Sparse Dynamic Programming*

David Eppstein†
Zvi Galil‡
Raffaele Giancarlo§
Giuseppe F. Italiano¶

Abstract

This paper provides a systematic study of the impact of sparsity on the computation of many different dynamic programming recurrences whose applications include sequence analysis. These recurrences are defined over a number of points that is quadratic in the input size. However, due to natural constraints of the problems, only a sparse set matters for the result. Based on new algorithmic techniques, we obtain algorithms that improve the best known time bounds by a factor almost linear in the density of the problems. All our algorithms are based upon a common unifying methodology.

1 Introduction

Sparsity is a phenomenon that has long been exploited in the design of efficient algorithms. For instance, it is known how to take advantage of sparsity in graphs, and indeed most of the best graph algorithms take time bounded by a function of the number of actual edges in the graph, rather than the maximum possible number of edges. Here we study the impact of sparsity in sequence analysis problems, which are typically solved by dynamic programming in a matrix indexed by positions in the input sequences. In this paper we use the term sequence analysis in a broad sense, to include sequence comparison problems and RNA structure computations in molecular biology. Although sparsity is inherent in the structure of this class of problems, there have been few attempts to exploit it and no general technique for dealing with it is available.

It is well known that a number of important computational problems in computer science, information retrieval, molecular biology, speech recognition, geology and other areas have been expressed as sequence analysis problems. Their common feature is that one would like to find the distance between two given input sequences under some cost assumptions. We are aware of only one problem which is efficiently solvable by previous algorithms taking advantage of sparsity: finding the longest common subsequence of two sequences [12].

The main contribution of this paper is a unifying methodology for designing algorithms which effectively exploits the sparsity of the dynamic programming matrix used for the solution of sequence analysis problems. For each point of the sparse problem we define a *range*, which is a fixed geometric region of the dynamic programming matrix in which that point might influence the values of other points. Different ranges may intersect and therefore compete in the computation of a value of a point lying in their common region. Solving those range conflicts is a nontrivial task even in the dense case, and dealing efficiently with it has required the discovery of new techniques and algorithmic tools [2, 3, 16, 27]. Sparsity introduces further complication to this scenario, which does not seem possible to tackle by means of known techniques. In order to solve these problems, we develop new algorithmic tools and apply several known techniques in a variety of novel ways. For instance, previous algorithms for some of the problems we study have used either data structures [4, 7, 8, 11] or matrix searching [2, 3, 16, 27]. No simple way of combining these two approaches has been known. By finding a way to combine both techniques in a suit-

*Work partially supported by by NSF Grants CCR-86-05353 and CCR-88-14977.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. Research performed while at Columbia University.

‡Columbia University, New York, NY 10027 and Tel Aviv University, Israel.

§Columbia University, New York, NY 10027 and University of Palermo, Italy.

¶Columbia University, New York, NY 10027 and University of Rome, Italy. Partially supported by an IBM Graduate Fellowship.

able framework, we achieve better bounds than either technique alone would give.

Based on our methodology, we show how to improve the solution to nine sequence analysis problems of practical importance. In what follows, the length of the input sequence(s) is denoted by n (and m). M is the number of points in the sparse problem. It is bounded for the sequence comparison problems by nm, and for the RNA structure problems by n^2. For the longest common subsequence problem D is the number of dominant matches (as defined in [10]), which is again bounded by nm. We remark that given the two input sequences, the M points of the sparse problem can be computed in $O(m+n+M)$ by means of string matching techniques. We recall that a function $w(i,j)$ is said to be *concave* if it satisfies the quadrangle inequality as defined by F. F. Yao [29], and it is said to be *convex* if it satisfies the inverse quadrangle inequality [29]. Furthermore, we say that $w(i,j)$ is a *simple concave or convex function* when it is possible to find a zero of the function in constant time. We solve the following problems in the times given.

1. Longest common subsequence: $O(n \log s + D \log\log\min(D, nm/D))$ [1], which improves upon the previous $O(n \log s + m \log n + D \log(nm/D))$ bound [4].

2. Circular longest common subsequence, and circular edit distance with linear cost functions: $O(n + m + M \log M \log\log\min(M, nm/M))$, which improves upon the previous $O(nm \log m)$ bound [15].

3. Circular edit distance with concave cost functions: $O(n + m + M \log^2 M)$.

4. Wilbur-Lipman sequence comparison problem with linear cost functions: $O(n + m + M \log\log\min(M, nm/M))$, which improves upon the previous bounds of $O(M^2)$ [28] and $O(nm)$ [21, 25]. The Wilbur-Lipman method aligns sequences in *fragments* of consecutive symbols; this leads to improved sparsity as well as fewer spurious alignments.

5. Wilbur-Lipman sequence comparison problem with concave or convex cost functions: $O(n + m + M \log M)$ for concave cost functions and $O(n + m + M \log M\, \alpha(M))$ (where $\alpha(M)$ is a functional inverse of Ackermann's function) for convex cost functions, which improves upon the previous $O(M^2)$ bound [28]. For simple convex or concave cost functions the bound reduces to $O(n + m + M \log M)$. The best bounds in the dense case are $O(nm)$ for concave cost functions [6] and $O(nm\alpha(n))$ for convex cost functions [16] as this problem reduces to sequence alignment with gaps [8].

6. RNA structure with linear cost functions for single loops: $O(n + M \log\log\min(M, n^2/M))$, which improves upon the previous $O(n^2)$ bound [14].

7. RNA structure with concave or convex cost functions: $O(n + M \log M \log\min(M, n^2/M))$, or $O(n + M \log M \log\log\min(M, n^2/M))$ for simple cost functions, which always improves upon the $O(n^2 \log n)$ bound of Aggarwal and Park [3] and for sparse sets improves upon the $O(n^2\alpha(n))$ bound of Larmore and Schieber [18].

8. RNA structure with linear cost functions for multiple loops $O(Mn)$, which improves upon the previous $O(n^3)$ bound [23]. The same algorithm can be used to solve within the same time bounds RNA structure with arbitrary cost functions for single loops or RNA loop matching [22].

9. RNA structure with arbitrary cost functions for multiple loops: $O(Mn^2)$, which improves upon the previous $O(n^4)$ bound [26].

The terms of the form $\log\min(M, x/M)$ reduce to $O(1)$ for dense problems. As the above time bounds show, all our algorithms will be asymptotically more efficient than the previous best algorithms when the problems are sparse. In addition, with the only exception of problems 5 and 7, they are always at least as efficient as the best known dense algorithms.

The need for fast and practical algorithms for all the above problems seems to be acute. For instance, the *fastp* program [19] based on Wilbur-Lipman's algorithm [28] is an important tool used daily by molecular biologists for similarity searches in huge protein and nucleic acid data banks. Those data banks (such as GenBank) contain millions of entries and their size is doubling every 2 to 3 years. Improvements to Wilbur-Lipman algorithm are therefore likely to be of practical importance. Most previous attempts

[1] Throughout this paper, we assume that $\log x = \max(1, \log_2 x)$.

to speed up their algorithm are heuristic in nature. Our algorithms greatly reduce the worst case time to solve this problem, while still allowing such heuristics to be performed.

In the remainder of this paper we will concentrate on RNA structure problems and Wilbur-Lipman's problem with linear, convex and concave cost functions. For sake of brevity, some proofs are either sketched or omitted.

2 Sparse RNA Structure

The following recurrence has been used to predict RNA structure [7, 23, 26]:

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

Here D is taken to be some simple function of E. The cost function w is the energy cost of a free loop between the two base pairs. Recurrence 1 can be computed in time $O(n^2)$ when w is linear [14], in time $O(n^2 \log n)$ when w is convex or concave [3] and in time $O(n^3)$ for general functions [7, 26]. The time complexity of these algorithms depends on the length of the input sequence. However, in RNA applications one needs to compute recurrence 1 only for $M \ll n^2$ points of the matrices E and D [23]. Thus it is quite natural to investigate whether bounds depending on M rather than on n can be achieved. We now show how to design algorithms with this feature for linear and then for convex and concave cost functions.

For brevity, let $C(k,l;i,j)$ stand for $D[k,l]+w(k+l, i+j)$. Define the *range* of a point (k,l) to be the set of points (i,j) such that $i > k$ and $j > l$. By the structure of recurrence 1, a point can only influence the value of other points when those other points are in its range. Two points (k,l) and (k',l') can have a non-empty intersection of their ranges.

2.1 Linear Cost Functions

We now outline an $O(n + M \log\log \min(M, n^2/M))$ time algorithm for solving sparse RNA structure when w is a linear function. Without loss of generality, we assume that there are no ties in range conflicts, since they can be broken consistently. In what follows, we say that point (i',j') is *better than* point (i'',j'') for point (i,j) if $C(i',j';i,j) < C(i'',j'';i,j)$. The following fact is useful for the computation of recurrence 1.

Fact 1 *Let (i,j) be a point in the range of both (k,l) and (k',l') and assume that $C(k,l;i,j) < C(k',l';i,j)$. Then, $C(k,l;x,y) < C(k',l';x,y)$ for each point (x,y) common to the range of both (k,l) and (k',l'). In other words, (k,l) is always better than (k',l') for all the points common to the range of both.*

Let $i_1, i_2, \ldots, i_p$, $p \le M$, be the non-empty rows of E and let $ROW[s]$ be the sorted list of column indices representing points for which we have to compute E in row i_s. Our algorithm consists of p steps, one for each non-empty row. During step $s \le p$, the algorithm processes points in $ROW[s]$ in increasing order. Processing a point means computing the minimization at that point, and, if appropriate, adding it to our data structures for later computations. For each step s, we keep a list of active points. A point (i_r, j') is *active* at step s if and only if $r < s$ and, for some maximal interval of columns $[j'+1, h]$, (i_r, j') is better than all points processed during steps $1, 2, \ldots, s-1$. We call this interval the *active interval* of point (i_r, j'). Notice that the active intervals partition $[1, n]$.

Given the list of active points at step s, the processing of a point (i_s, j_q) can be outlined as follows. The computation of the minimization at (i_s, j_q) simply involves looking up which active interval contains the column j_q. The remaining part of processing a point consists of updating the set of active points, to possibly include (i_s, j_q). This is done as follows. Suppose (i_r, j'), $r < s$, supplied the minimum value for (i_s, j_q) in recurrence 1. Then the range of (i_r, j') contains that of (i_s, j_q). By fact 1, if $C(i_r, j'; i_s+1, j_q+1) < C(i_s, j_q; i_s+1, j_q+1)$ then point (i_s, j_q) will never be active. Therefore we do not add it to the list. Otherwise, we must reduce the active interval of (i_r, j') to end at column j_q, and add a new active interval for (i_s, j_q) starting at column j_q. Further, we must test (i_s, j_q) successively against the active points with greater column numbers, to see which is better in their active intervals. If (i_s, j_q) is better, the old active point is no longer active, and (i_s, j_q) takes over its active interval. We proceed by testing against further active points. If (i_s, j_q) is worse, we have found the end of its active interval by fact 1 and this interval is split as described earlier.

The active points and active intervals can be maintained as a balanced search tree [17], sorted by the column numbers of the active points. Processing a

point consists of searching such a tree, then if the point is active adding it to the tree and possibly removing other points from the tree. Using such a tree scheme would yield an $O(M \log M)$ algorithm for the computation of recurrence 1. However, since the values being searched for in the tree are simply fixed column numbers from 1 to n, we may use the *flat tree* data structure described by van Emde Boas [24]. By a simple preprocessing of the list of points, we may remove from consideration those columns not containing any points, and thus reduce the number of columns in the flat tree to at most $O(M)$. This gives a total time of $O(M \log\log M)$. This time can be further improved by carefully exploiting the properties stated in the following lemma.

Lemma 1 *For each nonempty row of matrix E, it is possible to schedule the lookup, insert and delete operations on the list of active points in three different subsequences of homogeneous operations S_1, S_2, S_3 consisting only of lookups, insertions and deletions respectively.*

We now implement the list of active points by using Johnson's improvement to flat trees [13]. We recall that it takes $O(\log\log G)$ time to initialize Johnson's data structure, to insert or delete an item in the data structure, or to look up for the neighbors of an item not in the data structure. Here G denotes the length of the gap between the nearest integers in the structure below and above the item being inserted, deleted or searched for. Using lemma 1 and a novel analysis of Johnson's data structure, we can show:

Theorem 1 *Sparse RNA structure with linear cost functions can be solved in a total of $O(n + M \log\log \min(M, n^2/M))$ time.*

2.2 Convex or Concave Cost Functions

In this subsection we consider the sparse computation of RNA secondary structure given by recurrence 1 when the cost function is either convex or concave.

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 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. 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 row r, the minimum value in equation 1 only depends on the values of other points above r. For points below r, the value of equation 1 is the minimum between the values from points above r, and 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 $\mathcal{A}$ of points above a certain row of the matrix, and a set $\mathcal{B}$ of points below the row. Both sets are sorted by column number. The values of D in recurrence 1 are known for the points in $\mathcal{A}$, and we want to know their contributions to the computation of E for each of the points in $\mathcal{B}$. Each level of the divide and conquer recursion computes the solution to two such problems, one with $\mathcal{A}$ the points above row r and $\mathcal{B}$ the points on row r, and a second with $\mathcal{A}$ the points above or on row r and $\mathcal{B}$ the points below row r.

The "conquer step" of our divide and conquer approach is based upon the efficient solution of the minimization problem with dynamically changing input values given by the following recurrence:

$$E[x] = \min_y D[y] + w(x, y). \qquad (2)$$

Each of the indices x and y are taken from the set of

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

1. Compute the value of $E[x]$, for some index x, as determined by equation 2 from the present values of $D[y]$.
2. Decrease the value of $D[y]$, for some index y, to a new value that must be smaller than the previous value but may otherwise be arbitrary. This change may affect many values of E.

We refer to this problem as the *dynamic minimization problem*. By using modified versions of the algorithm of Galil and Giancarlo [7, 8], it is possible to show:

Lemma 2 *The dynamic minimization problem can be solved in $O(\log N)$ amortized time per operation. For simple cost functions, this time can be reduced to $O(\log \log N)$ amortized time per operation.*

Thus we can show the following theorem.

Theorem 2 *The RNA structure computation of recurrence 1, for a sequence of length n, with M possible base pair, and convex or concave cost functions, can be performed in time $O(n + M \log^2 M)$. For cost functions which satisfy the closest zero property, the computation can be performed in time $O(n + M \log M \log \log M)$.*

Proof (Sketch) : 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 for the dynamic minimization problem. The time per data structure operation in this case is either $O(\log M)$ or $O(\log \log M)$, as described in lemma 2. 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 paid once. The divide and conquer adds another logarithmic factor to this bound. We also need another bucket sort in a preprocessing stage taking time $O(t)$. The total time to solve recurrence 1 is $O(n + M \log^2 M)$ in general, or $O(n + M \log M \log \log M)$ for simple functions. □

In the full version of this paper we will show how these bounds can be further reduced to $O(n + M \log M \log \min(M, n^2/M))$, or for simple functions $O(n + M \log M \log \log \min(M, n^2/M))$. For non-simple functions, we must make a trade off between the matrix searching algorithms of Aggarwal et al. [2] and the dynamic programming algorithms of Galil and Giancarlo [7, 8]; either algorithm alone is not enough to achieve our time bounds. In particular, we need to solve an instance of equation 2, in which there are k values of $D[j]$ given. Each value of $D[j]$ supplies the minima for $E[i]$ with i in some interval of the range from 1 to n, and we need to find the boundaries of these intervals. By using the algorithm of Aggarwal et al., we can solve this problem in time $O(k + n)$. By using that of Galil and Giancarlo, we can solve it in time $O(k \log n)$. We find a way to 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 as the following theorem shows.

Theorem 3 *The RNA structure computation of recurrence 1, for a sequence of length n, with M possible points, and convex or concave cost functions, can be performed in a total of $O(n + M \log M \log \min(M, n^2/M))$ time. For cost functions which satisfy the closest zero property, the computation can be performed in time $O(n + M \log M \log \log \min(M, n^2/M))$.*

3 Sparse Sequence Comparison

In this section we consider Wilbur-Lipman fragment alignment problem. Given two sequences x and y of length m and n, respectively, and a set of subsequences (fragments) from the two sequences, the problem consists of determining the best alignment of x and y by using only the given fragments and by introducing gaps in suitable positions in x and y. A formal definition of this problem is given below. In the special case that the fragments are all possible matching and mismatching symbols from the two input strings this is the edit distance with gaps, for which many efficient algorithms have been proposed [7, 8, 16, 20, 21]. The reader is referred to [28] for a discussion of the advantages of the

Wilbur-Lipman method over edit distance with gaps in molecular biology applications.

Let the two input sequences be denoted $x_1x_2 \ldots x_m$ and $y_1y_2 \ldots y_n$. A *fragment* is defined to be 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, $x_i = y_j$, $x_{i+1} = y_{j+1}$, ..., $x_{i+k-1} = y_{j+k-1}$. The M 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 *diagonal* of a fragment $< i, j, k >$ is the number $j - i$. Then an *alignment* of x and y is defined to be a sequence of fragments such that, if $< i, j, k >$ and $< i', j', k' >$ are adjacent fragments in the alignment, either $< i', j', k' >$ is below $< i, j, k >$ on a different diagonal (a *gap*), or the two fragments are on the same 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 diagonals $w(|(j - i) - (j' - i')|)$.

Wilbur and Lipman obtained the following recurrence [28], which is defined only for triples $< i, j, k >$ giving rise to fragments.

$$D[i, j, k] = -k + \min\{D'[i, j, k], D''[i, j, k]\} \quad (3)$$

where

$$D'[i, j, k] = \min_{<i-l, j-l, k'>} D[i - l, j - l, k'] + \max(0, k' - l)$$

$$D''[i, j, k] = \min_{<i', j', k'> \text{ above } <i, j, k>} D[i', j', k'] + w(|(j - i) - (j' - i')|)\}.$$

Since there are only M fragments, the naive dynamic programming algorithm for this computation, given by Wilbur and Lipman, takes time $O(M^2)$.

We consider recurrence 3 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. From now on we ignore the distinction between these two kinds of points in the matrix, which can be easily dealt with by our algorithms. Thus, we ignore k in recurrence 3 and consider the following two-dimensional subproblem: Compute

$$E[i, j] = \min_{(i', j') \text{ above } (i, j)} D[i', j'] + w(|(j - i) - (j' - i')|), \quad (4)$$

where $D[i, j]$ is an easily computable function of $E[i, j]$. As in the RNA structure computation, each point has a range consisting of the points below and to the left of it. 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, $w(|p - q|) = w(p - q)$ or $w(|p - q|) = w(q - p)$; i.e. the division of the range in two parts removes the complication of the absolute value from the cost function. Indeed, we can write recurrence 4 as

$$E[i, j] = \min\{LI[i, j], RI[i, j]\}, \quad (5)$$

where

$$LI[i, j] = \min_{\substack{(i', j') \text{ above } (i, j) \\ j - i < j' - i'}} D[i', j'] + w((j' - i') - (j - i)) \quad (6)$$

$$RI[i, j] = \min_{\substack{(i', j') \text{ above } (i, j) \\ j' - i' < j - i}} D[i', j'] + w((j - i) - (j' - i')) \quad (7)$$

Both recurrences 6 and 7 look very similar to recurrence 1, except that they must be put together to compute recurrence 4. Thus, the order of computation of the points must be the same for the two recurrences. Moreover, now we have two collections of influences that are eighth-planar geometric regions while in the RNA structure computation we had ranges that were quarter-planar geometric regions. We describe our algorithm for gap cost functions that are linear in the distance between diagonals

in section 3.1, and for convex and concave gap cost functions in section 3.2.

3.1 Linear Cost Functions

We now outline an $O(n + M \log\log \min(M, nm/M))$ time algorithm for computing recurrence 4 when w is a linear function. For each point (i, j) for which recurrence 4 is defined, we perform the minimization separately over the left influences containing (i, j) (recurrence 6) and the right influences containing (i, j) (recurrence 7). The total minimum is then the smaller of the left and right minima. We notice that the computation of recurrence 7 by rows is similar to that of recurrence 1. The only difference is that now ranges are bounded by forward diagonals instead of by columns. However, the algorithm given in section 2.1 can be adapted to solve (7) by rows without any loss in time efficiency. If we could compute recurrence 6 in order by columns, we would get the same time bound given by theorem 1. However, this would conflict with the order of computation of (7) and the two recurrences could not be put together into a single algorithm. Instead we need a slightly more sophisticated approach, so that we can compute (6) in order by rows. This can be done as follows.

Our algorithm can be thought of as building a partition of the matrix LI into geometric regions while proceeding row by row. For each region R in the partition, there is a point (i, j) which is the best for the computation of (6) for points in R. Obviously, R is contained in (i, j)'s left influence. We refer to (i, j) as the *owner* of region R. A point may own more than one region. However, all regions owned by a point are disjoint. The boundaries of each region are composed of rows, forward diagonals and columns. Thus, each region is either a triangle or a convex quadrilateral. A region R is said to be *active* at row i, $1 \leq i \leq m$, if and only if R intersects row i. A point is said to be *active* at row i if it is the owner of at least one active region.

While processing matrix LI by rows, we dynamically maintain the list of active regions. We can think of active regions as keys and active points which own them as the information associated to the corresponding key. Indeed, given the list of active regions at row i, we can compute the minimum in (6) for point (i, j) by finding the active region that contains j. The owner of such region provides the minimum for $LI[i, j]$. The operations that we would like to perform on the list of active regions are inserting a region, deleting a region, splitting a region into two, and lookups. Since the computation is by rows, the possible horizontal boundaries of active regions are not meaningful. Each active region is completely characterized by only two types of non-horizontal boundary (i.e., forward diagonal or column boundaries). As a result, we represent each active region by means of two non-horizontal boundaries.

The main difficulty in efficiently maintaining the list of active regions is that the boundaries of the regions partitioning LI are not known in advance but are actually discovered row by row. Indeed assume that in the computation of (6) we are processing row i. At this step, our algorithm has computed the partition of the matrix LI up to row i, but we do not know the behavior of the currently active regions after row i. It can happen that a new point (i', j'), $i' > i$, contained in a region R active at i may split R into two parts, depending on whether (i', j') is better than the owner of R in their common left influence. In such a case, we wait until row i' before deciding whether R should be split. Furthermore, when we have a region bounded on the left by a forward diagonal and on the right by a column, we must remove it when the row on which these two boundaries meet is processed because such a region cannot be active any longer. At this point we compare the two regions on either side, to see whether their boundary should continue as a diagonal or as a column. Once again, we will decide it when considering the row in which their boundaries meet.

Even though there are two types of boundary, it can be shown that the active regions appear in a linear order for the row we are computing. This order can be maintained under the changes in the set of active regions required by the insertion and by the removal of regions. Therefore we may use a binary search tree to perform the computation in time $O(M \log M)$. Because of the two types of boundary, however, the items being searched for cannot be represented as a single set of fixed integers. Therefore the algorithm sketched above does not seem to benefit directly from the use of the flat trees of van Emde Boas, or Johnson's improvement to flat trees, since the keys in these data structures must be unchanging integers in a fixed range. However, we show that it is possible to use only two Johnson's data structures, one for column boundaries and one for diagonal boundaries. The diagonal boundaries can be represented as

the integer numbers of the diagonals, and the column boundaries can be represented as the integer numbers of the columns. Searching for the region containing a point is then accomplished by finding the rightmost boundary to the left of the point, and choosing among the two resulting column and diagonal boundaries the one that is closer to the point. Moreover, all the insert, delete, split and lookup operations that need to be performed for a given row i have to be scheduled so that we perform a homogeneous sequence of operations at a time.

Thus, we can show that recurrences 6 and 7 can be computed by rows in $O(m + n + M \log\log\min(M, nm/M))$ time when the cost function is linear. This implies the following theorem.

Theorem 4 *Wilbur and Lipman's fragment alignment problem, for sequences of length m and n, with M fragments, and linear cost functions can be solved in $O(m + n + M \log\log\min(M, nm/M))$ time.*

3.2 Concave and Convex Cost Functions

We now outline an algorithm for solving recurrence 4 when the cost function is either concave or convex. The time we achieve is $O(n + m + M \log M)$ for concave cost functions and $O(n + m + M \log M\alpha(M)))$ for convex cost functions. Our algorithm 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 geometric region of the 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 geometric regions, leaving only the forward-diagonal boundaries. The reduced subproblem can then be solved with matrix-searching techniques.

We show now how to compute recurrences 6 and 7, which cut the range of each point into right and left influences. Since those two recurrences must be computed according to the same order, we proceed for both of them by back diagonals. This order is symmetric with respect to the two recurrences, so without loss of generality from now on we need only consider the computation of recurrence 7.

As in the RNA structure computation, we use divide and conquer to produce the subproblems into which we divide the computation of recurrence 7. Using the same terminology as in section 2.2, in each subproblem the points in $\mathcal{A}$ are those above some row and the points in $\mathcal{B}$ are those below the same row. The minimization for point (i, j) in $\mathcal{B}$ depends on the value at a point (i', j') in $\mathcal{A}$ exactly when $j' - i' < j - i$. Thus we order the points in the subproblems by the numbers of their forward diagonals. Such an order can be maintained by initially bucket sorting all points, and then splitting the sorted list at each level of the recursion. Each point will be in set $\mathcal{A}$ for $O(\log n)$ subproblems and set $\mathcal{B}$ 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 $\mathcal{A}$ and $\mathcal{B}$. 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 we then proceed to solve the recurrence, in order by back diagonals as stated above. After we begin solving the recurrence, we maintain each subproblem dynamically. This is done by including the values of matrix D for points in set $\mathcal{A}$ as they become known, and by computing the subproblem minimum for each point in set $\mathcal{B}$ as all the information it depends upon becomes available.

The actual order in which the subproblems receive the values of $D[i', j']$ for points (i', j') in set $\mathcal{A}$ will 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 $\mathcal{B}$ 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 $D[i', j']$ 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 (i, j) in set $\mathcal{B}$, all values $D[i', j']$ for points (i', j') on previous forward diagonals of set $\mathcal{A}$ 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 Aggarwal and Klawe [1]. In the language of the

data structure we gave in the section 2.2 for the dynamic minimization problem, 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 $i' < i$.

If $w(i,j)$ is convex, the algorithm of Klawe and Kleitman [16] 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 $w(i,j)$ is concave, the algorithm of Wilber [27] 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. For more recent results in this area see [9, 18].

Each subproblem can be solved independently, including values from the points in $\mathcal{A}$ in the order they are needed and as they are available, and computing values for points in $\mathcal{B}$ 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 $\mathcal{A}$; 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 $\mathcal{B}$, and then include the computed value in the computations for which the point is in set $\mathcal{A}$.

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 final value for that point can in fact be computed. We need to show that each subproblem S with $(i,j) \in \mathcal{B}(\mathcal{S})$ is ready to supply the value at (i,j) when the computation reaches the back diagonal containing point (i,j). 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 $\mathcal{B}(\mathcal{S})$ for some subproblem S depends on the value at a point (i',j') in set $\mathcal{A}(\mathcal{S})$, 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 3.

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 (Sketch) : 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. □

References

[1] Alok Aggarwal, and Maria M. Klawe, Applications of Generalized Matrix Searching to Geometric Algorithms, Discrete Applied Mathematics, 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] A. Apostolico and C. Guerra, The Longest Common Subsequence Problem Revisited, Algorithmica 2, 1987, 315–336.

[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, J. Algorithms, to appear.

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

[8] Zvi Galil and Raffaele Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, Theor. Comput. Sci. 64, 1989, 107–118.

[9] Zvi Galil and Kunsoo Park, A linear-time algorithm for concave one-dimensional dynamic programming, Inform. Process. Letters, to appear.

[10] D.S. Hirschberg, Algorithms for the Longest Common Subsequence Problem, J. ACM 24, 1977, 341–343.

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

[12] J.W. Hunt and T.G. Szymanski, A Fast Algorithm for Computing Longest Common Subsequences, C. ACM 20(5), 1977, 350–353.

[13] 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.

[14] 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.

[15] Zvi M. Kedem and Henry Fuchs, On Finding Several Shortest Paths in Certain Graphs, 18th Allerton Conf., 1980, 677–686.

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

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

[18] L. Larmore and B. Schieber, On-line dynamic programming with applications to the prediction of the RNA secondary structure, These Proceedings.

[19] D.J. Lipman and W.L. Pearson, Rapid and Sensitive Protein Similarity Searches, Science 2, 1985, 1435–1441.

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

[21] 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.

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

[23] 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.

[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] R. Wagner and M. Fischer, The String to String Correction Problem, J. ACM 21(1), 1974, 168–178.

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

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

[29] F.F. Yao, Speed-up in dynamic programming, SIAM J. Alg. Disc. Methods 3, 1982, 532–540.