the three part division.

Within each level of the recursion, we will need the points of each set to be sorted by their column number. This can be achieved by initially bucket sorting all points, and then at each level of the recurrence performing a pass through the sorted list to divide it into the three sets. Thus the order we need will be achieved at a linear cost per level of the recurrence.

We note that for any point above or on r, the minimum value in equation 6 only depends on the values of other points above r. For points below r, the value of equation 6 is the minimum between the values from points above r, and the points below r. Thus we can compute all the minima by performing the following steps: (1) solve the problem above r by a recursive invocation of our algorithm, (2) use the values given by this solution to solve the problem for the points on r, (3) compute the influence of the points above or on r, on the values of the points below r, and (4) recursively solve the problem below r.

This divide and conquer technique is similar to the dynamic-to-static reduction of Bentley and Saxe [5]; it differs from the RNA structure algorithm of Aggarwal and Park [3] in that we divide only by rows, and not by columns. It does not seem possible to modify the algorithm of Aggarwal and Park to run in time depending on the sparsity of the problem, because at each level of their recursion they compute a linear number of matrix search problems, the size of each of which does not depend on the sparsity of the problem.

The problem remaining after our recursion is as follows. We are given a set A of points above a certain row of the matrix, and a set B of points below the row. Both sets are sorted by column number. The values of the points in A are known, and we want to know their contributions to the minimizations for each of the points in B. Each level of the divide and conquer recursion computes the solution to two such problems, one with A the points above row r and B the points on row r, and a second with A the points above or on row r and B the points below row r.

We now write a recurrence equation for the reduced subproblem:

$$E[i,j] = \min_{\substack{(i',j')\in A \\ 1\le j'<j}} D[i',j'] + w(i'+j', i+j). \tag{7}$$

The crucial difference between this and equation 6 is that now, the requirement that $i' < i$ has been subsumed by the separation into sets A and B. In other words, the horizontal boundaries of the quarter-planar regions of influence have been removed, leaving only the vertical boundaries. Thus the range of influence of each point in A is the subset of B to the right of the point, and points in A are totally ordered by inclusion of the ranges. We use the total order to add the points of A to the data structure described in the previous section, so that when we process each point of B exactly the points that influence it will have been added to the data structure.

In particular, we process the points of A and B in order by their column numbers. The details of this processing will be given below. Within a given column, we first process the points of B, and then the points of A. By proceeding along the sorted lists of points in each set, we need only spend time on columns that actually contain points, so there will be no time loss determining which points to process next. Clearly, if we use this order, then whenever we process a point (i,j) from the set B, the points (i',j') of A that will have been processed will be exactly those with $j' < j$

We process points by maintaining a copy of the data structure described in the previous section. To recall, the data structure maintains a matrix of values $D[y]$, initially all $+\infty$. At each step, the algorithm may either decrease one such value, or it may answer a query of the form

$$E[x] = \min_y D[y] + w(y, x). \tag{8}$$

It is easily seen that equation 8 is like equation 7, but with points (i, j) replaced by the numbers $i+j$ of their diagonals, and with the requirement that $j' < j$ removed. As we have described above, this last requirement will be taken care of by the order in which we process the points.

To process a point (i, j) from A, with value v, we let $y = i + j$ be the number of the diagonal containing the point, and reduce $D[y]$ to $\min(D[y], v)$. To process a point (i, j) from B, we let $x = i + j$ be the number of the diagonal containing the point, and compute the influence of the points in A on the value at (i, j) to be $E[x]$ as in equation 8.

This completes the solution of equation 7, and thus the solution of recurrence 6. To summarize, the algorithm solving the recurrence can be written in pseudo-code as follows:

```
procedure RNA($x, y$):
begin
    find sparse set $X$ of possible base pairs from the two strings;
    sort $X$ by column numbers;
    let arrays $E$ and $D$ be indexed by members of $X$;
    for $x \in X$ do $E[x] = +\infty$;
    Recurse($X$);
end
```

The recursive subprocedure called above solves the problem within the set of points given, assuming the influences of previous points have already been included in the computation. The input set of points is assumed sorted by column numbers, and the splitting of that set into subsets A, B, and C must maintain that sorted order. Note that there is no call to *Recurse*(B) because the points in B, being all on the same row, cannot influence each other.

```
procedure Recurse($X$):
begin
    let $j$ be a row with at most $|X|/2$ points above and below it;
    let $A$ be the points above row $j$ in $X$;
    let $B$ be the points on row $j$;
    let $C$ be the points below row $j$;
    if $A \neq \emptyset$ then begin
        Recurse($A$);
        Influence($A, B$);
    end;
    for $x \in B$ do
        compute $D[x]$ from $E[x]$;
    if $C \neq \emptyset$ then begin
        Influence($A \cup B, C$);
        Recurse($C$);
    end;
```

```
end
```

Finally we give pseudo-code for computing the influence of one set of points on another, in which all the actual work of solving the recurrence is performed. Again, the input sets are sorted by column.

```
procedure Influence(A, B):
begin
    let $X = A \cup B$, maintaining sorted order: if $A$ and $B$ both have
        points in the same column let those of $B$ come first;
    let $F[i] = \min_j G[j] + w(j, i)$ be solved by the
        data structure of the previous section;
    for $x \in X$ in order do begin
        $d \leftarrow row(x) + column(x)$;
        if $x \in A$ then $G[d] \leftarrow \min(G[d], D[x])$
        else $E[x] \leftarrow \min(E[x], F[d])$;
    end;
end
```

Before we give the time bound, let us first note that the time per data structure operation can be taken to be $O(\log M)$ or $O(\log\log M)$ rather than $O(\log n)$ or $O(\log\log n)$. This is because we need only consider diagonals of the dynamic programming matrix that actually contain some of the M points in the sparse problem. We number these non-empty diagonals in order by their real diagonal numbers: it is easily seen that this change does not affect the convexity or concavity of the cost function. However closest zero functions have the complication that the computation of *border*(x, y) is defined in terms of actual diagonal numbers. Therefore we need to translate actual diagonal numbers into the nearest non-empty diagonals; a table to perform this translation can be created in $O(n + M)$ time, and used throughout the algorithm.

Theorem 2. The RNA structure computation of recurrence 6, for a sequence of length n, with M possible base pairs, and convex or concave cost functions, can be performed in time $O(n + M \log^2 M)$. For cost functions with the closest zero property, the computation can be performed in time $O(n + M \log M \log\log M)$.

Proof: Denote the number of points processed at a given level of the recurrence by t. Then the time taken at that level is $O(t)$, together with $O(t)$ operations from the data structure of the previous section. The time per data structure operation is either $O(\log M)$ or $O(\log\log M)$, as described above. The latter version also requires $O(M)$ preprocessing time to set up the flat tree search structures; however the same structures can be re-used at different levels of the recursion and so this setup time need only be payed once. The divide and conquer adds another logarithmic factor to this bound. We also need to compute the possible base pairs and bucket sort them, in a preprocessing stage taking time $O(t)$. The details of this generation are given in the companion paper [8]. The total time to solve recurrence 6 is $O(n + M \log^2 M)$ in general, or $O(n + M \log M \log\log M)$ for simple functions. •

Improved RNA Structure Computation for Intermediate Density

In the introduction we promised a time bound for the RNA structure computation of $O(n + M \log M \log \min(M, n^2/M))$ in general, and $O(O(n + M \log M \log\log \min(M, n^2/M))$ for simple cost functions. Yet in the previous section the bounds we gave were only $O(n + M \log^2 M)$ or $O(n + M \log M \log\log M)$. Here we describe how to improve our algorithms to run within the time bounds we claimed. We assume without loss of generality that $n < M$; otherwise, the bounds given in the previous section reduce to those here.

First let us examine the algorithm for simple functions. The algorithm for arbitrary functions is similar but requires a few more ideas. Our algorithms will be similar to those of the previous section, but the divide and conquer scheme will be different. Instead of dividing only by rows, we divide alternately by rows and columns, similarly to the divide and conquer technique used in the non-sparse RNA structure algorithm of Aggarwal and Park [3]. More precisely, at even levels of the divide and conquer recurrence we divide the dynamic programming matrix at some row i as before; however we choose i to be the center of the matrix rather than the center of the sparse set of points in the matrix. At odd levels we similarly divide by columns. In this way, each level of the recursion performs a computation in a matrix that is either square or close to square; there can be $O(\log n)$ levels before the recursion bottoms out at single points.

In terms of the pseudo-code given in the previous section, we need two versions of *Recurse*, one that divides by rows and one that divides by columns, each of which calls the other. We also need two versions of *Influence*, one to be called by each version of *Recurse*. All of these procedures keep the sets of points they handle in two sorted orders: sorted by rows, and sorted by columns. Unlike the code of the previous section, we divide into only two sets A and C; the line of division between them will be halfway between two actual rows or columns, and so there is no in-between set B. Further this line of division is chosen by halving the number of columns in the sets, instead of halving the number of points.

As before, we compute the values of the points in A recursively, compute the influence of these values on those of the points in C, and then finish the computation of the values in C recursively. In the description that follows we assume that the current level in the divide and conquer recursion is even, so that as in the previous section the division between A and C occurs on a row boundary; the computation for odd levels is similar.

In the previous section, we computed the value D from E for each point when it was part of set B. Because here there is no such set, we must do so at another time; in particular we do so when the recursion bottoms out, and all points are on a single row or column. In this way the value is computed exactly once for each point, before it is needed.

Thus the **pseudo-code** for the procedures can be written as follows. We have merged the *Influence* **procedure** in with *Recurse*, because it would have been called only in one place. We only show one of the two mutually recursive procedures; the other can be found by replacing rows by columns and vice versa.

```
procedure RecurseColumn($X$):
begin
    let $i$ and $k$ be the first and last rows occurring in $X$;
    if $i = k$ then
        for $x \in X$ do
```

```
            compute $D[x]$ from $E[x]$;
    else begin
        $j \leftarrow \lceil (i+k)/2 \rceil$;
        let $A$ be the points above row $j$ in $X$;
        let $C$ be the points on or below row $j$;
        $RecurseRow(A)$;
        let $F[i] = \min_j G[j] + w(j,i)$ be solved by the data structure
            of the first section, modified as described below;
        for $x \in X$ in order by columns do begin
            $d \leftarrow row(x) + column(x)$;
            if $x \in A$ then $G[d] \leftarrow \min(G[d], D[x])$
            else $E[x] \leftarrow \min(E[x], F[d])$;
        end;
        $RecurseRow(C)$;
    end;
end
```

In the data structure of the first section, in place of the flat trees of van Emde Boas, we use Johnson's improved flat trees [12]. This is again a structure in which one can insert, delete, and search for points numbered from 1 to n. However, whereas flat trees take time $O(\log\log n)$ per operation, improved flat trees take time $O(\log\log D)$, where D is the length of the gap between points in the structure containing the point being searched for, inserted, or deleted. More importantly for our analysis, a sequence of k operations, all of one type (insertions, deletions, or searches), can be performed in total time $O(k \log\log n/k)$. For insertions and deletions this follows from Johnson's analysis of his algorithm.

For a sequence of searches in order by the positions being searched for, the time bound follows because consecutive searches in the same gap can be detected in constant time, by simply comparing each new search point with the endpoints of the gap containing the previous search point. Therefore we need pay the $O(\log\log D)$ cost at most once per gap. The cost of the sequence of search operations is therefore $\sum_{i=1}^{k} O(\log\log D_i)$. Because the function $f(x) = \log\log x$ is convex, any sum $\sum_{i=1}^{k} f(x_i)$ is maximized when the x_i are all equal, and so the sum is bounded by $kf(\sum_{i=1}^{k} x_i/k)$. In particular, the cost of the search sequence can be reduced to

$$O(k \log\log(\sum_{i=1}^{k} D_i/k)) = O(k \log\log n/k).$$

For a sequence of k searches in non-sorted order we can use another instance of Johnson's flat trees to sort the search points, in time $O(k \log\log n/k)$, and then perform the searches in sorted order as above; however all our sequences of searches will in fact be already in sorted order.

Theorem 3. The RNA structure computation of recurrence 6, for a sequence of length n, with M possible base pairs, and convex or concave cost functions with the closest zero property, can be performed in time $O(n + M \log M \log\log \min(M, n^2/M))$.

Proof: Consider the time for the top level of the recursion, which we denote $T(0)$. The algorithm from the previous section consists of, for each column, performing a sequence of searches and then a sequence of insertions and deletions. Let the number of searches in column i be denoted

s_i, and the number of insertions and deletions be denoted d_i. Then $\sum d_i$ is at most twice the total number of points in set A, and $\sum s_i$ is the total number of points in set C. The time taken is $\sum s_i \log\log n/s_i + \sum d_i \log\log n/d_i$. In the function $f(x) = x \log\log n/x$, the $\log\log n/x$ term decreases as x increases, and so the function as a whole is sublinear and therefore convex. Because of this convexity the total time taken at the given recursive stage in the algorithm can be reduced to

$$\begin{aligned} T(0) &= \sum_{i=1}^{n} O(f(s_i) + f(d_i)) \\ &\leq O(nf(\sum_{i=1}^{n} s_i/n)) \\ &= O(nf(M/n)) \\ &= O(n(M/n)\log\log \frac{n}{M/n}) \\ &= O(M \log\log n^2/M). \end{aligned}$$

An identical analysis applies to each even level recursive subproblem, with n replaced in the bound by the size of the matrix for the subproblem. Similar bounds hold for the odd levels.

Now let us consider the sum of the times for all stages at a given level $2i$. As before, the analysis for odd levels is similar. Let M_j, for j from 1 to 2^{2i}, be the number of points in subproblem j. Further, at the given level, there will be 2^{2i} subproblems, each of having 2^i rows and columns. Then, by convexity, the total time for the level is

$$\begin{aligned} T(i) &= O(\sum_{j=1}^{2^{2i}} M_j \log\log \frac{(n/2^i)^2}{M_j}) \\ &\leq O(2^{2i}(M/2^{2i})\log\log \frac{(n/2^i)^2}{M/2^{2i}}) \\ &= O(M \log\log n^2/M). \end{aligned}$$

There are $O(\log M)$ levels in the recursion, and as we have shown above each takes time bounded by $O(M \log\log n^2/M)$, so the total time is $O(M \log M \log\log \min(M, n^2/M))$. •

For non-simple functions, we must also take into account the binary searches required to compute *border*(i,j) when including new values from A into the data structure. Assume that k such computations need be performed for a given column. If all the binary searches occurred in disjoint intervals of the range from 1 to n, the total time would be $O(k \log n/k)$ and a similar analysis to that for simple functions would give a total time bound of $O(M \log M \log \min(M, n^2/M))$. To force the search intervals to be disjoint, we first find the borders among the points being inserted

In particular, we need to solve an instance of equation 2, in which there are k new values of $D[j]$ given. By corollary 2 of the first section, each value of $D[j]$ supplies the minima for $E[i]$ with i in some interval of the range from 1 to n, and further these intervals appear in the same order as the positions of $D[j]$. Clearly, *border*(i,j) for a newly added point j need only be computed within the interval in which $D[j]$ is better than the other new points. Further, all computations of *border*(i,j) have at least one of the indices i or j being a newly added point. If, given the set of new values to be inserted, we can compute the partition of $[1 \ldots n]$ into intervals in which each of these values is best, guaranteed to exist by corollary 2, we can use this partition to perform each *border*(i,j) computation in a disjoint interval, and therefore the total time for these computations for k new points will be $O(k \log n/k)$.

The algorithm of Aggarwal et al. [2] can find the minima at all n points, and therefore the boundaries between the intervals, in time $O(k+n)$. That of Galil and Giancarlo [7, 9], which uses binary searches to find interval borders as in the data structure of the first section, but which needs only a stack instead of a more complicated search structure, can find the boundaries in time $O(k \log n)$. We combine these two algorithms to achieve a bound of $O(k \log n/k)$, which is what we need to solve the RNA structure problem in the given time bound.

This is done as follows. We first select the points $E[n/k]$, $E[2n/k]$, etc, and find for each of these points which value of $D[j]$ supplies the minimum. This computation involves only k points $E[i]$, and so we can solve it in time $O(k)$ using the algorithm of Aggarwal et al. The remaining points in the range from 1 to n are divided up by this computation into k segments, each of length n/k. For each boundary between values $D[j-1]$ and $D[j]$, we know from the above computation which segments it falls in. If the two endpoints of an interval both have the same value of $D[j]$ supplying their minima, there can be no boundary within that interval. Otherwise, if $D[i]$ is the left minimum and $D[j]$ the right minimum, the segments will contain only those boundaries of intervals corresponding to positions between i and j.

Thus for each segment we can perform a binary search, as used in the computation of Galil and Giancarlo, for the boundaries that may fall within that interval. Each binary search is thus limited to a range of n/k points, and so it will take time $O(\log n/k)$. Each value of $D[j]$ is involved in the computations for at most two segments, those to the left and to the right of the segment border points for which it supplies the minima. If $D[j]$ does not supply the minimum for any segment border point, it will be involved in the computation for exactly one segment. The time for a segment containing b boundaries will be $b \log n/k$, and so the total time for computing boundaries between new point intervals is $O(k \log n/k)$ as desired.

Once we have computed the boundaries between the intervals of the new points being inserted, we can insert the points into the data structure as before, computing *border*(i,j) by a binary search that stays within the interval of the point being inserted. The sum of the interval lengths is bounded by n, so the time for insertion is bounded by $O(k \log n/k)$.

Theorem 4. The RNA structure computation of recurrence 6, for a sequence of length n, with M possible base **pairs, and** arbitrary convex or concave cost functions with the closest zero property, can be performed **in time** $O(n + M \log M \log \min(M, n^2/M))$.

Proof: As we have shown above, the time for computing borders in k consecutive insertions in the data structure can be reduced to $O(k \log n/k)$. For k consecutive deletions, we delay recomputing any borders until all deletions have been made, so that each recomputed border is between points that will not be deleted. In this way the binary searches are again in disjoint intervals and the time is again $O(k \log n/k)$. Lookups, as well as the remaining computations for insertions and deletions, can be handled with Johnson's data structure in time $O(k \log\log n/k) = O(k \log n/k)$ Thus for any sequence of consecutive operations of the same type, the time is $O(k \log n/k)$. By a similar analysis as that for simple functions, the total time is $O(n + M \log M \log \min(M, n^2/M))$. •

Sparse Sequence Comparison

In this section we are concerned with the comparison of two sequences, of lengths n and m, which differ from each other by a number of mutations. An alignment of the sequences is a non-crossing matching of positions in one with positions in the other, such that the number of unmatched positions (insertions and deletions) and matched positions with the symbol from one sequence not the same as that from the other (point mutations) is kept to a minimum. This is a well-known problem, and a standard dynamic programming technique solves it in time $O(nm)$ [20]. In a more realistic model, a sequence of insertions or deletions would be considered as a unit, with the cost being some simple function of its length; sequence comparisons in this more general model can be solved in time $O(n^3)$ [27]. The cost functions that typically arise are convex; for such functions this time has been reduced to $O(n^2 \log n)$ [9, 7, 18] and even $O(n^2\alpha(n))$, where α is a very slowly growing function, the functional inverse of the Ackermann function [14].

Since the time for all of these methods is quadratic or more than quadratic in the lengths of the input sequences, such computations can only be performed for fairly short sequences. Wilbur and Lipman [29, 30] proposed a method for speeding these computations up, at the cost of a small loss of accuracy, by only considering matchings between certain subsequences of the two input sequences. In particular, their algorithm finds the best alignment in which each matched pair of symbols is part of a contiguous sequence of at least k matched symbols, for some fixed number k.

Let the two input sequences be denoted $x_1x_2 \ldots x_m$ and $y_1y_2 \ldots y_n$. Wilbur and Lipman's algorithm first selects a small number of *fragments*, where each fragment is a triple (i, j, k) such that the k-tuple of symbols at positions i and j of the two strings exactly match each other; that is, $x_i = y_j$, $x_{i+1} = y_{j+1}$, $\ldots$, $x_{i+k-1} = y_{j+k-1}$. We do not describe here how such fragments are found; such a description can be found in the companion paper [8]. It suffices to mention that such fragments can be found in time $O(n + m + M)$ using standard string matching techniques.

A fragment (i', j', k') is said to be *below* (i, j, k) if $i + k \leq i'$ and $j + k \leq j'$; i.e. the substrings in fragment (i', j', k') appear strictly after those of (i, j, k) in the input strings. Equivalently we say that (i, j, k) is *above* (i', j', k'). The *length* of fragment (i, j, k) is the number k. The *forward diagonal* of a fragment (i, j, k) is the number $j - i$. This differs from the *back diagonals* $i + j$ used for the RNA structure computation. Here we will use both back and forward diagonals.

An *alignment* of fragments is defined to be a sequence of fragments such that, if (i, j, k) and (i', j', k') are adjacent fragments in the sequence, either (i', j', k') is below (i, j, k) on a different forward diagonal (a *gap*), or the two fragments are on the same forward diagonal, with $i' > i$ (a *mismatch*). The cost of an alignment is taken to be the sum of the costs of the gaps, minus the number of matched symbols in the fragments. The number of matched symbols may not necessarily be the sum of the fragment lengths, because two mismatched fragments may overlap. Nevertheless, it is easily computed as the sum of fragment lengths minus the overlap lengths of mismatched fragment pairs. The cost of a gap is some function of the distance between forward diagonals $g(|(j - i) - (j' - i')|)$.

When the fragments are all of length 1, and are taken to be all pairs of matching symbols from the two strings, these definitions coincide with the usual definitions of sequence alignments. When the fragments are fewer, and with longer lengths, the fragment alignment will typically approximate fairly closely the usual sequence alignments, but the cost of computing such an alignment may be much less.

The method given by Wilbur and Lipman for computing the least cost alignment of a set of fragments is as follows. Given two fragments, at most one will be able to appear after the other in any alignment, and this relation of possible dependence is transitive; therefore it is a partial order. We process fragments in the order of any topological sorting of this order. Some such orders are by rows (i), columns (j), or back diagonals ($i+j$).

For each fragment, the best alignment ending at that fragment is taken as the minimum, over each previous fragment, of the cost for the best alignment up to that previous fragment together with the gap or mismatch cost from that previous fragment. The mismatch cost is simply the length of the overlap between two mismatched fragments; if the fragment whose alignment is being computed is (i,j,k) and the previous fragment is $(i-x, j-x, k')$ then this length can be computed as $\max(0, k'-x)$. From this minimum cost we also subtract the length of the new fragment; thus the total cost of any alignment includes a term linear in the total number of symbols aligned. Formally, we have

$$C(i,j,k) = -k + \min \begin{cases} \min\limits_{(i-x,j-x,k')} C(i-x, j-x, k') + \max(0, k'-x) \\ \min\limits_{(i',j',k') \text{ above } (i,j,k)} C(i',j',k') + g(|(j-i)-(j'-i')|) \end{cases} \tag{9}$$

The naive dynamic programming algorithm for this computation, given by Wilbur and Lipman, takes time $O(M^2)$. If M is sufficiently small, this will be faster than many other sequence alignment techniques. However we would like to speed the computation up to take time linear or close to linear in M; this would make such computations even more practical for small M, and it would also allow more exact computations to be made by allowing M to be larger.

In the companion paper [8], we show how to perform this computation for linear functions $g(x)$ in time $O(n+m+M \log\log \min(M, nm/M))$. Here we consider the problem for convex and concave cost functions.

We consider recurrence 9 as a dynamic program on points in a two-dimensional matrix. Each fragment (i,j,k) gives rise to two points, (i,j) and $(i+k-1, j+k-1)$. We compute the best alignment for the fragment at point (i,j); however we do not add this alignment to the data structure of already computed fragments until we reach $(i+k-1, j+k-1)$. In this way, the computation for each fragment will only see other fragments that it is below. We compute separately the best mismatch for each fragment; this is always the previous fragment from the same diagonal and so this computation can easily be performed in linear time. From now on we will ignore the distinction between the two kinds of points in the matrix, and the complication of the mismatch computation.

As in the RNA structure computation, each point has a range consisting of the points below and to the left of it. However for this problem we divide the range into two portions, the *left influence* and the *right influence*. The left influence of (i,j) consists of those points in the range of (i,j) which are below and to the left of the forward diagonal $j-i$, and the right influence consists of the points above and to the right of the forward diagonal. Within each of the two influences, $g(|p-q|) = g(p-q)$ or $g(|p-q|) = g(q-p)$; i.e. the division of the range in two parts removes the complication of the absolute value from the cost function.

Thus the computation looks very similar to that for RNA structure, except that here we have two separate computations, and where in the RNA structure computation we had ranges that were quarter-planar geometric regions, now we have two collections of influences that are eighth-planar geometric regions. The minimization over either the left or the right influences turns out to be an affine transformation of the RNA structure problem, and so one would think that the same methods apply. In fact the algorithms for this problem are very similar, but more complicated, because we must use the same evaluation order to solve both the left and right subproblems.

Our algorithm for this problem can be viewed as a novel application of the Bentley-Saxe dynamic-to-static reduction: we perform two such reductions, in two different orders, one for each type of eighth-plane piece of the fragment point ranges. The differing order leaves the problem dynamic, but the reduction instead can be imagined as removing the vertical or horizontal boundaries of the pieces, leaving only the forward-diagonal boundaries. The reduced subproblem can then be solved with matrix-searching techniques.

We first cut the domains of each point into right and left pieces, as described above. We divide the points into subproblems, and then proceed to compute the values of the recurrence at each point. Each value we compute is derived from the subproblems containing the given point, and once we have computed this value we apply it in the subproblems depending on it. The order in which we will compute the values at each point will be by back diagonals. This order is is symmetric with respect to the two kinds of pieces, so without loss of generality from now on we need only consider the subproblems derived from the right pieces, i.e. those eighth-plane pieces which are bordered on two sides by rows and forward diagonals.

As in the RNA structure computation, we use divide and conquer to produce the subproblems into which we divide the computation. Each point will be in set A for $O(\log n)$ subproblems and set B for $O(\log n)$ subproblems. However within the divide and conquer we only compute the structure of the subproblems; that is, we determine for each subproblem its corresponding sets A and B. We do not immediately attempt to solve the subproblems, because that would violate the processing order by back diagonals. Instead we produce a data structure maintaining the state of each subproblem. Only after all subproblems have been so constructed do we then proceed to solve the recurrence, in order by back diagonals as stated above. After we begin solving the recurrence, we will maintain each subproblem dynamically, including the values from points in set A as they become known, and computing the subproblem minima for points in set B as they become available

In each subproblem, as in the RNA structure computation, the points in A are those above some row and the points in B are those below the same row. The minimization for point (i,j) in B depends on the value at a point (i',j') in A exactly when $j'-i' < j-i$. Thus we order the points in the subproblems by the numbers of their forward diagonals. As in the RNA structure computation such an order can be maintained by initially bucket sorting all points, and then splitting the sorted list at each level of the recursion.

The actual order in which the subproblems receive the values of $D[x]$ for points in set A w...

be more arbitrary than that described above, as will be the order in which the values that have been determined within the subproblem for points in set B are requested by the main program. However the forward diagonals totally order the points by their dependence on each other. The subproblem solution proceeds by saving each given value of $D[x]$ until all previous values in the dependence order are known, and then computing as many derived values as possible with the known values and saving these derived values until the main program asks for them. In this way each subproblem solution operates asynchronously of the main program. All we require is that, whenever the main program asks for the subproblem's value at a point x in set B, all values $D[y]$ for points y on previous forward diagonals of set A will have already been given to the subproblem.

It turns out that, with the forward diagonal dependence order, each subproblem is exactly a *dynamic monotone staircase matrix* problem as defined by Klawe and Kleitman [14]. In the language of the data structure we gave in the first section, once we have reduced the value at $D[j]$, we never reduce any $D[j']$ with $j' < j$, and once we have computed $E[i]$, we never compute any $E[i']$ with $i' < i$.

Such problems could of course be solved by our more general data structure, which does not depend on the order of reductions and computations; however because of the staircase ordering we can use matrix searching techniques to solve the problem more quickly. If $g(x)$ is convex, the algorithm of Klawe and Kleitman solves the problem for t points in time $O(t\alpha(t))$; here α is the inverse Ackermann function, a very slowly growing function. If $g(x)$ is concave, the algorithm of Wilber [28] solves a single instance of the problem in linear time. However we need to solve many such problems with the inputs to some depending on the outputs of others, and Wilber's analysis breaks down for this case. Eppstein [6] has extended Wilber's algorithm to allow such interleaved computations, while remaining within the linear time bound.

Another possible solution to these monotone staircase problems is to use the algorithms of Galil and Giancarlo [9]. These solve both the convex and concave problems in time $O(t \log t)$, or for functions with the closest zero property in time $O(t)$. However they are much simpler than the matrix searching algorithms, and so even the $O(t \log t)$ version of the algorithms will likely be better than matrix searching in practice.

Each subproblem can be solved independently, including values from the points in A in the order they are needed and as they are available, and computing values for points in B when all the points they depend on have been included. When we split the computation into subproblems, we also keep for each point a list of the subproblems for which that point is in set A; thus when the point's value is computed we need only look at the list to determine which subproblems can proceed in their computation. Along with these subproblem computations, we also proceed as we have said along back diagonals; for each point on a given back diagonal we compute the value as the minimum of the $O(\log n)$ values from the subproblems for which the point is in set B, and then include the computed value in the computations for which the point is in set A.

Let us now summarize the outline of the sparse alignment algorithm in pseudo-code:

```
begin
    find sparse set $X$ of fragments;
    divide-and-conquer by rows to produce subproblems for right influences;
    divide-and-conquer by columns for left influences;
```

```
        for $diag \leftarrow 2$ to $2n$ do
            for $x \in X$ with $row(x) + column(x) = diag$ do begin
                $E[x] \leftarrow +\infty$;
                for subproblem $S$ with $x \in B(S)$ do
                    $E[x] \leftarrow \min(E[x]$, value at $x$ in $S)$;
                compute $D[x]$ from $E[x]$;
                for subproblem $S$ with $x \in A(S)$ do
                    include value of $D[x]$ in $S$;
            end
        end
    end
```

It remains to show that, when the back diagonal computation reaches each point, the subproblems giving the point's value will all have computed their separate minimizations for that point, so that the total value for that point can in fact be computed. In terms of the pseudo-code above, we need to show that each subproblem S with $x \in B(S)$ is ready to supply the value at x when the computation reaches the back diagonal containing point x.

For clarity of explanation, assume the subproblem S is one involving right influences: the assertion for left influence subproblems follows by symmetry. If a point (i, j) in set $B(S)$ for some subproblem S depends on the value at a point (i', j') in set $A(S)$, then clearly $i' < i$ and $j' - i' < j - i$. But then $j' + i' = (j' - i') + 2i' < (j - i) + 2i = j + i$; that is, the back diagonal containing $(i' + j')$ appears before that containing (i, j). Because we process points in order by back diagonals, $D[(i', j')]$ will already have been computed and included in subproblem S. Therefore all subproblem results will in fact be computed in time for them to be combined by the back diagonal computation, and the algorithm correctly computes recurrence 9.

Theorem 5. The problem of sequence alignment from a sparse set of fragments can be solved in time $O(n + m + M \log M \alpha(M))$ for convex gap cost functions, and time $O(n + m + M \log M)$ for concave functions.

Proof: As we have said, the time for each subproblem of size t is $O(t\alpha(t))$ in the convex case, and $O(t)$ in the concave case. The divide and conquer adds a logarithmic factor to these time bounds, giving $O(n + m + M \log M)$ in the concave case, and $O(n + m + M \log M \alpha(M))$ in the convex case. •

If we use the algorithms of Galil and Giancarlo, the bound for fragment alignment with simple functions is $O(n + m + M \log M)$, for both the convex and concave cases. For arbitrary convex and concave functions the time rises to $O(n + m + M \log^2 M)$. However the latter algorithms do not use matrix searching and are therefore likely to be more efficient in practice.

Conclusions

We have described algorithms for two dynamic programming problems, sequence alignment from a sparse set of fragments, and RNA structure prediction. We use the fairly general assumption that an associated cost function in the dynamic programming problems is either convex or concave.

In each case, the dynamic programming matrix is sparse, and our algorithms take advantage of that sparsity. Our algorithms have time bounds that vary almost linearly in the density of the problems. For the sequence alignment problem, the previous solution already assumed sparsity however our algorithm improves on it by almost an order of magnitude. For the RNA structure problem, no sparse solution was previously known. Even when the problem is dense, our algorithms for this problem are no worse than the best known algorithms; when the problem is sparse our time bounds become much better than those of previous algorithms.

References

[1] Alok Aggarwal and Maria M. Klawe, Applications of Generalized Matrix Searching to Geometric Algorithms, SIAM J. Discr. Appl. Math., to appear.

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

[3] Alok Aggarwal and James Park, Searching in Multidimensional Monotone Matrices, 29th FOCS, 1988, 497–512.

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

[5] J.L. Bentley and J.B. Saxe, Decomposable Searching Problems I: Static-to-Dynamic Transformation. J. Algorithms 1(4), December 1980, 301–358.

[6] David Eppstein, Sequence Comparison with Mixed Convex and Concave Costs, manuscript.

[7] David Eppstein, Zvi Galil, and Raffaele Giancarlo, Speeding Up Dynamic Programming, 29th FOCS, 1988, 488–496.

[8] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano, Sparse Dynamic Programming I: Linear Cost Functions, manuscript.

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

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

[11] A.J. Hoffman, On Simple Linear Programming Problems, Convexity, Proc. Symp. Pure Math 7, AMS, 1961, 317–327.

[12] Donald B. Johnson, A Priority Queue in Which Initialization and Queue Operations Take $O(\log\log D)$ Time, Math. Sys. Th. 15, 1982, 295–309.

[13] M.I. Kanehisi and W.B. Goad, Pattern Recognition in Nucleic Acid Sequences II: An Efficient Method for Finding Locally Stable Secondary Structures, Nucl. Acids Res. 10(1), 1982, 265 277.

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

[15] Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching Addison-Wesley, 1973.

[16] Donald E. Knuth and Michael F. Plass, Breaking Paragraphs into Lines, Software Pra[illegible] and Experience 11, 1981, pp. 1119-1184.

[17] D.J. Lipman and W.L. Pearson, Rapid and Sensitive Protein Similarity Searches, Scien[illegible] 1985, 1435–1441.

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

[19] G. Monge, Déblai et Remblai, Mémoires de l'Académie des Sciences, Paris, 1781.

[20] 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, 1970, p. 443.

[21] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman, Algorithms for Loop Matchings. SIAM J. Appl. Math. 35(1), 1978, 68–82.

[22] David Sankoff, Joseph B. Kruskal, Sylvie Mainville, and Robert J. Cedergren, Fast Algorithms to Determine RNA Secondary Structures Containing Multiple Loops, in D. Sankoff and J.B. Kruskal, editors, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983, 93–120.

[23] Robert E. Tarjan, Data Structures and Network Algorithms, SIAM, 1985.

[24] Peter van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time, 16th FOCS, 1975, and Info. Proc. Lett. 6, 1977, 80–82.

[25] Michael S. Waterman and Temple F. Smith, RNA Secondary Structure: A Complete Mathematical Analysis, Math. Biosciences 42, 1978, 257–266.

[26] Michael S. Waterman and Temple F. Smith, Rapid Dynamic Programming Algorithms for RNA Secondary Structure, in Adv. Appl. Math. 7, 1986, 455–464.

[27] Michael S. Waterman, Temple F. Smith, and W.A. Beyer, Some Biological Sequence Matrices, Adv. Math. 20, 1976, 367–387.

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

[29] W.J. Wilbur and D.J. Lipman, Rapid Similarity Searches of Nucleic Acid and Protein Data Banks, Proc. Nat. Acad. Sci. USA 80, 1983, 726–730.

[30] W.J. Wilbur and David J. Lipman, The Context Dependent Comparison of Biological Sequences, SIAM J. Appl. Math. 44(3), 1984, 557-567.

[31] F.F. Yao, Efficient Dynamic Programming Using Quadrangle Inequalities, 12th STOC, 1980, 429–435.