Speeding up Dynamic Programming

David Eppstein [1,2]
Zvi Galil [1,2,3]
Raffaele Giancarlo [1,2,4]

[1] Computer Science Department, Columbia University, New York, USA
[2] Work supported in part by NSF grants DCR-85-11713 and CCR-86-05353
[3] Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel
[4] Dipartmento di Informatica, Universitá di Salerno, I84100 Italy;
Supported by Italian Ministry of Education, Project "Teoria degli Algoritmi"

Summary: A number of important computational problems in molecular biology, geology, speech recognition, and other areas, can be expressed as recurrences which have typically been solved with dynamic programming. By using more sophisticated data structures, and by taking advantage of further structure from the applications, we speed up the computation of several of these recurrences by one or two orders of magnitude. Our algorithms are simple and practical.

Introduction

Dynamic programming is one of several widely used problem-solving techniques. In applying the technique, one always seeks to take advantage of special properties of the problem at hand to speed up the algorithm. There are very few general techniques for speeding up dynamic programming routines and ad hoc approaches seem to be characteristic. The only general technique known to us is due to F. Yao [22]. She considered the following recurrence relations:

$$\begin{aligned} &c(i,i) = 0; \\ &c(i,j) = w(i,j) + \min_{i<k\leq j}\{c(i,k-1) + c(k,j)\}, \text{ for } i < j. \end{aligned} \tag{1}$$

Yao proved that if the weight function w satisfies the *quadrangle inequality*

$$w(k,j) + w(l,j') \leq w(l,j) + w(k,j'), \text{ for all } k \leq l \leq j \leq j', \tag{2}$$

then the obvious $O(n^3)$ algorithm can be sped up to $O(n^2)$. A corollary of this result is an $O(n^2)$ algorithm for computing optimum binary search trees, an earlier remarkable result of Knuth [9].

In this paper we consider the problem of computing two similar recurrences: the one-dimensional case

$$E[j] = \min_{1\leq i<j} D[i] + w(i,j), \tag{3}$$

and the two-dimensional case

$$E[i,j] = \min_{\substack{1\leq i'<i \\ 1\leq j'<j}} D[i',j'] + w(i'+j',i+j) \tag{4}$$

We assume that the values of D are easily computed from the corresponding values of E; as a simple example (which turns out not to be interesting for the cases we consider) we might take $D[i] = E[i]$. To begin the recurrence we are given the value of $E[1]$, or in the two-dimensional case $E[i,1]$ and $E[1,j]$ for all i and j.

Two dual cases arise in applications. In the *concave* case, the weight function w satisfies the quadrangle inequality above. In the *convex* case, the weight function satisfies the inverse quadrangle inequality. We are more concerned with the convex case, but simple modifications of our algorithms solve the concave case as well.

The obvious dynamic programs for the two recurrences above solve them in time $O(n^2)$ and $O(n^4)$ respectively. By useing the assumption of convexity, we reduce the times to $O(n \log n)$ and $O(n^2 \log^2 n)$. If w satisfies an additional property, held by many simple functions such as logarithms and roots, these times can be further reduced to $O(n)$ and $O(n^2 \log n \log\log n)$.

Despite the similarity, Yao's result does not seem to apply in our case. She uses the quadrangle inequality to limit the possible values of the index k which could achieve the minimum in recurrence 1. In our case, no such limit seems to apply; instead we use more sophisticated data structures to achieve our speedup. It should also be noted that our solution of recurrence 4 does not follow from that for recurrence 3. However similar techniques and data structures are used in both cases. We believe that these techniques and data structures will be useful in speeding up other dynamic programs.

0272-5428/88/0000/0488$01.00 © 1988 IEEE

Dimension 1: Sequence Comparison with Gaps

In this section we consider the problem of computing recurrence 3. This problem appears as a subproblem in dynamic programming solutions to various problems. Obviously, it can be solved in time $O(n^2)$, and for a general weight function no better algorithm is possible.

Two dual cases arise in applications. In the *concave* case, the weight function satisfies the quadrangle inequality. In this case we say that w is concave. In the *convex* case, the weight function satisfies the inverse quadrangle inequality, and we say that w is convex. In both cases we show how to use the assumed property of w to derive an $O(n \log n)$ algorithm. Even better, linear-time algorithms are obtained if w satisfies the following, which we call the *closest zero property*: for every two integers l and k, with $l < k$, and for each real number a, the smallest value x such that $x > k$ and $w(l,x) - w(k,x) - a = 0$ can be found in constant time.

Surprisingly, the two algorithms are also dual in the following sense: Both work in stages. In the j-th stage they compute $E[j]$, which is viewed as a competition among indices $1, 2, \ldots, j-1$ for the minimum in recurrence 3. They maintain a set of candidates which satisfies the property that $E[j]$ depends only on $D[k] + w(k,j)$ for k's in the set. Moreover, each algorithm discards candidates from the set, and discarded candidates never rejoin the set. To be able to maintain such a set of candidates efficiently one uses the following dual data structures: a queue in the concave case and a stack in the convex case. The algorithm for the convex case is sketched below; more complete details of both cases can be found in [5].

Notice that in the special case that $D[j] = E[j]$ our problem is the single source shortest path problem for the complete acyclic graph where edge lengths are given by the weight function w. However, neither the convex nor the concave case is interesting, since the quadrangle inequality implies the inverse triangle inequality and the inverse quadrangle inequality implies the triangle inequality. Thus in the convex case $E[j] = D[1] + w(1,j)$ and in the concave case $E[j] = D[1] + w(1,2) + w(2,3) + \cdots + w(j-1,j)$.

We use recurrence 3 to compute various versions of the *modified edit distance* defined 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 single characters from x, inserting single 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. The minimum cost edit distance can be computed by a well known dynamic program in time $O(mn)$.

Notice that a sequence of deletes (inserts) corresponds to a gap in x (y, respectively). 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

$$w(l,k) = f^1(x_l, x_{l+1}) + f^2(x_k, x_{k+1}) + g(k-l). \qquad (5)$$

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. If g is convex (or concave), then w will also be convex (concave). 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 5, and similarly the costs of gaps in y are derived from an analogous weight function w'.

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

$$\begin{aligned} D[i,j] &= \min\{D[i-1,j-1] + s(x_i,y_j), E[i,j], F[i,j]\} \\ E[i,j] &= \min_{0 \le k \le j-1} D[i,k] + w(k,j) \\ F[i,j] &= \min_{0 \le l \le i-1} D[l,j] + w'(l,i) \end{aligned} \qquad (6)$$

with initial conditions $D[i,0] = w'(0,i)$ for $1 \le i \le m$ and $D[0,j] = w(0,j)$ for $1 \le j \le n$.

The obvious dynamic program to solve this recurrence takes time $O(mn \cdot \max(m,n))$. Notice that the computation of $D[i,j]$ reduces to the computation of $E[i,j]$ and $F[i,j]$, and the computation of a row of E and of a column of F are each just the same as the problem discussed above. So if the weight functions w and w' satisfy the inverse quadrangle inequality, we obtain an algorithm that computes the matrix D in $O(mn \log mn)$ time, and even better $O(mn)$ time if the weight functions satisfy the closest zero property.

This dynamic programming scheme arises in the context of sequence comparison in molecular biology [17], geology [19], and in speech recognition [14]. In those fields, the most natural weight functions are convex. In molecular biology, for instance, the motivation for the use of convex weight functions is the following. When a DNA sequence evolves into another by means of the deletion, say, of some contiguous bases, this should be seen as a single event rather than as the combination of many separate deletions of smaller regions. Accordingly, the cost of the contiguous deletion must be less than the total cost of the smaller deletions. Experimental evidence supports this theory [4]. In geology and speech recognition, analogous reasoning motivates the use of convex weight functions.

For the concave case, good algorithms were already known. Hirschberg and Larmore [7] assumed a restricted quadrangle inequality with $k \le l < j \le j'$ in inequality 2 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 [11] with general penalty functions. The algorithm of Hirschberg and Larmore, like our algorithm, uses a queue. Surprisingly, our algorithm, which solves a more general case, is slightly simpler and in many cases faster, as in our algorithm the queue is sometimes emptied in a single operation.

Wilber [21] obtained an ingenious $O(n)$ algorithm, also for the concave case, based on previous work by Aggarwal et al. [1]. His algorithm is recursive and its recursive calls use another recursive algorithm, so the constant factor in the time bound is quite large. Wilber claims that his algorithm is superior to our $O(n \log n)$ one only for n in the thousands.

Miller and Myers [12] independently discovered an algorithm for the convex case which is similar to ours. Their treatment however is considerably more complicated. Klawe [8] has recently found a different algorithm, again using that of Aggarwal et al., which solves the convex case in time $O(n \log^* n)$. She later improved it even further obtaining an $O(n\alpha(n))$ time bound (personal communication). As in the case of Wilber's algorithm, the constant factors involved are large, so Klawe's algorithm is mainly of theoretical interest.

The convex one-dimensional algorithm (sketch)

Let $C(k,r)$ denote $D[k]+w(k,r)$. Given a pair of indices l and k, with $l < k$, let $h(l,k)$ be the minimal index h, with $k < h \leq n$, such that $C(l,h) \leq C(k,h)$; or if no such index exists let $h(l,k) = n+1$. Then it can be shown that $C(l,j') > C(k,j')$ for all j' satisfying $k < j' < h$, and also that $C(l,j') \leq C(k,j')$ for all j' satisfying $h \leq j' \leq n$. If w satisfies the closest zero property, $h(l,k)$ can be computed in constant time. For more general w we may compute $h(l,k)$ in time $O(\log n)$ by using a binary search, taking advantage of the above inequalities.

The list of candidates is represented in a stack S of pairs (k_{top}, h_{top}), (k_{top-1}, h_{top-1}), …, (k_0, h_0). At stage j of the computation, the pairs on the stack will form a set of properly nested intervals around point j. The values of $C(k,j)$ will be monotonically nondecreasing as we go down the stack. Thus the pair (k_{top}, h_{top}) will form the narrowest interval, and $k = k_{top}$ will have the best value of $C(k,j)$ among all candidates for j. When we start to compute points past h_{top}, the candidate from the next pair below the top will have a better value, and this will remain true throughout the computation, so we then pop the stack and no longer consider the old k_{top} as a candidate.

To find the value of $E[j]$ we need only compare $C(k_{top}, j)$ with the value from the newly added candidate $C(j-1,j)$. If $j-1$ is worse, it remains worse for the rest of the computation, so we need not add it to the candidate list. Otherwise, we insert $j-1$ as a new candidate onto the stack, and remove the candidates that because of this insertion can no longer win any of the future comparisons. It can be shown that, after these steps, the stack properties described above will continue to hold for $j+1$, and we can repeat the computation.

Let $K(r)$ and $H(r)$ denote the first and second component of the rth pair from the bottom in stack S. The bottom pair of the stack is a dummy pair, with $H(0) = n+1$; the value of $K(0)$ will vary with the course of the algorithm. The algorithm for solving the convex case of recurrence 3 can be written more formally as follows.

```
begin
    push $(1, n+1)$ onto $S$;
    for $j \leftarrow 2$ to $n$ do begin
        if $C(j-1,j) \geq C(K(top),j)$ then
            $E[j] \leftarrow C(K(top),j)$;
        else begin
            $E[j] \leftarrow C(j-1,j)$;
            while $S \neq \emptyset$ and
                        $C(j-1, H(top)-1) <$
                            $C(K(top), H(top)-1)$ do
                pop $S$;
            if $S = \emptyset$ then
                push $(j-1, n+1)$ onto $S$
            else
                push $(j-1, h(K(top), j-1))$ onto $S$
        end;
        if $H(top) = j+1$ then pop $S$
    end
end
```

Dimension 2: RNA Secondary Structure

In this section we examine recurrence 4, which for convenience we repeat here:

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

The recurrence can be solved by a simple dynamic program in time $O(n^4)$; fairly simple techniques suffice to reduce this time to $O(n^3)$ [20]. In this paper we present a new algorithm, which when w is convex solves recurrence 4 in time $O(n^2 \log^2 n)$. For many common choices of w, a more complicated version of the algorithm solves the recurrence in time $O(n^2 \log n \log\log n)$. Similar techniques can be used to solve the concave case of recurrence 4. Our algorithms do not follow from those of the previous section, and are more complicated, but should still be simple enough for practical application.

The recurrence above has an important application to the computation of RNA secondary structure [13, 20]. After a simple change of variables, one can use it to solve the following recurrence:

$$C[p,q] = \min_{p<p'<q'<q} G[p',q'] + g((p'-p)+(q-q')) \quad (7)$$

Recurrence 7 has been used to calculate the secondary structure of RNA, with the assumption that the structure contains no multiple loops [13]. Our algorithm computes this structure in worst case time $O(n^2 \log^2 n)$, under the realistic assumption that the energy function w of a loop is a convex function of the number of exposed bases in that loop. It is possible to calculate RNA secondary structure with multiple loops, but this seems to require time $O(n^3)$ for linear energy functions, or $O(n^4)$ for general functions [20].

Contention Within a Diagonal

In recurrence 4, we call each of the points (i',j') that may possibly contribute to the value of $E[i,j]$ *candidates*. We consider the computation of $E[i,j]$ as a contest between candidates; the *winner* is the point (i',j') with the minimum value of $D[i',j']+w(i'+j',i+j)$. If we can find a way of eliminating many candidates at once, we can use this to reduce the time of an algorithm for solving recurrence 4.

We say that two points (i,j) and (i',j') in the matrices D or E are on the same *diagonal* when $i+j=i'+j'$. By the *length* of a diagonal we mean the number of points on it; e.g. the longest diagonal in the matrix has length n rather than $n\sqrt{2}$. We say that (k,l) is in the *range* of (i,j) when $k>i$ and $l>j$; that is, when point (i,j) is a candidate for the value of $E[k,l]$.

In this section we describe a way of eliminating candidates within the same diagonal. Using these methods, any given point (i,j) need only compare the values of candidates from different diagonals; there will be only one possible choice for the winning candidate from any given diagonal. In the next section we describe how to compare candidates from different diagonals in order to achieve our time bounds.

In what follows we will assume that the region below a diagonal is a right triangle, having as its hypotenuse the diagonal below the given one, and having as its opposite right angled corner the point (n,n). In fact this region need not be triangular, but if we pretend that our matrices D and E are at the bottom right corner of $2n\times 2n$ matrices we can extend the region to a triangle of the given form (figure 1). This extension will not change the time bounds of our algorithms.

We denote rectangles by their upper left and lower right corners; that is, by the rectangle extending from (i,j) to (i',j') we mean the set of points (x,y) such that $i\le x\le i'$ and $j\le y\le j'$. The range of a point (i,j) is the rectangle extending from $(i+1,j+1)$ to (n,n) (figure 2).

Lemma 1. If (i,j) and (i',j') are on the same diagonal, and if $D[i,j]\le D[i',j']$, then for all (k,l) in the range of both points, $D[i,j]+w(i+j,k+l)\le D[i',j']+w(i'+j',k+l)$. In other words, (i',j') need not be considered as a candidate for those points in the range of (i,j).

Proof: Immediate from the assumption that $i+j=i'+j'$. •

Given a point (i,j) on some diagonal, define the *upper bound* of (i,j) to be the point (i',j') with $i'<i$, with $D[i',j']\le D[i,j]$, and with i' as large as possible within the other two constraints. If there is no such point, take the upper bound to be (n,n). Informally, the upper bound is the closest point above (i,j) on the diagonal that has a lower value than that at (i,j). Similarly, define the *lower bound* to be the point (i'',j'') with $i''>i$, with $D[i'',j'']<D[i,j]$, and with i'' as small as possible, or (n,n) if there is no such point. Note the asymmetry in the above inequalities—we resolve ties in favor of the point that is further toward the top right corner of the matrix.

Observe that, if we order points (i,j) from a single diagonal lexicographically by the pair of values $(D[i,j],i)$ then the result is a well-ordering of the points such that, if (i',j') is a bound of (i,j), then $(i',j')<(i,j)$ in the well-ordering.

Define the *domain* of (i,j) to be the rectangular subset of the range of (i,j), extending from $(i+1,j+1)$ to (i'',j'), where (i',j') is the upper bound of (i,j) and (i'',j'') is the lower bound.

Lemma 2. Each point (i,j) of a given diagonal need only be considered as a candidate for the domain of (i,j). The domains for all points on the diagonal are disjoint and together cover the set of all points below the diagonal (figure 3).

Proof: If a point below row i'' is within the range of (i,j), it is also within the range of (i'',j''), so (i,j) will never win the competition there. Similarly, if a point in the range of (i,j) is after column j' it will be won by (i',j'), or by some other point that is even better.

For any given two points on the diagonal, either one is a bound of the other or there is a bound of one of the two points between the two. Therefore no two domains can overlap. To see that all points below the diagonal are contained in some domain, first note that each such point is contained in the range of some point (i,j). Then the only way it can be removed from that point's domain is if it is also contained in the range of one of the bounds of (i,j). But because of the well-ordering described above we can not continue this process of taking bounds of bounds forever; therefore there must be some point on the diagonal containing the original point in its domain. •

Lemma 3. The domains for each of the points on a diagonal having m total points can be found in time $O(m)$.

Proof: We process the points (i,j) in order by increasing values of i. We maintain a stack of some of the previously processed points; for each point in the stack, the point below it in the stack is that point's upper bound. Each point that we have already processed, but that is no longer in the stack, will already have had its domain computed. No lower bound has yet been reached for any of the points still on the stack. Initially the stack contains (n,n), which is a lower bound for some points on the diagonal but which is itself not on the diagonal.

To process a point (i,j), we look at the point i',j' at the top of the stack. If $(i',j')\ne(n,n)$ and $D[i,j]<D[i',j']$, then (i,j) is a lower bound for (i',j'), so we can pop (i',j') from the stack, compute its domain from this lower bound and the upper bound found at the next position on the stack, and repeat with the point now at the top of the stack. Otherwise, (i,j) is not a lower bound for any stacked points, but (i',j') can be seen to be an upper bound for (i,j), so we push (i,j) on the stack. Finally, when all points have been processed, the points remaining on the stack have (n,n) as their lower bound, so we may pop them one at a time and compute their domains as before.

Each point is pushed once and popped once, so the total time taken by the above algorithm is linear. As we have seen the processing of each point maintains the required properties of the stack, so the algorithm correctly computes the upper and lower bounds, and therefore also the domains.

We now give a more formal description of the algorithm described above. We denote the stack by S. Each position p on the stack consists of three components: $V(p)$, $I(p)$, and $J(p)$. $V(p)$ is the value of D at the point indexed by $I(p)$ and $J(p)$. The stack contains a dummy member at its bottom, which is marked by having a V value of $-\infty$. We use $k = i + j$ to denote the number of the diagonal for which we are computing the domains.

```
begin
    push $(-\infty, n, n)$ onto $S$;
    for $i \leftarrow \max(1, k - n - 1)$ to $\min(k, n - 1)$ do
        begin
        $j \leftarrow k - i$;
        while $V(top) > D[i,j]$ do begin
            $domain(I(top), J(top)) \leftarrow$
                rectangle from
                $(I(top) + 1, J(top) + 1)$
                to $(i, J(top - 1))$;
            pop $S$
        end;
        push $(D[i,j], i, j)$ onto $S$
    end;
    while $V(top) > -\infty$ do begin
        $domain(I(top), J(top)) \leftarrow$
            rectangle from
            $(I(top) + 1, J(top) + 1)$
            to $(n, J(top - 1))$;
        pop $S$
    end
end
```

Convex Weight Functions

In the previous section we described a method for quickly resolving the competition among candidates within a single diagonal; here we will add to this an algorithm for resolving competition among candidates from different diagonals. The competition between diagonals works for any weight function, but here we require the weight function to be convex. We later describe similar techniques for solving the recurrence when the weight function is concave.

We will need for our algorithm a data structure that maintains a partition of the sequence of numbers from 1 through n into intervals. We perform the following operations in the data structure:

(1) Find which interval contains a given number.

(2) Find which interval follows another given interval in the sequence.

(3) Join two adjacent intervals into one larger interval.

(4) Split an interval at a given point into two smaller intervals.

Such a data structure may be implemented at a cost of $O(\log n)$ per operation using balanced search trees [2, 10, 15]. A different algorithm, due to Peter van Emde Boas [16], implements these operations at a cost of $O(\log\log n)$ per operation. In general the simple search tree version of the data structure will be sufficient.

We keep a separate such partition for each row and column of the matrix of the original problem. Each interval in each partition will have a pointer to its *owner*, one of the points (i,j) for which we have already calculated $E[i,j]$. Any point (i,j) may own either some set of row intervals in the row partitions, or some set of column intervals in the column partitions; but no point may own both row and column intervals.

We will maintain as an invariant that, if the owner of the row interval containing point (i,j) is (i_r, j_r), then (i,j) is in the range of (i_r, j_r), and $D[i_r, j_r] + w(i_r + j_r, i + j)$ is the minimum such value among all points (i', j') owning rows. Similarly, the owner (i_c, j_c) of the column interval containing (i,j) is the best point among all points owning columns. When we compute $E[i,j]$ it will be the case for each point (i', j') such that (i,j) is in the range of (i', j'), that either (i', j') owns some intervals or else (i', j') can not be the winning candidate for (i,j). Therefore we may calculate $E[i,j]$ as the minimum between $D[i_r, j_r] + w(i_r + j_r, i + j)$ and $D[i_c, j_c] + w(i_c + j_c, i + j)$, which requires only two queries to the interval data structures, followed by a constant number of arithmetic operations.

It remains to show how to add a point (i,j) to the interval data structures, after $E[i,j]$ has been computed, so that the invariants above are maintained. We will add points a diagonal at a time. First we use the previously described algorithm to compute the domains for each point on the diagonal. Each domain is a rectangle; we cut it into strips, which will be intervals either of rows or columns. We choose whether to cut the domain into row intervals or column intervals according to which direction results in the fewer strips (figure 4). Then we combine each strip with the corresponding row or column partition so that (i,j) ends up owning the subinterval of the strip containing exactly those points for which (i,j) is better than the points previously owning intervals in the partition. First let us compute a bound on the number of strips formed when we cut the domains.

Lemma 4. The total number of strips from a single diagonal is $O(n \log n)$.

Proof: Assume without loss of generality, as in the previous section, that the region to be cut into domains and then strips is a triangle, rather than having its corners cut off. The length of the diagonal of the triangle is at most $2n$.

Let $T(m)$ be the largest number of strips obtainable from a triangle having m elements on the diagonal. As in the proof of lemma 3, the point on the diagonal having the least value has a rectangular domain extending to the corner of the triangle, leaving two smaller triangular regions to be divided up among the remaining diagonal points. Let us say the sides of this outer rectangular domain are $i + 1$ and $j + 1$; then i and j are the diagonal lengths of the smaller triangles, and $i + j = m - 1$. The number of strips formed by this outer domain is the smaller of $i+1$ and $j+1$; without loss of generality we will assume this to be $i+1$. Then

$$T(m) = \max_{\substack{i+j=m-1 \\ i \leq j}} T(i) + T(j) + i + 1. \tag{8}$$

Now assume inductively that $T(k) \leq ck \log k$ for $k < m$ and some constant $c \geq 1$. Let i and j be the indices giving the maximum in equation 8; note that $i < m/2$. By induction, $T(m) = O(m \log m)$ for all m. But the number of strips for any diagonal is certainly no more than $T(2n) = O(n \log n)$. •

Now if we can add a single strip to our row and column partition data structures in logarithmic time, the resulting algorithm will take the time bounds claimed in the introduction. In fact our algorithm may take more than logarithmic time to add a strip, so we cannot bound the time so easily. The result of adding a strip will be the creation of at most two intervals in the partition, so the total number of intervals ever created is proportional to the number of strips. When we add a strip to the partition, we may also remove some intervals from the partition; we will charge the time for this removal to the previous creation of these intervals. Therefore when we add a strip to the data structure we will be charged $O(\log n)$ time for the creation of intervals, and another $O(\log n)$ for those intervals' later removal, for a total time of $O(\log n)$ per strip.

Before we describe how to perform the insertion of a strip, we need the following lemma, which is where we use the assumption that w is convex.

Lemma 5. If w is convex, and if all intervals in the partition of a row (or column) currently belong to points on an earlier diagonal or the same diagonal as that of point (i,j), then if (i,j) is better than the the previous owners for any points in that row (or column) contained within the domain of (i,j), it is better in a single interval starting at the lowest numbered point of the row (or column) in the domain for (i,j).

Proof: We prove the lemma for rows; the proof for columns is the same.

We know from lemma 2 that (i,j) is the best of all points on the diagonal within its own domain. An alternate way of stating the lemma is that, if (i',j') comes from an earlier diagonal and is better than (i,j), then it continues to be better for later points in the row. If this holds, we know that (i,j) is worse than the previous owners of all remaining points in the row, for if it is worse than (i',j') it is certainly worse than any points which have already been found to be better than (i',j').

Let (i'',j'') be the first point in the row for which (i,j) is worse than (i',j'); that is,

$$D[i,j] + w(i+j, i''+j'') \geq D[i',j'] + w(i'+j', i''+j''). \tag{9}$$

Then if $k \geq 1$, point $(i'', j''+k)$ follows (i'',j'') in this row, and $i''+j'' < i''+j''+k$. By assumption $i'+j' < i+j$. Then by convexity of w, the inverse quadrangle inequality

$$\begin{multline} w(i+j, i''+j''+k) - w(i+j, i''+j'') \geq \\ w(i'+j', i''+j''+k) - w(i'+j', i''+j''), \end{multline} \tag{10}$$

holds, and adding equations 9 and 10 gives

$$D[i,j] + w(i+j, i''+j''+k) \geq D[i',j'] + w(i'+j', i''+j''+k), \tag{11}$$

which states that (i,j) continues to be worse than (i',j'). •

We are now ready to describe how to insert the strip for (i,j) into the row (or column) interval partition, once we have calculated the domain as described in the previous section. We first look at the first point (i',j') of the strip, and find the interval containing that point. If the interval owner is at least as good as (i,j) at that point, then (i,j) is never a winner in this strip, and we are done inserting the strip. Otherwise, we split the interval containing (i',j') into two intervals, so that the second of the two starts at (i',j'). The first interval will remain with its original owner, and all or part of the second will eventually become an interval owned by (i,j). But for now we leave both intervals having the same original owner. It may be that when we have finished, more than one interval in the row has the same owner; this is not a problem.

We have found an interval at the start of which (i,j) is better than the other points owning intervals in this row. This interval, which we call the *candidate interval*, is currently owned by some other point; this owner may be better than (i,j) at the other end of the candidate interval. We also need to remember the interval we have already assigned to owner (i,j); for now this is the empty interval.

We repeat the following steps: First we find the next interval following the candidate interval. If this interval starts within the domain of (i,j), and if (i,j) is better than the owner of this new interval at the first point of the interval, then (i,j) must be better than the owner of the candidate interval for all of the candidate interval by lemma 5, and because it is better at the start of the new interval. In this case we merge the candidate interval with the interval owned by (i,j), and set the owner of the merged interval to be again (i,j). We remember the following interval as the new candidate interval, and continue the loop. Otherwise the interval in which (i,j) is best is contained within the candidate interval, so we halt the loop.

We now know that the interval in which (i,j) is best stops somewhere in the candidate interval. The interval needs to be divided into two parts; the first part will be owned by (i,j), and the second part left to its original owner. We find the point at which to split the interval by binary search, at each step comparing the value of $D+w$ for (i,j) with that of the previous owner of the candidate interval. The points searched start at the start of the candidate interval and end either at the end of the interval or the end of the strip, whichever comes first.

At each step of the loop other than the last, an interval is removed from the partition, and we can charge the costs of that step to the interval being removed as mentioned earlier. The costs not charged are a constant number of interval operations, including the creation of a constant number of new intervals, for a cost of $O(\log n)$, and the binary search to find the split point, for another $O(\log n)$. For many functions w we can compute the split point directly, without performing a binary search; if w is such a function we can use the more complicated data structure of van Emde Boas [16] to achieve a time of $O(\log\log n)$ per strip insertion.

The algorithm as a whole proceeds as follows. For each diagonal $i + j = k$, k from 2 to $2n$, we perform the following steps:

(1) Look up the owners of each point (i,j) of the diagonal in the row and column partition data structures, and compute $E[i,j]$ and $D[i,j]$.

(2) Compute the domains of the diagonal points.

(3) For each point (i,j), cut the domain into strips, either by rows or by columns depending on which gives fewer strips, and combine the strips with the appropriate partitions.

Theorem 1. The above algorithm computes the values $E[i,j]$ of recurrence 4 for convex weight functions in total time $O(n^2 \log^2 n)$, or in time $O(n^2 \log n \log\log n)$ for simple weight functions.

Proof: The time taken for each diagonal is as follows. Step 1 takes at most $O(\log n)$ time for each point, for a time of $O(n \log n)$. Step 2 takes time $O(n)$ as described in the previous section. Step 3 takes time $O(\log n)$ time per strip, or $O(\log\log n)$ per strip for simple w. There are $O(n \log n)$ strips, so step 3 takes time $O(n \log^2 n)$ or $O(n \log n \log\log n)$. There are $O(n)$ diagonals, so the total time for the algorithm is $O(n^2 \log^2 n)$, or $O(n^2 \log n \log\log n)$ for simple w. •

Concave Weight Functions

The concave case is somewhat more complex than the convex case. In the convex case, lemma 5 states that, when a strips is added to the collection of strips from earlier diagonals, it remains in one piece, and so it is not so difficult to add the strip to our data structures. Unfortunately the concave equivalent of lemma 5 is that, if a strip were to be added to the collection of strips from *later* diagonals, it would remain in one piece. There are algorithmic techniques that can reverse the order of strip insertion to comply with this lemma [3], but they would add another logarithmic factor to the time used for our algorithm. Instead we replace the partitions of the convex case with the following data structure.

We maintain for each row and column of the matrix a complete binary tree, with the leaves labeled in order from 1 to n. Each vertex in the tree, including the leaves, will contain the name of a strip in the appropriate row or column. If we are processing a given diagonal, then every strip named in each tree will begin on or before that diagonal, so the point in the row or column of the tree that is on the diagonal will be contained in the ranges of the owners of each strip in the tree. We look up the winning candidate at point (i,j) of the tree for column j simply by comparing the owners of each strip at the nodes at and above leaf i in the corresponding tree. Similarly, we look up the winning candidate for row j in its tree. Then we choose the winning candidate at (i,j) to be the minimum of these two candidates.

Finally we must describe how to insert new strips into the data structure so that the above lookup procedure results in the best candidate being found. When we have a newly computed strip, we add it to a queue of strips kept at its start point, and only add it to the tree when the computation of diagonal values reaches that start point. This maintains the property that each remaining point in the row or column is contained in the range of the strip owner.

From this property, and using the assumption of concavity, we can prove that, given two strips, the portions of the row for which the values have yet to be computed, in which each of the two strip owners is better than the other, form two contiguous intervals in the row. In particular, suppose we have two strips, the owners of which are contending for the points in the interval from a to b, and suppose some point x appears in the middle of this interval. Then at most one of the strips can be better both for some points in the interval from a to x, and also for some other points in the interval from $x + 1$ to b. Further, we can determine which of the strips this is in constant time by comparing the values for the two owners at points x and $x + 1$.

Note that these facts are also true in the convex case, so the algorithm described here will also work if the weight function is convex. However in the convex case we can use the simpler algorithms already described.

We insert a strip into the tree as follows. First we try to add it at the root of the tree. If there is not already a strip at that node, we simply insert the new strip there. Otherwise we compare the new and old strips, and find one of them for which the interval in which it is better is contained in the interval corresponding to the left or right subtree under the root. We call this strip the loser; the other strip is the winner. Then we leave the winner as the strip pointed to at the root of the tree, and continue inserting the loser in the appropriate subtree. It can be seen that if we insert strips in this fashion, the lookup procedure given earlier will always return the best strip for the given point.

Theorem 2. The values $E[i,j]$ of recurrence 4 for concave weight functions can be computed in total time $O(n^2 \log^2 n)$.

Proof: Each lookup and insertion takes time bounded by the height of the tree, which is $O(\log n)$. Each point will be looked up once in the tree for its row, and once in the tree for its column, so the total time taken for lookups is $O(n^2 \log n)$. Since there will be at most $O(n^2 \log n)$ strips inserted, the total time taken for insertions is $O(n^2 \log^2 n)$. And the time taken to compute domains is $O(n^2)$ as before. •

Conclusions

We have described simple algorithms that speed up two families of dynamic programs by one and two orders of magnitude. This may be the first time that incorporation of data structures into dynamic programming has yielded such a meaningful speedup.

Our results are a first step in two closely related programs of study. The first is to speed up more dynamic programs, and to determine under what conditions these programs may be sped up. The second is to find efficient algorithms for molecular biology. Since in this area of application the problems typically have very large inputs, theoretical improvements can also have great practical importance. We have identified over a dozen open problems concerning time and space improvements for dynamic programming in molecular biology. One example is reducing the time for the computation of general (multi-loop) RNA secondary structures, alluded to earlier, which seems to represent a three dimensional version of the programs solved in one and two dimensions by the results in this paper.

Another important open problem related to these results is to reduce the space consumed by our algorithms. In particular, the minimum edit distance with affine gap costs was shown by Hirschberg to be computable with only $O(n)$ space, without sacrificing the $O(n^2)$ best known time bound [6]. Unfortunately Z. Galil and Y. Rabani have shown that, in our algorithm for edit sequences with concave or convex gap costs, there are many cases in which the stacks require $\Omega(n^2)$ space. Thus, Hirschberg's techniques are not sufficient to reduce the space in this case.

It is well known that, when the weight function is linear (both convex and concave), the two dimensional dynamic program given by recurrence 4 may be computed in time $O(n^2)$. We have recently discovered that this time bound may also be achieved for a different special case of the recurrence, in which $D[i,j] = E[i,j]$ and the weight function may be either convex or concave. Perhaps this improvement can be extended to other, more general, cases, and in particular to the computation of RNA structure.

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

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

[4] Walter M. Fitch and Temple F. Smith, Optimal Sequence Alignment, Proc. Nat. Acad. Sci., 1983, pp. 1382–1385.

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

[6] Dan S. Hirschberg, A Linear Space Algorithm for Computing Maximal Common Subsequences, Comm. ACM 18, 1975, pp. 341–343.

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

[8] Maria M. Klawe, Speeding Up Dynamic Programming, manuscript.

[9] Donald E. Knuth, Optimum Binary Search Trees, Acta Informatica 1, 1973, pp. 14–25.

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

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

[12] Webb Miller and Eugene W. Myers, Sequence Comparison with Concave Weighting Functions, Bull. Math. Biol., to appear.

[13] 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, pp. 93–120.

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

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

[16] Peter van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time, Proc. 16th Symp. Found. Comput. Sci., 1975.

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

[18] Michael S. Waterman and Temple F. Smith, RNA Secondary Structure: A Complete Mathematical Analysis,

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

[20] 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.

[21] Robert Wilber, The Concave Least Weight Subsequence Problem Revisited, J. Algorithms, to appear.

[22] F. Frances Yao, Speed-up in Dynamic Programming, SIAM J. Alg. Disc. Methods 3, 1982, pp. 532–540.

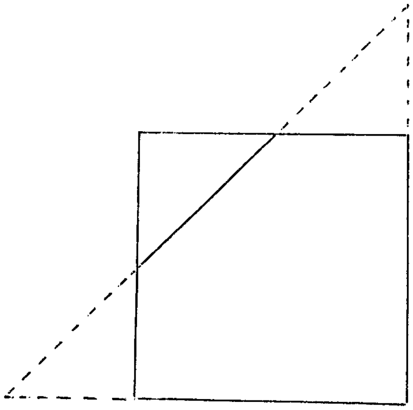

Figure 1. Making the region below a diagonal triangular.

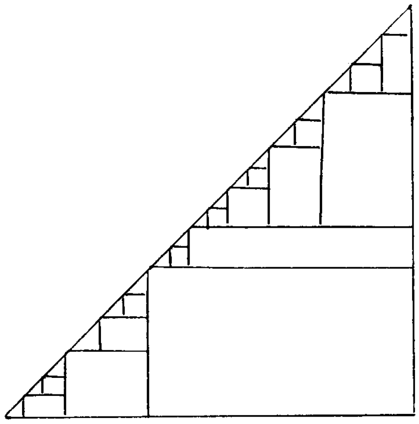

Figure 3. Domains of the points on a single diagonal.

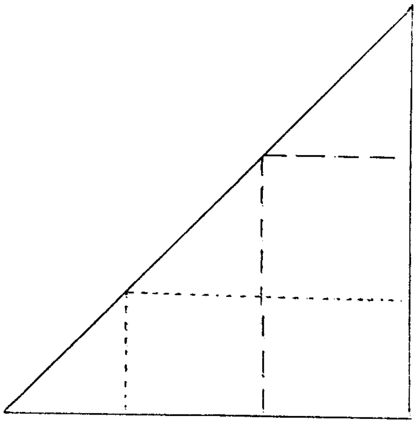

Figure 2. Ranges of points are rectangles extending to (n, n).

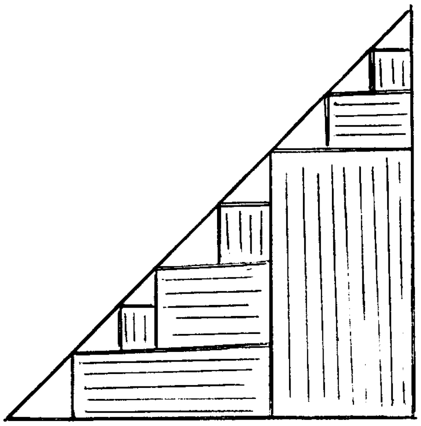

Figure 4. Cutting domains into strips.