Sparse Dynamic Programming I: Linear Cost Functions

David Eppstein
Zvi Galil
Raffaele Giancarlo
Giuseppe F. Italiano

CUCS-471-89

Sparse Dynamic Programming I: Linear 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 a number of different recurrences for sequence comparison and for RNA secondary structure prediction. 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 these problems, when the weight functions used in the recurrences are taken to be linear. Our algorithms reduce the best known bounds by a factor almost linear in the density of the problems: when the problems are sparse this results in a substantial speed-up.

Introduction

Sparsity is a phenomenon that has long been exploited for efficient algorithms. For instance, most of the best known 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. The algorithms we study in this paper perform various kinds of sequence analysis, which are typically solved by dynamic programming in a matrix indexed by positions in the input sequences.

Only two such problems are already known to be solved by algorithms taking advantage of sparsity: sequence alignment [27, 28] and finding the longest common subsequence [5, 12].

In the sequence alignment problem, as solved by Wilbur and Lipman [27, 28], a sparse set of matching fragments between two sequences is used to build an alignment for the entire sequences in $O(n + m + M^2)$ time. Here n and m are the lengths of the two input sequences, and $M \leq nm$ is the number of fragments found. The *fastp* program [16], based on their algorithm, is in daily use by molecular biologists, and improvements to the algorithm are likely to be of practical importance. Most previous attempts to speed up the Wilbur-Lipman algorithm are heuristic in nature, for instance reducing the number of fragments that need be considered. Our algorithm runs in $O(n + m + M \log\log \min(M, nm/M))$[1] time for linear cost functions and therefore greatly reduces the worst case time needed to solve this problem, while still allowing such heuristics to be performed.

The second problem where sparsity was taken into consideration is to determine the longest common subsequence of two input sequences of length m and n. This can be solved in $O(nm)$ time by a simple dynamic program, but if there are only M pairs of symbols in the sequences that match, this time can be reduced to $O((M + n)\log s)$ [12]. Here s is the minimum of m and

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

the alphabet size. The same algorithm can also be implemented to run in $O(n \log s + M \log\log n)$ time. Apostolico and Guerra [5] showed that the problem can be made even more sparse, by only considering *dominant matches* (as defined by Hirschberg [10]); they also reduced the time bound to $O(n \log s + m \log n + d \log(nm/d))$, where $d \leq M$ is the number of dominant matches. A different version of their algorithm instead takes time $O(n \log s + d \log\log n)$. We give an algorithm which runs in $O(n \log s + d \log\log \min(d, nm/d))$ and therefore improves all these time bounds. Longest common subsequences have many applications, including sequence comparison in molecular biology as well as to the widely used *diff* file comparison program [4].

We show also that sparsity helps in solving the problem of predicting the RNA secondary structure with linear cost functions for single loops [23]. We give an $O(n + M \log\log \min(M, n^2/M))$ algorithm for this problem, where n is the length of the input sequence, and $M \leq n^2$ is the number of possible base pairs under consideration. The previous best known bound was $O(n^2)$ [13]. Our bound improves this by taking advantage of sparsity.

In the companion paper [7] we study the case where the cost of a gap in the alignment or of a loop in the secondary structure is taken as either a convex or a concave function of the gap or loop length. In particular, we show how to solve the Wilbur-Lipman sequence comparison with concave cost functions in $O(n + m + M \log M)$ and with convex cost functions in $O(n + m + M \log M \alpha(M))$. Moreover, we give a $O(n + M \log M \log \min(M, n^2/M))$ algorithm for RNA structure with concave and convex cost functions for single loops. This time reduces to $O(n + M \log M \log\log \min(M, n^2/M))$ for many simple cost functions. Again, 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.

The terms of the form $\log \min(M, x/M)$ degrade gracefully to $O(1)$ for dense problems. Therefore all our times are always at least as good as the best known algorithms; when M is smaller than nm (or n^2) our times will be better than the previous best times.

Our algorithms are based on a common unifying framework, in which we find for each point of the sparse problem a *range*, which is a geometric region of the matrix in which that point can influence the values of other points. We then resolve conflicts between different ranges by applying several algorithmic techniques in a variety of novel ways.

The remainder of the paper consists of five sections. In section 2 we present an algorithm for the sparse RNA secondary structure, whose running time will be analyzed in section 3. Section 4 deals with Wilbur and Lipman's sequence alignment problem. In section 5 we describe how to get a better bound for the longest common subsequence problem. Section 6 contains some concluding remarks.

2. Sparse RNA Structure

In this section we are interested in finding the minimum energy secondary structure with no multiple loops of an RNA molecule.

An RNA molecule is a polymer of nucleic acids, each of which may be any of four possible choices: adenine, cytosine, guanine and uracil (in the following denoted respectively by the letters A,C,G and U). Thus, an RNA molecule can be represented as a string over an alphabet of four symbols. The string or sequence information is known as *primary structure* of the RNA. In an

actual RNA molecule, hydrogen bonding will cause further linkage to form between pairs of bases. A typically pairs with U, and C with G. Each base in the RNA sequence will base pair with at most one other base. Paired bases may come from positions of the RNA molecule that are far apart from each other. The set of linkages between bases for a given RNA molecule is known as its *secondary structure*. Such secondary structure is characterized by the fact that it is thermodynamically stable, i.e. it has minimum free energy.

Many algorithms are known for the computation of RNA secondary structure. For a detailed bibliography, we refer the reader to [20]. The common aspect of all these algorithms is that they compute a set of dynamic programming equations. In what follows, we are interested in a system of dynamic programming equations predicting the minimum energy secondary structure with no multiple loops of an RNA molecule. Let $y = y_1 y_2 \dots y_n$ be an RNA sequence and let $x = y_n y_{n-1} \dots y_1$. Waterman and Smith [23] obtained the following dynamic programming equation:

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

where

$$V[i,j] = \min_{0<k<i} D[k,j-1] + w'(k,i) \tag{2}$$

$$H[i,j] = \min_{0<l<j} D[i-1,l] + w'(l,j) \tag{3}$$

$$E[i,j] = \min_{\substack{0<k<i-1 \\ 0<l<j-1}} D[k,l] + w(k+l, i+j). \tag{4}$$

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 [20] for definitions of these terms). For the sake of simplicity and without loss of generality, we assume from now on that k and l in recurrence 4 are bounded above by i and j instead of $i-1$ and $j-1$.

The obvious dynamic programming algorithm solves recurrence 1 for sequences of length n in time $O(n^4)$ [23]: this can be improved to $O(n^3)$ [24]. When w and w' are linear functions of the difference of their arguments, another easy dynamic program solves the problem in time $O(n^2)$ [13]. We discuss this case below. Eppstein et al. [6] considered cost functions satisfying certain convexity or concavity conditions, and found an $O(n^2 \log^2 n)$ algorithm for such costs: this was later improved to $O(n^2 \log n)$ [1]. We treat this case in the companion paper [7].

In recurrence 1, it may be that D is not defined for certain pairs (i,j): for RNA structure this occurs when two bases do not pair. For the energy functions that are typically used, base pairs will not have sufficiently negative energy to form unless they are stacked without gaps at a height of three or more; thus we can restrict our attention to pairs that can be part of such a stack [20]. Further, the RNA structure computation really uses only half of the dynamic programming matrix. These factors combine to greatly reduce the number of possible pairs, which we denote by M, from its maximum possible value of n^2 to a value closer to $n^2/128$. If we required base pairs to form even higher stacks, this number would be further reduced. The computation and minimization in this case is taken only over positions (i,j) which can combine to form a base pair. Such problems can still be solved with the algorithms listed above, by giving a value of $+\infty$ to $D[i,j]$ at the missing

positions. However, the complexity measures of the previous algorithms for the problem do not depend on the number of possible base pairs, but only on the length of the input sequence. We would like to speed up these algorithms by taking more careful advantage of the existence of the missing positions, rather than simply working around them.

Before we start our discussion on how to compute recurrence 1, we outline an algorithm for finding all the M pairs (i, j) for which we have to compute D. Assume that we are interested in finding all pairs stacked without gaps at a height of k or more. This is equivalent to finding all substrings $x_i x_{i+1} \ldots x_{i+l}$ and $x_{j+l} x_{j+l-1} \ldots x_j$ of length k such that x_{i+h} base pairs with x_{j+l-h} for $0 \leq h \leq l$. We obtain such substrings as follows.

We find a sequence x^* complementary to x and in the reverse order; i.e., if x_i is A, C, G, or U we set x^*_{n-i+1} equal to C, A, U, or G respectively. We then find all common substrings of length k between x and x^*. This task, and in fact the more general task of finding all common substrings of length k between two any two sequences x and y (which we will use for the sparse sequence alignment problem), can be performed in linear time by using the suffix tree data structure.

We outline the steps involved in such computation, pointing out the time bound for each of them. The reader is referred to [17, 29] for a definition of suffix tree of a string z, as well as for an algorithm that constructs it in $O(|z| \log s)$, where s is the size of the alphabet. In general s can be taken without loss of generality to be less than $|z|$; however here $s = 6$ (we add two new endmarker symbols to the four possible bases) and so the $\log s$ term vanishes.

We build the suffix tree for string $x\$_1 y\$_2$, where $\$_1$ and $\$_2$ are two different endmarkers which match no symbol of x and y. Each leaf ℓ_i of the tree corresponds to a suffix of the string, starting from position i in the string. Further, every node in the tree has a string associated with it, which is the common prefix of all suffixes corresponding to leaves below the node in the tree. In particular, given two leaves ℓ_i and ℓ_j corresponding to positions in x and y, the least common ancestor of the leaves corresponds to the maximal common prefix of the two leaves, which is the maximum common substring of the two strings starting at positions i and j.

Thus to accomplish our goal we need only find each node u of the tree with the length $l(u)$ of the corresponding string satisfying $l(u) \geq k$; and for each such node find all pairs i and j with $i \leq n$ and $j > n$ (so that i corresponds to a position in x, and j to a position in y), and with u the least common ancestor in the tree of ℓ_i and ℓ_j. The first part of this task, finding nodes with long enough corresponding substrings, is easily accomplished with a pre-order traversal of the suffix tree. We mark these nodes, so that we can quickly distinguish them from nodes with corresponding substrings that are too short.

Next observe that a node u is the least common ancestor of ℓ_i and ℓ_j if, and only if, ℓ_i and ℓ_j descend from different children of u. Thus to enumerate the desired substrings corresponding to u, we need simply take each pair v and w of children of u, such that $v \neq w$, and list pairs (i, j) with ℓ_i a descendant of v with $i \leq n$ and ℓ_j a descendant of w with $j > n$. To speed this procedure we should consider only those v having descendants ℓ_i meeting the condition above, and similarly for w; in this way each pair of children considered generates at least one substring, except for the pairs v, v of which there are linearly many in the tree.

To be able to perform the above computation, at the time we consider node u we must have for each of its children two lists of their descendant leaves, corresponding to positions in the two input

strings. By performing a post-order traversal of the tree, we can list the substrings corresponding to each node u as above, and then merge the lists of leaves at the children of u to form the lists at u ready for the computation at the parent of u.

Thus to summarize the generation of matching substrings, we first compute a suffix tree; next we perform a pre-order traversal to eliminate those nodes corresponding to suffixes that are too short; and finally we perform a post-order traversal, maintaining lists of leaves descended from each node, to generate pairs of positions corresponding to the desired common substrings. The generation of the suffix tree and the pre-order traversal each takes time $O(n)$. The post-order traversal and maintenance of descendant lists also takes time $O(n)$, and the generation of pairs of leaves corresponding to common substrings takes time $O(M)$. Thus the total time for these steps is $O(n + M)$. For arbitrary input strings x and y taken from an alphabet of size s, the time would be $O(n \log s + M)$.

Now let us return to the computation of recurrence 1. As we have said, we assume in this paper that w and w' are linear function of the difference of their arguments, i.e. $w(s,t) = c \cdot (t - s)$ and $w'(s,t) = c' \cdot (t - s)$ for some fixed constants c and c'. In the companion paper [7], we will investigate the case when these weight functions are either convex or concave.

It can be easily shown that $H[i,j]$ and $V[i,j]$ can be computed in constant time for each of the M pairs we are interested in. For instance, in the computation of $H[i,j]$, we simply maintain for each i the value of l, with $l < j$, minimizing $D[i-1,l] - c'l$, which supplies the minimum in recurrence 3; then the minimum for $j+1$ can be found by a single comparison between the previous minimum and $D[i-1,j] - c'j$. Thus the difficulty in the computation is to efficiently compute $E[i,j]$ given the required values of D. We will perform this computation of E in order by rows.

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 4, 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. The following fact is useful for the computation of recurrence 4.

Fact 1: Let (i,j) be a point in the range of both (k,l) and (k',l') and assume that $C(k,l;i,j) \leq C(k',l';i,j)$. Then, $C(k,l;x,y) \leq C(k',l';x,y)$ 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 or equal to (k',l') for all the points common to the range of both.

Proof: The difference

$$\begin{aligned}(D[k,l] + c \cdot ((i+j) - (k+l))) - (D[k',l'] + c \cdot ((i+j) - (k'+l'))) \\ = (D[k,l] - c \cdot (k+l)) - (D[k',l'] - c \cdot (k'+l'))\end{aligned}$$

depends only on (k,l) and (k',l'). •

From now on we will assume that there are no ties in range conflicts, since they can be broken consistently.

For the non-sparse version of the dynamic program, it can be further shown from the above fact that the point (k,l) giving the minimum for (i,j) is either $(i-1, j-1)$ or it is one of the points

giving the minimum at $(i-1, j)$ or $(i, j-1)$. Thus at each point we need only compare three values to find the minimum of recurrence 4. This gives a simple $O(n^2)$ dynamic programming algorithm, first pointed out by Kanehisi and Goad [13]. We now describe how to improve this time bound, when M is less than n^2, by taking advantage of the sparsity of the problem.

Let $i_1, i_2, \ldots, i_p$, $p \leq M$, 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 $s \leq p$, 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. We will see later how to perform this lookup. 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). 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 above.

A detailed description of step s is as follows. Let $ACTIVE$ denote the list of all active points $(leader_1, c_1), (leader_2, c_2), \ldots, (leader_u, c_u)$ during step s. Therefore, each element in $ACTIVE$ is composed of an information field ($leader$) and of a key field ($column$); the meaning of each pair $(leader_l, c_l)$, $1 \leq l < u$, is that the point denoted by the pair has active interval $[c_l+1, c_{l+1}]$. The first and last pair are dummy pairs taking care of boundary conditions. We set $c_1 = 0$, $c_u = n$ and $leader_1 = leader_u = 0$. Moreover, we set $C(leader_1, c_1; i, j) = +\infty$ and $C(leader_u, c_u; i, j) = -\infty$

The list $ACTIVE$ satisfies the following invariants which will be maintained by our algorithm

1. $0 = c_1 < c_2 < \ldots < c_u = n$;
2. $(leader_l, c_l)$, $1 \leq l < u$, has active interval $[c_l+1, c_{l+1}]$;
3. All points (i_r, j_q), $r < s$, not in $ACTIVE$ need not be considered for the computation of I on row i_s and beyond.

For a given j_q in $ROW[s]$, the computation of $E[i_s, j_q]$ is performed as follows. Using the keys in $ACTIVE$, we look up which active interval j_q belongs to. That is, we find an l such that $c_l < j_q \leq c_{l+1}$. If $c_l = 0$, then $E[i_s, j_q]$ does not depend on any of the points processed in previous steps and it is therefore set to the value given by the initial conditions in recurrence 4. If $c_l \neq 0$

$E[i_s, j_q]$ is set to $C(leader_l, c_l; i_s, j_q)$. By definition of active point and the invariants 2 and 3 above, $E[i_s, j_q]$ is correctly computed. We refer to the operation of obtaining for a given j_q in $ROW[s]$ the interval which it belongs to as $LOOKUP(ACTIVE, j_q)$. Namely, $LOOKUP(ACTIVE, j_q)$ returns the largest column number in $ACTIVE$ less than j_q.

Once we have all values of E on row i_s, we can compute the corresponding values of D. Based on these latter values, not all the points in either $ACTIVE$ or $ROW[s]$ may turn out to be active at later steps, because new range conflicts may now arise. We resolve such conflicts between different points by first doing the following for each j_q in $ROW[s]$. Let $(leader_l, c_l)$, $1 \leq l$, be the point that provides the minimum in recurrence 4 for (i_s, j_q). We check whether $C(leader_l, c_l; i_s + 1, j_q + 1) \leq C(i_s, j_q; i_s + 1, j_q + 1)$. If this is the case, we delete j_q from $ROW[s]$ since (i_s, j_q) cannot be active by Fact 1. Otherwise, we check whether $j_q = c_{l+1}$. If this test is negative, we do nothing and (i_s, j_q) remains in $ROW[s]$. If the test is positive, we have that the range of (i_s, j_q) is completely contained in the range of (i_{l+1}, c_{l+1}). Thus, one of the two points must be deleted. Indeed, if $C(leader_{l+1}, c_{l+1}; i_s+1, j_q+1) \leq C(i_s, j_q; i_s+1, j_q+1)$, we delete (i_s, j_q) from $ROW[s]$ since it cannot be active by Fact 1. Otherwise, we will later delete $(leader_{l+1}, c_{l+1})$ from $ACTIVE$.

Let $j_1'', j_2'', \ldots, j_g''$ be the column indices of the surviving points in $ROW[s]$ (listed in sorted order). Starting from j_1'', we discard all the column indices j_q'' immediately following it such that $C(i_s, j_1''; i_s+1, j_q''+1) \leq C(i_s, j_q''; i_s+1, j_q''+1)$. When we find a point q' such that $C(i_s, j_1''; i_s+1, j_{q'}''+1) > C(i_s, j_{q'}''; i_s+1, j_{q'}''+1)$, we stop and repeat the same process for q'. As a result, we discard from $ROW[s]$ all column indices j_p'' such that $C(i_s, j_{p'}''; i_s+1, j_p''+1) \leq C(i_s, j_p''; i_s+1, j_p''+1)$, for some $p' < p$. Again, by Fact 1, all discarded points cannot be active at later steps. The result is a list of points $(i_s, j_1'), \ldots, (i_s, j_f')$ that must all be inserted in $ACTIVE$. We refer to the process of obtaining the sorted list $(i_s, j_1'), \ldots, (i_s, j_f')$ from $(i_s, j_1'').(i_s, j_2''), \ldots, (i_s, j_t'')$ as $REDUCE(ROW[s])$. It is implemented as a simple scan of a sorted list and therefore requires $O(|ROW[s]|)$ time. As a consequence of Fact 1, for each j_q and j_l in $ROW[s]$, $q < l$, we have that

$$C(i_s, j_q; i_s + 1, j_l + 1) > C(i_s, j_l; i_s + 1, j_l + 1). \tag{5}$$

We must now insert into $ACTIVE$ the remaining points in ROW. However, the insertion of such points may cause the deletion of other points in $ACTIVE$. We proceed by first deleting, in increasing order by column, all points in $ACTIVE$ that cannot be active any longer. Then, we insert points from $ROW[s]$. The detection of all points that must be deleted from $ACTIVE$ can be performed as follows. We start with the first column index j_1 in ROW. Let l be such that $c_l < j_1 \leq c_{l+1}$. By "walking" on $ACTIVE$, we find the minimal h, $l < h < n$, such that $C(i_s, j_1; i_s+1; c_h+1) > C(leader_h, c_h; i_s+1; c_h+1)$ and $C(i_s, j_1; i_s+1, c_q+1) \leq C(leader_q, c_q; i_s + 1, c_q + 1)$, $l < q < h$. During this walk, we mark as deletable all points $(leader_q, c_q)$, $l < q < h$ from $ACTIVE$. We repeat the above process with j_2 starting at h, if $j_2 \leq c_h$. Otherwise, we start at an index l such that $c_l < j_2 \leq c_{l+1}$. We iterate through this process with successive indices in $ROW[s]$ and $ACTIVE$ until we reach the end of either list. Then, we remove all deletable points from $ACTIVE$. We refer to the operation of deleting a pair $(leader, c)$ from $ACTIVE$ as $DEL(ACTIVE, c)$. By Fact 1 and inequality 5, all the deleted points cannot be active in any of the subsequent steps.

After this step, all the possible range conflicts between points in ROW and $ACTIVE$ have been examined and solved. Therefore, all the remaining points in $ACTIVE$ and ROW will be active for later computation. Thus we insert in $ACTIVE$ all the points with column index in $ROW[s]$. We refer to each insertion as $INSERT(ACTIVE, j)$.

Let $NEXT(LIST, item)$ denote the operation that returns the element succeeding *item* in $LIST$ and assume that the last element in $ROW[s]$ is a dummy column index, say $n+1$. Moreover, let $APPEND(LIST, item)$ denote the operation that appends *item* at the end of $LIST$. The algorithm discussed above can be formalized as follows:

```
Algorithm SRNA:
ACTIVE ← ((0,0),(0,n));
for s ← 1 to p do begin
      j ← NEXT(ROW[s], φ);
      while j ≠ n + 1 do begin
/* compute E[i_s, j] and decide whether to keep j in ROW[s] */
          jdead ← false;
          (leader, c) ← LOOKUP(ACTIVE, j);
          E[i_s, j] ← D[leader, c] + w(leader + c, i_s + j);
          nextj ← NEXT(ROW[s], j);
          if C(leader, c; i_s + 1, j + 1) ≤ C(i_s, j; i_s + 1, j + 1) then begin
               DEL(ROW[s], j);
               jdead ← true;
          end;
          c ← NEXT(ACTIVE, c);
          if (c = j) and C(leader, c; i_s + 1, j + 1) ≤ C(i_s, j; i_s + 1, j + 1) then
               if jdead = false then
                    DEL(ROW[s], j);
          j ← nextj;
     end;
/* remove from ROW[s] the points no longer able to be active */
     ROW[s] ← REDUCE(ROW[s]);
/* delete elements from ACTIVE */
     j ← NEXT(ROW[s], φ);
     (leader, c) ← LOOKUP(ACTIVE, j);
     (leader, c) ← NEXT(ACTIVE, c);
     OLD ← φ;
     while j ≠ n + 1 and (c ≠ n) do begin
          while C(leader, c; i_s + 1, c + 1) > C(i_s, j; i_s + 1, c + 1) do begin
               APPEND(OLD, c);
               (leader, c) ← NEXT(ACTIVE, c);
          end;
          j ← NEXT(ROW[s], j);
          if j > c then (leader, c) ← LOOKUP(ACTIVE, j);
     end;
     APPEND(OLD, n + 1);
/* delete points in OLD from ACTIVE */
     c ← NEXT(OLD, o);
     while c ≠ n + 1 do begin
```

```
            DEL(ACTIVE, c);
            c ← NEXT(OLD, c);
        end;
    /* insert points from ROW into ACTIVE */
        j ← NEXT(ROW, φ);
        while j ≠ n + 1 do begin
            INSERT(ACTIVE, j);
            j ← NEXT(ROW[s], j);
        end;
    end;
```

Theorem 1. Algorithm **SRNA** correctly computes recurrence 4.

Proof: By induction, using the discussion preceding the algorithm. •

In order to simplify the presentation of algorithm **SRNA**, we have assumed that each column index is an integer between 0 and n. We remark that a slight variation of the same algorithm works correctly if we label column indices to be integers between 0 and $\min(n, M)$. Such a labeling can be clearly obtained in $O(n)$ time.

3. Time complexity

In this section we analyze the running time of algorithm **SRNA**. We must account for a preprocessing phase of $O(n)$, which is also the time we need to read the input. Furthermore, it is easily seen that there are no more than $O(M)$ insertions, deletions and lookup operations on $ACTIVE$ and the rest of the algorithm takes just $O(M)$. Therefore the total time of **SRNA** is $O(n+M+T(M))$, where $T(M)$ is the time required to perform the $O(M)$ insertions, deletions and lookup operation on $ACTIVE$. This time complexity depends on which data structure we use for the implementation of $ACTIVE$.

If $ACTIVE$ is implemented as a binary search tree [3, 15, 21], we obtain an $O(n + M \log M)$ time bound. However, we can obtain a better time bound by exploiting the fact that $ACTIVE$ contains integers in $[0, \min(n, M)]$. Indeed, if $ACTIVE$ is implemented as a flat tree [22] we obtain a bound of $O(n+M \log\log \min(n, M))$, since each operation on $ACTIVE$ costs $O(\log\log \min(n, M))$. Even better, by using the fact that the operations performed on $ACTIVE$ are blocks of either insertions or deletions or lookup operations, we can use Johnson's variation to flat trees [9] to obtain an $O(n + M \log\log \min(M, n^2/M))$ time bound. We now discuss such an implementation as well as its timing analysis; this requires some care and some knowledge of the internal working of Johnson's data structure.

Johnson's priority queue maintains a set of items with priorities that are integers in the interval $\{1, \ldots, n\}$. It takes $O(\log\log G)$ time to initialize the 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, where G is the length of the gap between the nearest integers in the structure below and above the priority of the item being inserted, deleted, or searched for.

We need to know the following facts about Johnson's data structure. The items are kept in n buckets, one for each integer in the domain $\{1, \ldots, n\}$. Each bucket contains items of the

corresponding priority. Non-empty buckets are maintained in a doubly linked list sorted according to the priority.

As for van Emde Boas' flat trees, the idea is to maintain a complete binary tree with n leaves and traverse paths in this tree using binary search. The leaves of the binary tree correspond in a left-to-right order to the items in the priority domain. Each integer in $\{1, \ldots, n\}$ and therefore each bucket defines a unique path to the root of the tree. The length of such paths is at most $O(\log n)$.

These paths are dynamically constructed whenever needed. When an item has to be inserted, a new path segment is added to the tree, while the deletion of an item implies the removal of a path segment. In both cases, the length of the path segment involved is $O(\log G)$ in the worst case. By constructing and visiting just a logarithmic number of nodes in each path segment, we get the $O(\log \log G)$ bounds.

The following lemma was implicit in [9].

Lemma 1. A homogeneous sequence of $k \leq n$ operations (i.e. all insertions, all deletions, or all lookups) on Johnson's data structure requires at most $O(k \log \log(n/k))$ time.

Proof: We first prove that it suffices to consider just sequences of insertions. In fact, k deletions are just the reversal of the corresponding k insertions and therefore require the same time. On the other hand, k lookup operations can be performed by performing the corresponding insertions, then finding the lookup results by inspecting the linked list of buckets, and finally deleting the k inserted items. Thus the total time of k lookup operations is bounded above by the total time of k insertions. In the companion paper [7] we present an alternate proof of the time bound for lookups that does not require the modification of the data structure.

It remains for us to bound the cost of k insertions. Denote by t_i, $1 \leq i \leq k$, the length of the new path added to the data structure because of the i-th insertion. The total cost of k insertions will therefore be $O(\sum_{i=1}^{k} \log t_i)$. Let us now consider the total additional size of the resulting tree after the k insertions, $\sum_{i=1}^{k} t_i$. This will be maximized when the k items to be inserted are equally spaced in the priority domain $\{1, \ldots, n\}$, giving rise to $\sum_{i=1}^{k} t_i \leq k + k \log(n/k)$. By convexity of the log function, $\sum_{1=1}^{k} \log t_i$ is $O(k \log(1 + \log(n/k)))$ and therefore the total cost of k insertions is $O(k \log \log(n/k))$. •

We are now able to analyze the overall time bound of the **SRNA** algorithm.

Theorem 2. Algorithm **SRNA** solves the sparse RNA secondary structure problem in a total of $O(n + M \log \log \min(M, n^2/M))$ time.

Proof: By the above discussion, **SRNA** requires at most $O(n + M + T(M))$ time, where $T(M)$ is the worst-case time of performing the $O(M)$ insertions, deletions and lookup operations in $ACTIVE$. By implementing $ACTIVE$ with Johnson's data structure, $T(M)$ is $O(M \log \log G)$ and therefore $O(M \log \log M)$.

It remains to show that the time complexity of **SRNA** is bounded by $O(n + M \log \log(n^2/M))$. By lemma 1, the total time spent by algorithm **SRNA** on row i, $1 \leq i \leq p$, is $O(M_i \log \log(n/M_i))$, where $M_i \leq n$ denotes the number of points in row i. This gives a total of $O(\sum_{i=1}^{p} M_i \log \log(n/M_i))$ time. Define $\alpha_i = n/M_i$, $1 \leq i \leq p$, for each row i. Then, the total time of **SRNA** is asympotically bounded by $\sum_{i=1}^{p} \sum_{j=1}^{M_i} \log \log \alpha_i$ subject to the constraint $\sum_{i=1}^{p} \sum_{j=1}^{M_i} \alpha_i \leq n^2$. By convexity of the log log function, $\sum_{i=1}^{p} \sum_{j=1}^{M_i} \log \log \alpha_i \leq M \log \log(n^2/M)$.

Therefore, SRNA requires at most $O(n + M \log\log \min(M, n^2/M))$ time. •

4. Wilbur-Lipman Fragment Alignment Problem

In this section we will consider 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 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 $O(nm)$ [19]. 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 $O(n^3)$ [25]. The cost functions that typically arise are convex: for such functions this time has been reduced to $O(n^2 \log n)$ [6, 8, 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 [27, 28] 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. Since the expected number of point insertions, deletions and mutations in the optimal alignment of two random sequences is very low, especially for small alphabets, considering longer subsequences has also the advantage of computing more meaningful alignments.

Let the two input sequences be denoted $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$. 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, $x_i = y_j, x_{i+1} = y_{j+1}, \ldots, x_{i+k-1} = y_{j+k-1}$. Wilbur and Lipman took their set of fragments to be all pairs of matching substrings of the two input strings having some fixed length k. Recall that in the description of the RNA structure algorithm, we gave a procedure for finding all such fragments; they may be found in time $O((n+m)\log s + M)$, where n and m are the lengths of the two input sequences, and s is the number of symbols in the input alphabet; we will assume in our time bounds that this is the procedure used to generate the fragments. However our algorithm for sparse sequence alignment does not require that this procedure be used, and in fact it gives the correct results even when we allow different fragments to have different lengths.

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 $j - i$. 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 diagonal (a *gap*), or the two fragments are on the same diagonal with $i' > i$ (a *mismatch*). Note that with this definition, mismatched fragments may overlap. For instance if x =AUGCUUAGCCUUA and y =AUGGCUUAGAUCUA, a possible alignment of fragments is $f_1 = (1, 1, 3), f_2 = (4, 5, 3), f_3 = (6, 7, 3), f_4 = (11, 12, 3)$, which shows a gap between f_1 and f_2, an overlapping mismatch between f_2 and f_3 and a non-overlapping mismatch between f_3

and f_4. 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 cost of a gap is some function of the distance between diagonals $w(|(j-i)-(j'-i')|)$. 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.

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 [28] 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. Fragments are processed according to 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 being taken care of by the total number of matched symbols in the fragments: if the fragment whose alignment is being computed is $f=(i,j,k)$ and the previous fragment is $f'=(i-l,j-l,k')$, then the number of matched symbols added by f is k if f' and f are non-overlapping and $k-(k'-l)$ otherwise. Therefore, in both cases the number of matched symbols added by f is $k-\max(0,k'-l)$. For instance, in the example given above the number of matched symbols added by f_3 is $3-\max(0,3-2)=2$, while the number of matched symbols added by f_4 is $3-\max(0,3-5)=3$. Formally, we have

$$D(i,j,k) = -k + \min \begin{cases} \min\limits_{(i-l,j-l,k')} D[i-l,j-l,k'] + \max(0,k'-l) \\ \min\limits_{(i',j',k') \text{ above } (i,j,k)} D[i',j',k'] + w(|(j-i)-(j'-i')|) \end{cases} \tag{6}$$

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.

We consider recurrence 6 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 total time of $O(M)$. From now on we will ignore the distinction between the two kinds of points in the matrix, and the complication of

the mismatch computation. Thus, we ignore k in recurrence 6 and consider the following two-dimensional subproblem: Compute

$$E[i,j] = \min_{(i',j') \text{ above } (i,j)} D[i',j'] + w(|(j-i)-(j'-i')|), \tag{7}$$

where $D[i,j]$ is an easily computable function of $E[i,j]$.

As in the RNA structure computation, each point in which we have to compute recurrence 7 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, $w(|p-q|) = w(p-q)$ 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.

For brevity, let $C(i',j';i,j) = D[i',j'] + w(|(j-i)-(j'-i')|)$. We have the following Fact:

Fact 2: Let (i,j) be a point in the left influence (right influence, respectively) of both (k,l) and (k',l') and assume that $C(k,l;i,j) - C(k',l';i,j) \leq 0$. Then (k,l) is always better than (k',l') for the computation of recurrence 7 on all points common to the left influence (right influence, respectively) of both.

Notice that if we had not split the ranges of points into two parts, we could not show such a fact to be true. Since it is similar to the central one used in the RNA structure computation, one would expect that the computation of recurrence 7 (and thus recurrence 6) can be performed along the same lines of recurrence 4. Indeed, we can write recurrence 7 as

$$E[i,j] = \min\{LI[i,j], RI[i,j]\}, \tag{8}$$

where

$$RI[i,j] = \min_{\substack{(i',j') \text{ above } (i,j) \\ j'-i' < j-i}} D(i',j') + w((j-i)-(j'-i')) \tag{9}$$

and

$$LI[i,j] = \min_{\substack{(i',j') \text{ above } (i,j) \\ j-i < j'-i'}} D(i',j') + w((j'-i')-(j-i)). \tag{10}$$

Both recurrences 9 and 10 look very similar to recurrence 4, except that they must be put together to compute recurrence 7. 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.

In what follows, we choose to compute the values at points in order by their rows. As a consequence, we have that the computation of recurrence 9 is the same as (4), the only difference is that here regions are bounded by forward diagonals instead of by columns. That is, algorithm

SRNA can compute recurrence 9 provided that each point (i_r, j), $1 \leq r < s$, is represented in $ACTIVE$ by the pair $(i_r, j - i_r)$ and that each point (i_s, j) is represented by $j - i_s$ in $ROW[s]$.

If we could perform the minimization for left influences in order by columns, we would get that SRNA could be adapted to compute recurrence 10. However this would conflict with the order of computation for right influences. Instead we need a slightly more complicated algorithm, so that we can compute recurrence 10 in order by rows.

We now briefly outline our approach to the computation of (10) by rows. We again maintain a collection of active points, each of which is best for some of the remaining uncomputed points. As a consequence, the matrix LI can be partitioned into geometric regions such that for each region R there is a point (i, j) which is the best for the computation of (10) 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.

However, unlike in the computation of (9), the regions in which such points are best may be bounded also by forward diagonals, according to the boundaries of the left influences. As a result, each region is either a triangle or a convex quadrilateral, since the boundaries of each region are composed of rows, diagonals and columns (see Fig. 1). Furthermore, it is no longer true that each point will own at most one region: when we insert a new point in the set of active points it may split a region into two parts. As a consequence, each point may own more than one region. However, all regions owned by a point are disjoint.

There is one further complication: we do not know in advance the boundaries of these regions, but we actually discover them row by row. Assume that in the computation of (10) 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. In fact, it can happen that a new point (i', j'), $i' > i$, cointained 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. 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.

A region R is said to be *active* at row i, $1 \leq i \leq m$, if and only if R intersects row i. While processing row i, the active points are the owners of active regions. In our computation of (10) by rows, we maintain a set of active regions under the updates required by the insertions and deletions of regions described above.

Even though there are two types of border, it can be shown that the regions appear in a linear order for the row we are computing, and this order can be maintained under the changes in the set of active regions required by the insertion of new points 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 border, however, the points 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, which depend on the points being dealt with being unchanging integers. However, we can use two flat trees, 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 in each flat tree, and choosing among the two resulting column and diagonal boundaries the one that is closer to the point. Thus we may perform the computation of fragment alignment in the same time bounds as for RNA structure computation.

Let $i_1, i_2, \ldots, i_p$, $p < M$, be the non-empty rows of LI in (10). Our algorithm for the computation of LI consists of p steps. At step s, we compute LI for row i_s. Assume that we have q active regions $R_1, R_2, \ldots, R_q$ listed in sorted order of their appearance on row i_s. We keep the owners of these regions in a doubly linked list $OWNER$. The i-th element in $OWNER$ is the owner of R_i. Initially, $OWNER$ contains the dummy point (λ, λ) that owns the whole matrix LI. $OWNER$ implicitly maintains the order in which active regions appear in row i_s.

We maintain the boundaries of the q active regions by means of two sorted lists *C-BOUND* (column boundary) and *D-BOUND* (diagonal boundary). Each element in *C-BOUND* is a pair (*rightr*,c), where *rightr* is a pointer to an element in $OWNER$ and c is a column number. The meaning of such pair is that column c is the boundary of two active regions. The region whose owner is pointed to by *rightr* is to the right of c. Pairs are kept sorted according to their column number.

Similarly, each element in *D-BOUND* is a pair (*abover*,d), where *abover* is a pointer to an element in $OWNER$ and d is a diagonal number. The meaning of such pair is that diagonal d is the boundary of two active regions. The region whose owner is pointed to by *abover* is above d. Pairs are kept sorted according to their diagonal number.

We notice that given two adjacent column boundaries in *C-BOUND*, it may happen that the two regions bounded by those columns are not adjacent since regions bounded by diagonals may be in between these two regions. A similar thing may happen to adjacent diagonal boundaries in *D-BOUND*. Thus, we need to use both data structures to locate in which active region a point (i_s, j_l) falls.

We also keep lists $INTERSECT[r]$, $1 \leq r \leq m$, for each row of LI. Such lists contain points in which we must resolve a conflict between the two active regions meeting at that point. At step s, we maintain the invariant that $INTERSECT[r] = \phi$, $1 \leq r \leq i_s - 1$. A column index c is in $INTERSECT[c-d]$, $c - d > i_s$, if and only if c is left boundary of an active region R and the region to the left of R has diagonal d as its bottom boundary. We refer to the point $(c-d, c)$ as an *active* intersection point. Equivalently, $(c-d, c)$ is an active intersection point if and only if c and d are boundaries of two active regions and neither c intersects another diagonal boundary nor d intersects another column boundary in any row from i_s to $c - d - 1$.

Each column c can be in at most one intersection list. We assume that each column in some intersection list has a pointer to the item representing it in such list. We also assume that each diagonal has a flag which is on when that diagonal is involved in an active intersection point. However, we will ignore the details of the update of the intersection lists and of the corresponding pointers for columns and flags for diagonals.

Assuming that $INTERSECT[r] = \phi$, $1 \leq r \leq i_s - 1$ and that *C-BOUND* and *D-BOUND* correctly represent the active regions at step s, we locate the active region containing (i_s, j_l) as

follows. We find a pair $(rightr_u, c_u)$ in *C-BOUND* such that $c_u < j_l \leq c_{u+1}$ and we find a pair $(above r_v, d_v)$ in *D-BOUND* such that $d_v \leq j_l - i_s < d_{v+1}$. Consider the row $c_u - d_v$ in which column c_u and diagonal d_v intersect. If $c_u - d_v \geq i_s$, then (i_s, j_l) belongs to the region owned by the element in *OWNER* pointed to by $rightr_u$ (see Fig. 2a.) since column c_u "hides" the region bounded from below by d_v as well as the regions bounded by columns preceding c_u. If $c_u - d_v < i_s$, then (i_s, j_l) falls into the region owned by the element in *OWNER* pointed to by $above r_v$ (see Fig. 2b.) since diagonal d_v "hides" the region bounded by c_u as well as the regions bounded by diagonals preceding d_v.

We refer to the process of finding pairs in *C-BOUND* (*D-BOUND*, respectively) for a given (i_s, j_l) in $ROW[s]$ as $LOOKUP(C\text{-}BOUND, j_l)$ ($LOOKUP(D\text{-}BOUND, j_l)$, respectively).

We also denote the process of computing the owners of regions containing given (i_s, j_l) in $ROW[s]$ as $WINNER(j_l)$. Given the results of the two *LOOKUP* operations, $WINNER(j_l)$ can be performed in constant time. Moreover, based on the results of *WINNER*, we can compute recurrence 10 and then recurrence 6 (via (8)) in constant time for (i_s, j_l).

After computing D for row i_s, not all points on this row may turn out to generate active regions. Indeed, assume that (i_r, j_q) provided the minimum in recurrence 10 for (i_s, j_l). The left influence of (i_r, j_q) totally contains the left influence of (i_s, j_l). If $C(i_r, j_q; i_s + 1, j_l + 1) \leq C(i_s, j_l; i_s + 1, j_l + 1)$, we can discard (i_s, j_l) since, by Fact 2, it will never own an active region. Consequently, we discard all points in row i_s that are dominated by the owners of their active regions. We refer to this process as $REDUCE(ROW[i_s])$. It produces a sorted subsequence of the column indices in $ROW[s]$. Once we know the values of D for points in row i_s and the outcomes of the *LOOKUP* operations, *REDUCE* can be implemented in $O(|ROW[s]|)$ time.

We must now show how to update *C-BOUND* and *D-BOUND* so as to include the boundaries of active regions owned by points in $ROW[s]$ that survived *REDUCE*. Moreover, we must update *OWNER*. The insertion of these boundaries may also cause the insertion of new active intersection points and the deletion of old ones. Thus, we have to update such lists as well.

Assume that we have correctly processed the first $l - 1$ points in $ROW[s]$ and let (i_s, j_l) be the next point to be processed. Let $(rightr_h, c_h)$ and $(rightr_{h+1}, c_{h+1})$ be two pairs in *C-BOUND* such that $c_h < j_l \leq c_{h+1}$. Similarly, let $(above r_k, d_k)$ and $(above r_{k+1}, d_{k+1})$ be two pairs in *D-BOUND* such that $d_k \leq j_l - i_s < d_{k+1}$. All these pairs can be found by means of *LOOKUP* operations. We now proceed as shown in the following cases.

a. Point (i_s, j_l) falls in the region owned by the point in *OWNER* pointed to by $rightr_h$. Let this point be (i_r, c_h). We distinguish two subcases: $c_{h+1} > j_l$ and $c_{h+1} = j_l$.

   a.1 $c_{h+1} > j_l$. The region owned by (i_r, c_h) must be split into two and the region owned by (i_s, j_l) must be inserted between them (see Fig. 3a.). Thus, we generate two new entries for *OWNER*, i.e. (i_s, j_l) and (i_r, c_h), and we insert them (in the order given) in *OWNER* immediately after the entry pointed to by $rightr_h$. We insert also the pair (γ, j_l) in *C-BOUND* and the pair $(\sigma, j_l - i_s)$ in *D-BOUND*, where γ points to (i_s, j_l) and σ points to the new occurrence of (i_r, c_h) in *OWNER*. The insertion of the region owned by (i_s, j_l) may cause the creation of an active intersection point, i.e. $(c_{h+1} - (j_l - i_s), c_{h+1})$. Indeed, if c_{h+1} is not in any intersection list, we insert it in $INTERSECT[c_{h+1} - (j_l - i_s)]$.

a.2 $c_{h+1} = j_l$. Point (i_s, c_{h+1}) falls on the border of two active regions, one owned by (i_r, c_h) and the other owned by $(i_{r'}, c_{h+1})$, where this latter point is pointed to by $rightr_{h+1}$ in $OWNER$ (see Fig. 3b.). We know that $C(i_{r'}, c_{h+1}; i_s+1, c_{h+1}+1) \leq C(i_r, c_h; i_s+1, c_{h+1}+1)$ and that $C(i_s, c_{h+1}; i_s+1, c_{h+1}+1) \leq C(i_r, c_h; i_s+1, c_{h+1}+1)$. We have to establish whether $(i_{r'}, c_{h+1})$ is better than (i_s, c_{h+1}) in the left influence of this latter point. If this is the case, we do nothing. Otherwise, (i_s, c_{h+1}) conquers part of $(i_{r'}, c_{h+1})$'s left influence. The border between these two regions is diagonal $c_{h+1} - i_s$. Accordingly, we insert the entry (i_s, c_{h+1}) in $OWNER$ immediately before the entry pointed to by $rightr_{h+1}$. We insert also $(\sigma, j_l - i_s)$ in *D-BOUND*, with $\sigma = rightr_{h+1}$ and set $rightr_{h+1}$ in *C-BOUND* to point to the newly inserted entry in $OWNER$. The insertion of the region owned by (i_s, c_{h+1}) may cause the creation of an active intersection point, i.e. $(c_{h+2} - (c_{h+1} - i_s), c_{h+2})$. Indeed, if c_{h+2} is in no intersection list, we insert it in $INTERSECT[c_{h+2} - (c_{h+1} - i_s)]$.

b. Point (i_s, j_l) falls in the region owned by the point in $OWNER$ pointed to by $above_k$. Let this point be $(i_r, d_{k'} + i_r)$, $k' > k$. We have three subcases: $d_k < j_l - i_s$ and $c_{h+1} > j_l$; $d_k = j_l - i_s$ and $c_{h+1} > j_l$; $c_{h+1} = j_l$.

b.1 $d_k < j_l - i_s$ and $c_{h+1} > j_l$. The region owned by $(i_r, d_{k'} + i_r)$ must be split into two and the region owned by (i_s, j_l) must be inserted in between them (see Fig. 4a.). The details for the corresponding update of $OWNER$ are analogous to the ones reported in case (a.1) and are left to the reader. The insertion of the region owned by (i_s, j_l) can cause the creation of two active intersection points, $(c_{h+1} - (j_l - i_s), c_{h+1})$ and $(j_l - d_k, j_l)$, and the deletion of a possibly active intersection point, $(c_{h+1} - d_k, c_{h+1})$. Indeed, if column c_{h+1} is in $INTERSECT[c_{h+1} - d_k]$ we delete it from there and insert it in $INTERSECT[c_{h+1} - (j_l - i_s)]$. Finally, we must insert column j_l in $INTERSECT[j_l - d_k]$

b.2 $d_k = j_l - i_s$ and $c_{h+1} > j_l$. Point (i_s, j_l) falls on the border between two regions, one owned by $(i_r, d_{k'} + i_r)$ and the other owned by $(i_{r'}, d_k + i_{r'})$, where this latter point is the immediate predecessor of the element in $OWNER$ pointed to by $above_k$ (see Fig. 4b.). If $C(i_{r'}, d_k + i_{r'}; i_s + 1, j_l + 1) \leq C(i_s, j_l; i_s + 1, j_l + 1)$, we can discard (i_s, j_l) by Fact 2. Otherwise, we insert the point (i_s, j_l) in $OWNER$ immediately to the left of the element pointed to by $above_k$. This is equivalent to creating a new active region. We insert the pair (γ, j_l) in *C-BOUND*, where γ points to the newly inserted element. The insertion of this new region may cause the creation of an active intersection point, $(j_l - d_{k-1}, j_l)$. Indeed, if diagonal d_{k-1} has its flag off, we must insert j_l in $INTERSECT[j_l - d_{k-1}]$.

b.3 $c_{h+1} = j_l$. This case is analogous to case (a.2).

We notice that at most a constant number of lookups, insertions and deletions in *C-BOUND* and *D-BOUND* is performed. Furthermore, the sum of the time taken by all the other operations involved in the corresponding update of $OWNER$ and the intersection lists adds up to a constant. We have the following lemma.

Lemma 2. The total number of active regions is at most $2M$.

Proof: Each point (i, j) inserted for the first time in $OWNER$ introduces a new active region and splits an old one into two. Since there are at most M points that can be inserted in $OWNER$, the bound follows immediately. •

In order to finish step s, we must process all active intersection points in between rows i_s and $i_{s+1} - 1$. Assume we have processed intersection lists for rows $i_s, \ldots, t-1$. Here we show how to process $INTERSECT[t]$, $t < i_{s+1}$.

If $INTERSECT[t]$ is empty, we ignore it. Thus, let $INTERSECT[t] \neq \phi$. We first bucket sort the indices (column numbers) in such list. Proceeding in increasing order, we find $(rightr, j_q)$ in *C-BOUND* and $(above r, j_q - t)$ in *D-BOUND* for each j_q in $INTERSECT[t]$. This can be performed using $LOOKUP$. As a result, we obtain two sorted lists of pairs, one from *C-BOUND* and the other from *D-BOUND*. We process these lists in increasing order taking a pair from each list. Assume that we have processed the first $l-1$ pairs in both lists. This corresponds to having processed the first $l-1$ points in $INTERSECT[t]$. We now show how to process $(rightr, j_l)$ and $(above r, j_l - t)$. This is equivalent to processing (t, j_l).

Since (t, j_l) is an active intersection point, three active regions meet there (see Fig. 5). Namely, the active region having diagonal $j_l - t$ as an upper boundary, let it be R'', the active region having $j_l - t$ as lower boundary, let it be R', and the active region having column j_l as its left boundary, let it be R. Moreover, let $(i_{r''}, j_l - t + i_{r''})$, $(i_{r'}, c')$ and (i_r, j_l) be the owners of regions R'', R' and R, respectively. We can find those points in $OWNER$ by using either *rightr* or *above r*.

R' cannot be active any more since $(i_{r'}, c')$ is worse than $(i_{r''}, j_l - t + i_{r''})$ ((i_r, j_l), respectively) for points in R'' (R, respectively). We delete its owner from $OWNER$. Next, we have to decide whether R'' gets extended to the right of column j_l.

If $C(i_r, j_l; t+1, j_l+1) \leq C(i_{r''}, j_l - t + i_{r''}; t+1, j_l+1)$, R'' does not extend to the right of j_l. Thus, we remove $(above r, j_l - t)$ from *D-BOUND* since $j_l - t$ is not bottom boundary of any region. The removal $j_l - t$ may cause the creation of a new active intersection point between column j_l and some diagonal boundary d, $d < j_l - t$. It may also cause the deletion of one active intersection point. This involves the update of intersection lists with row number greater that t. The possible intersection points to be inserted or deleted can be easily located as explained above. Each insertion in the intersection lists can be accomplished in constant time. As for the deletions, we defer the actual removal of the items from the intersection list to the time when the list is considered and bucket sorted. This will give a constant amortized time complexity also for each deletion.

Otherwise, R'' gets extended to the right of j_l. We set $above r = rightr$ and delete $(rightr, j_l)$ from *C-BOUND* since R'' and R now share a diagonal boundary. Again, the removal of column j_l may create a new active intersection point between diagonal $j_l - t$ and some column boundary c, $c > j_l$. Again, this involves the update of intersection lists with row number greater than t, which can be accomplished as explained above.

We remark that the bucket sorting of $INTERSECT[t]$ is not really required for its processing. Indeed, there is a more complicated processing of the points in $INTERSECT[t]$ that avoids the bucket sorting of the points. However, it achieves no gain in time complexity.

We have the following lemma.

Lemma 3. The total number of active intersection points is bounded above by $4M$.

Proof: The algorithm creates active intersection points either when inserting a point in $OWNER$ for the first time or when processing an active intersection point. Each new point inserted in $OWNER$ can create at most 2 active intersection points. Thus, no more than $2M$ active intersection points can be created while updating $OWNER$.

Each new active intersection point introduced during the processing of intersection lists may be amortized against an active region being deleted. Thus, by lemma 2, no more than $2M$ new active intersection points may be created during this phase. •

Let $ITEM(LIST, pointer)$ and $PREVIOUS(LIST, pointer)$ denote the operations that return the item in $LIST$ pointed to by *pointer* and the item in $LIST$ preceding it, respectively. Furthermore, let $ESSENTIALS$ be a list which contains all the boundaries of active regions generated by points in $ROW[s]$. The above algorithm can be formalized as follows.

```
Algorithm Left Influence:
OWNER ← (λ, λ);
for s ← 1 to p do begin
    j ← NEXT(ROW[s], φ);
    while j ≠ n + 1 do begin
/* compute LI[i_s, j] and decide whether to keep j in ROW[s] */
        (rightr, c) ← LOOKUP(C-BOUND, j);
        (abover, d) ← LOOKUP(D-BOUND, j);
        (i, cl) ← WINNER(j);
        LI[i_s, j] ← D[i, cl] + w((cl − i) − (j − i_s));
        if C(i, cl; i_s + 1, j + 1) ≤ C(i_s, j; i_s + 1, j + 1) then
            DEL(ROW[s], j)
        else APPEND(ESSENTIALS, (rightr, c), (abover, d));
        j ← NEXT(ROW[s], j);
    end;
/* insert the boundaries of the active regions owned by points in ROW[s] */
    j ← NEXT(ROW[s], φ);
    (rightr, c) ← NEXT(ESSENTIALS, φ);
    (abover, d) ← NEXT(ESSENTIALS, (rightr, c));
    while j ≠ n + 1 do begin
        if ITEM(OWNER, rightr) = WINNER(j) then
            update C − BOUND, D − BOUND, OWNER,
                and INTERSECT following case (a);
        else update C − BOUND, D − BOUND, OWNER,
                and INTERSECT following case (b);
        end;
        j ← NEXT(ROW[s], j);
        (rightr, c) ← NEXT(ESSENTIALS, (abover, d));
        (abover, d) ← NEXT(ESSENTIALS, (rightr, c));
    end;
/* remove active intersection points between rows i_s and i_{s+1} */
    for t ← i_s to i_{s+1} − 1 do begin
        if INTERSECT[t] ≠ φ then begin
            INTERSECT[t] ← BUCKETSORT(INTERSECT[t]);
            j ← NEXT(INTERSECT[t], φ);
            while j ≠ n + 1 do begin
                (rightr, c) ← LOOKUP(C-BOUND, j);
                (abover, d) ← LOOKUP(D-BOUND, j);
                APPEND(ESSENTIALS, (rightr, c), (abover, d));
                j ← NEXT(INTERSECT[t], j);
```

```
                end;
                APPEND(ESSENTIALS,(λ, n + 1))
                (rightr, c) ← NEXT(ESSENTIALS, o);
                (abover, d) ← NEXT(ESSENTIALS,(rightr, c));
                while (rightr, c) ≠ (λ, n + 1) do begin
                    (i'', c'') ← PREVIOUS(OWNER, abover);
                    (i, j) ← ITEM(OWNER, rightr);
/* remove the region no longer active */
                    DEL(OWNER, abover)
                    if C(i, j;t + 1, j + 1) ≤ C(i'', c'';t + 1, j + 1) then begin
                        DEL(D-BOUND, j − t);
                        update intersection lists;
                    end else begin
                        DEL(C-BOUND, j);
                        update intersection lists;
                    end;
                    (rightr, c) ← NEXT(ESSENTIALS,(abover, d));
                    (abover, d) ← NEXT(ESSENTIALS,(rightr, c));
                end;
            end;
        end;
    end;
```

We have the following theorem.

Theorem 3. Algorithm **Left Influence** correctly computes recurrence 10.

Proof: By induction, using the discussion preceding the algorithm. •

The time complexity can be analyzed as follows.

Theorem 4. Wilbur and Lipman's fragment alignment problem can be solved in a total of $O(m + n + M \log\log \min(M, nm/M))$ time.

Proof: The problem can be solved by computing recurrence 6. As we mentioned, this can be reduced to the computation of (8), (9) and (10). Recurrence 9 can be computed using algorithm **SRNA** and therefore by theorem 2 in $O(M \log\log \min(M, nm/M))$ time.

To bound the overall time required to compute recurrence 10, we need to analyze algorithm **Left Influence**. By the above discussion, the time required by this algorithm is $O(m + n + M + T(M))$, where $T(M)$ is the total time required to maintain the lists *C-BOUND* and *D-BOUND* and to bucket sort each *INTERSECTION* list.

By lemma 2 and by lemma 3, there can be at most $O(M)$ insertions, deletions and lookup operations in *C-BOUND* and *D-BOUND*. Furthermore, **Left Influence** requires that for each row at most a constant number of homogeneous sequences of these operations (i.e., all insertions, all deletions, or all lookups) is performed. If we use Johnson's data structure to support them, an argument completely analogous to the proof of theorem 2 gives a total of $O(M \log\log \min(M, nm/M))$ time.

As for bucket sorting the *INTERSECTION* lists, assume there are c_i points to bucket sort at row i, $1 \leq i \leq m$. If we use again Johnson's data structure [9], this can be done

in $O(c_i \log\log \min(M, n/c_i))$. Therefore the total time is $O(\sum_{i=1}^{m} c_i \log\log \min(M, n/c_i))$. By lemma 3, $\sum_{i=1}^{m} c_i \leq 4M$. Again, a total of $O(M \log\log \min(M, nm/M))$ time results by convexity of the log log function.

Once the value of $LI[i,j]$ and $RI[i,j]$ are available, the computation of $E[i,j]$ and $D[i,j]$ can be performed in constant time.

Therefore the total time required to solve the fragment alignment problem is $O(m + n + M \log\log \min(M, nm/M))$. •

5. The Longest Common Subsequence Problem

In this section we describe how to solve efficiently the longest common subsequence problem. We assume that the reader is familiar with the algorithms of Apostolico and Guerra [5].

Recurrence 4, used for the computation of RNA structure with linear loop cost functions, can also be used to find a longest common subsequence of two input sequences. The differences are that now we are looking for the maximum rather than the minimum, and that $D[i,j]$ depends only on $E[i,j]$. Indeed, $D[i,j] = E[i,j] + 1$ for pairs of symbols (i,j) that match, and $D[i,j] = E[i,j]$ otherwise. The cost function $w(x,y)$ is always zero (and therefore linear). Thus any bounds on the time for solving recurrence 4 will also apply to the longest common subsequence problem. As we have stated the solution, the time bound applies with M being the total number of matching positions between the two input strings.

Apostolico and Guerra [5] cleverly showed that the problem can be made even more sparse, by considering only *dominant matches*. They give an algorithm which runs in $O(n \log s + m \log n + d \log(nm/d))$, where d is the number of dominant matches. A different version of this algorithm can be also implemented in $O(n \log s + d \log\log n)$ time. We now outline how to achieve a better time bound, by modifying their algorithm to take advantage of our techniques.

The key observation is to replace the C-trees defined and used in [5] with Johnson's data structure [9]. Apostolico and Guerra showed that their algorithm performs at most $O(d)$ insertions, deletions and lookup operations on integers in $\{1, \ldots, n\}$. Furthermore, their algorithm can be implemented in such a way that for each step insertions, deletions and lookup operations are never intermixed on the same priority queue. Therefore we can apply lemma 1 and the same argument of theorem 2 to obtain an algorithm which runs in $O(d \log\log \min(d, nm/d))$.

As in the algorithm of Apostolico and Guerra, and other similar algorithms for this problem, our algorithm also includes a preprocessing phase; this takes time $O(n \log s)$, where s is the alphabet size (without loss of generality at most $m + 1$). We must also initialize $O(s)$ search structures, with total cardinality of at most n; using Johnson's data structure this can be accomplished in $O(s \log\log(n/s))$ time which is dominated by the $O(n \log s)$ term.

Therefore the total time is $O(n \log s + d \log\log \min(d, nm/d))$.

6. Conclusions

We have shown how to efficiently solve the Wilbur-Lipman sequence alignment problem, the minimal energy RNA secondary structure with single loops and the longest common subsequence problem. Our approach takes advantage of the fact that all the above problems can be solved by computing a dynamic programming recurrence on a sparse set of entries of the corresponding

dynamic programming matrix. We have also assumed that the weight functions involved are linear. In the companion paper [7] we will analyze the case where the weight functions are either convex or concave. Our algorithms have time bounds that vary almost linearly in the density of the problems. Even when the problems are dense, our algorithms are no worse than the best known algorithms; when the problems are sparse, our time bounds become much better than those of previous algorithms.

We remark that all our algorithms are independent of the particular heuristics used to make the input sparse. This is especially important for the Wilbur-Lipman sequence alignment algorithm, where such heuristics may vary depending on which application the algorithm is used for.

References

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

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

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

[5] A. Apostolico and C. Guerra, The Longest Common Subsequence Problem Revisited, Algorithmica 2, 1987, 315–336.

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

[7] David Eppstein, Zvi Galil, Raffaele Giancarlo and Giuseppe F. Italiano, Sparse Dynamic Programming II: Convex and Concave Cost Functions, manuscript.

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

[9] 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.

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

[17] E.M. McCreight, A Space Economical Suffix Tree Construction Algorithm, J. ACM 23, 1976, 262-272.

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

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

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

[27] 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.

[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] P. Wiener, Linear Pattern Matching Algorithms, 14th Symposium on Switching and Automata Theory, 1973, 1-11.

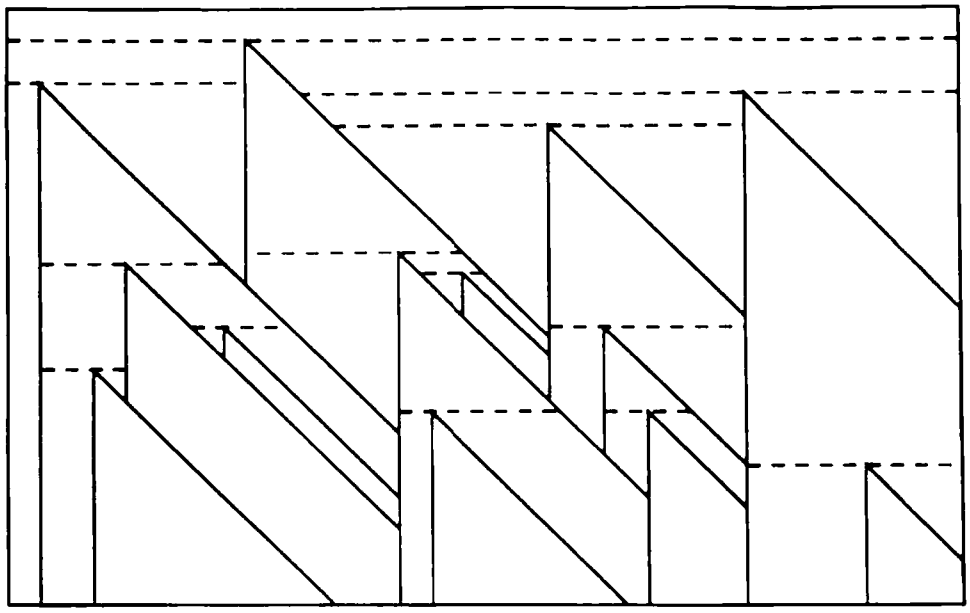
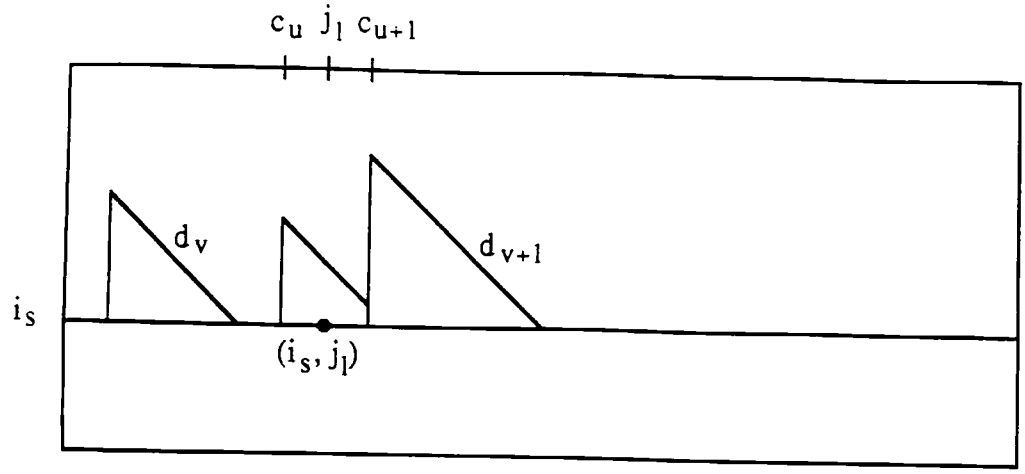
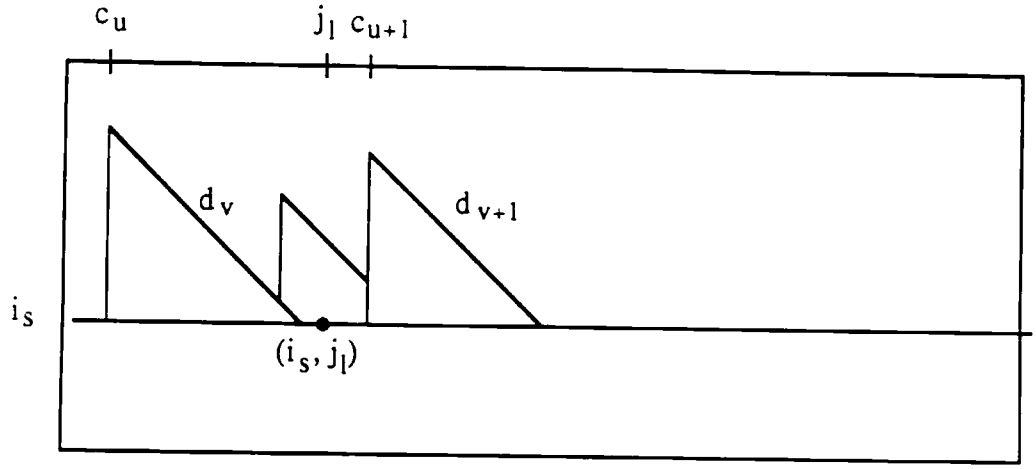


Figure 1

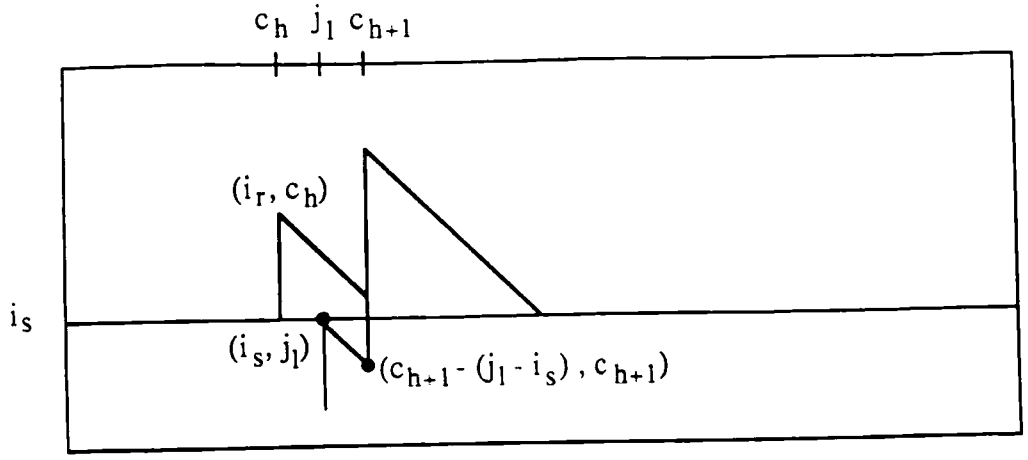


(a)

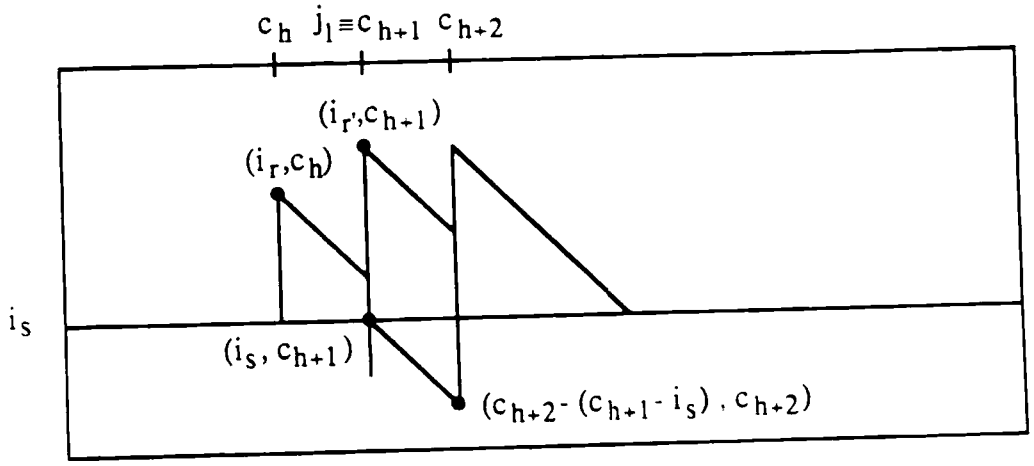


(b)

Figure 2

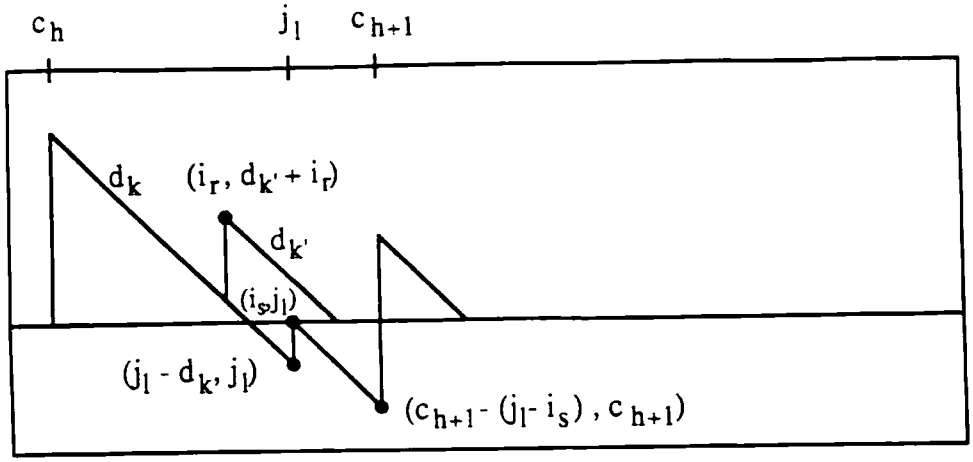

c_h j_1 c_{h+1}
(i_r, c_h)
i_s
(i_s, j_1)
$(c_{h+1} - (j_1 - i_s), c_{h+1})$

(a)

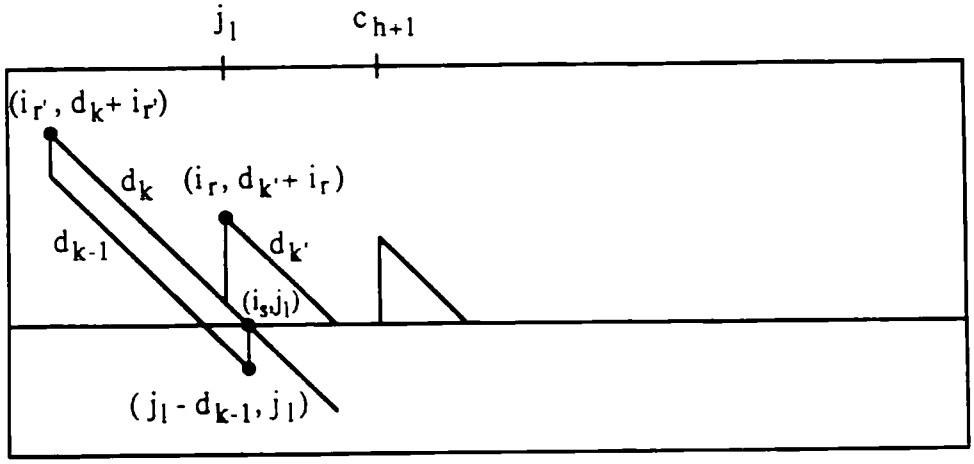

c_h $j_1 \equiv c_{h+1}$ c_{h+2}
$(i_{r'}, c_{h+1})$
(i_r, c_h)
i_s
(i_s, c_{h+1})
$(c_{h+2} - (c_{h+1} - i_s), c_{h+2})$

(b)

Figure 3


c_h
j_l
c_{h+1}
d_k
$(i_r, d_{k'} + i_r)$
$d_{k'}$
(i_s, j_l)
$(j_l - d_k, j_l)$
$(c_{h+1} - (j_l - i_s), c_{h+1})$

(a)


j_l
c_{h+1}
$(i_{r'}, d_k + i_{r'})$
d_k
$(i_r, d_{k'} + i_r)$
d_{k-1}
$d_{k'}$
(i_s, j_l)
$(j_l - d_{k-1}, j_l)$

(b)

Figure 4

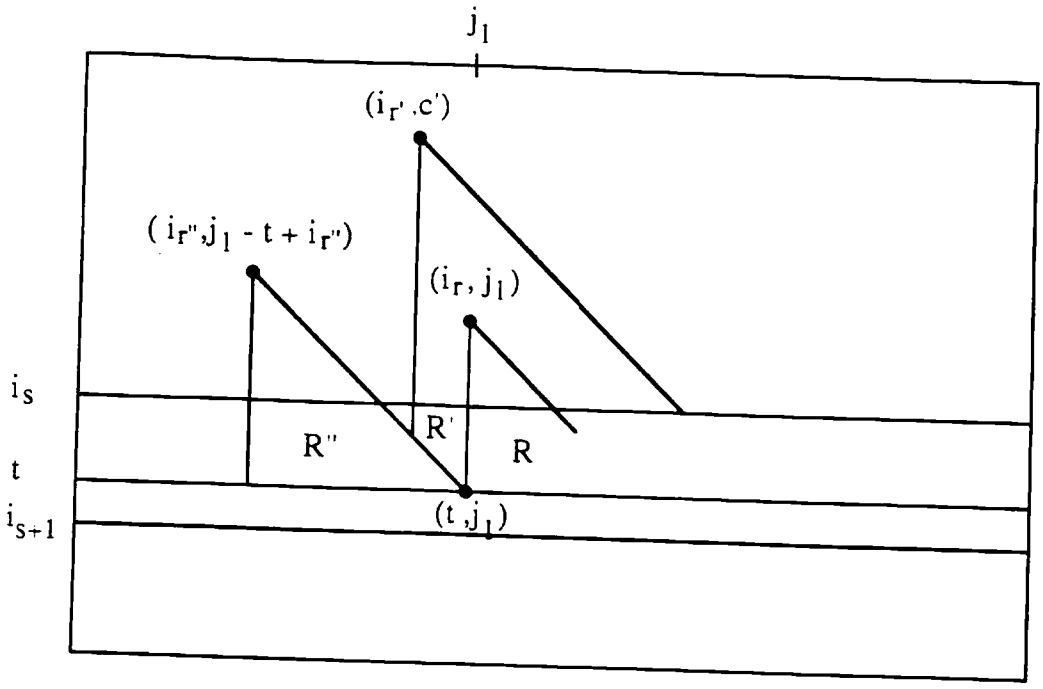


Figure 5