On-Line Dynamic Programming with Applications to the Prediction of RNA Secondary Structure

Lawrence L. Larmore*
Baruch Schieber#

Abstract

We define an *on-line* problem to be a problem where each input is available only after certain outputs have been calculated. The usual kind of problem, where all inputs are available at all times, is referred to as an *off-line* problem. We present an efficient algorithm for the on-line two dimensional dynamic programming problem that is used for the prediction of RNA secondary structure. Our algorithm uses as a module an algorithm for solving the on-line one dimensional dynamic programming problem. The time complexity of our algorithm is n times the complexity of the on-line one dimensional dynamic programming problem. For the concave case, we present a linear time algorithm for the on-line one dimensional problem. This yields an optimal $O(n^2)$ time algorithm for the on-line two dimensional concave problem. The constants in the time complexity of this algorithm are fairly small, which make it practical. For the convex case, we use an $O(n\alpha(n))$ time algorithm for the on-line one dimensional problem, where $\alpha(.)$ is the functional inverse of Ackermann's function. This yields an $O(n^2\alpha(n))$ time algorithm for the on-line two dimensional convex problem. Both algorithms improve on previously known algorithms.

* Dept. of Mathematics and Computer Science, University of California at Riverside, Riverside, CA 92521.

IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

1. Introduction

Dynamic programming is a widely used problem-solving technique. It has many applications in various fields, among them: Computer Science, Operation Research and Molecular Biology. (See, e.g., [3, 11].) In this paper we consider a variant of dynamic programming that is used to predict RNA secondary structure from the primary (linear) RNA sequence ([12, 13]). An efficient method for deducing the secondary structure directly from the primary structure is very useful, since empirical results are costly to obtain and can often be interpreted in several ways.

Following the formulation given in [5], we consider the problem of computing the two dimensional recurrence

$$E[i,j] = \min\{F[i',j'] + W(i'+j', i+j) \mid 0 \le i' < i, 0 \le j' < j\}, \text{ for } 1 \le i,j \le n, \quad (1)$$

under the following two assumptions:

1. The entries of the matrix F are easily computed from the corresponding entries of the the matrix E. That is, each input $F[i,j]$ is available only after the output $E[i,j]$ has been calculated. We define such a problem where each input is available only after certain outputs have been calculated to be an *on-line* problem. A problem where all inputs

are available at all times is defined to be an *off-line* problem.

2. The *weight function* $W(.,.)$ is either *concave* or *convex*. We define a bivariate function $W(.,.)$ to be *concave* if the quadrangle inequality: $W(i,j) + W(i',j') \leq W(i,j') + W(i',j)$ holds for all $i < i' < j < j'$. Similarly, a bivariate function $W(.,.)$ is defined to be *convex* if the inverse quadrangle inequality holds for all $i < i' < j < j'$.

We present an efficient algorithm for the on-line two dimensional dynamic programming problem defined above. Our algorithm uses as a module an algorithm for solving the following on-line one dimensional dynamic programming recurrence:

$$E[i] = \min\{F[i'] + W(i',i) \mid 0 \leq i' \leq C_i\}, \qquad (2)$$

for $1 \leq i \leq n$, where $0 \leq C_1 \leq \ldots \leq C_n < n$.

The assumptions are similar to the ones given for the two dimensional case, namely:

1. The values of $F[j]$, for $j = C_{i-1} + 1, \ldots, C_i$ are easily computed from the value of $E[i-1]$. (For convenience, we define $C_0 = 0$.)

2. The *weight function* $W(.,.)$ is either *concave* or *convex*.

The time complexity of our algorithm is n times the complexity of the on-line one dimensional dynamic programming problem.

We give an optimal linear time algorithm for the on-line one dimensional concave problem. This yields an $O(n^2)$ time algorithm for the on-line two dimensional concave problem. Notice that the size of the input in this problem is $O(n^2)$, and thus, our algorithm is optimal. The constants in the time complexity of the algorithms for both the one and the two dimensional cases are fairly small, which make them practical.

For the convex case, we use an $O(n\alpha(n))$ time algorithm for the on-line one dimensional problem given in [9], where $\alpha(.)$ is the functional inverse of Ackermann's function. This yields an $O(n^2\alpha(n))$ time algorithm for the on-line two dimensional convex problem. However, any improvement to the on-line one dimensional algorithm will result in a corresponding improvement to our algorithm.

The on-line two dimensional dynamic programming problem that is considered here was first considered by Waterman and Smith [13]. They show its relevance to predicting the RNA secondary structure and give an $O(n^3)$ time algorithm for the problem. Eppstein, Galil, and Giancarlo [5] improve it to $O(n^2 \log^2 n)$ time. They also give the formulation used in this paper (which is implicit in [13]). The previously best known algorithm for the problem is due to Aggarwal and Park [2]. Their algorithm runs in $O(n^2 \log n)$ time. Notice that our algorithm is superior to the previous algorithms in both the concave and the convex cases. We remark that Yao [15] uses the quadrangle inequality to accelerate the computation of another variant of two dimensional dynamic programming.

An $n \times m$ triangular matrix M is defined to be *concave totally monotone* if for all $1 \leq i < i' \leq n$ and $0 \leq j < j' < m$, the inequality $M[i,j] > M[i,j']$ implies that $M[i',j] > M[i',j']$. Similarly, an $n \times m$ triangular matrix M is defined to be *convex totally monotone* if for all $1 \leq i < i' \leq n$ and $0 \leq j < j' < m$, the inequality $M[i,j] < M[i,j']$ implies that $M[i',j] < M[i',j']$. (To make the presentation clearer we assume that all the finite elements in M are distinct.)

An $n \times m$ matrix M is defined to be *generalized upper triangular* if there are $0 \leq C_1 \leq C_2 \leq \ldots \leq C_n = m-1$, such that $M[i,j] = \infty$ for all $C_i < j < m$. A generalized upper triangular matrix is concave (resp. convex) totally monotone if the above concavity (resp. convexity) conditions hold for any four non-infinity entries of M which form a rectangular submatrix.

The on-line one dimensional dynamic programming problem considered here can be viewed as an on-line searching in a generalized upper tri-

angular totally monotone matrix. Define the $n \times m$ upper triangular matrix M by $M[i,i'] = F[i'] + W(i',i)$, for $1 \leq i \leq n$, $0 \leq i' \leq C_i$. (The rest of the elements of M are defined to be ∞.) Then, solving the recurrence (2) is equivalent to finding the minimum element in each row of the matrix M. It is easy to see that the concavity assumption on the weight function translates to the condition that M is concave totally monotone. Similarly, the convexity assumption translates to the condition that M is convex totally monotone. The on-line assumption translates to the constraint that the elements of columns $C_{i-1}+1,\ldots,C_i$ of M (that are not defined to be ∞) are available only after the minimum element in row $i-1$ has been found.

Eppstein, Galil, and Giancarlo [5] give an $O(n \log n)$ time algorithm for the on-line one dimensional dynamic programming. Their algorithm is a generalization of the algorithm used by Hirschberg and Larmore for the concave least weight subsequence problem [7]. Wilber [14] solves the concave least weight subsequence problem in $O(n)$ time. However, his algorithm cannot be used for the on-line one dimensional concave dynamic programming since it does not satisfy the on-line constraint. That is, in his algorithm some entries of F are considered before the corresponding entry of E is computed. Wilber's algorithm uses totally monotone matrix searching techniques introduced in [1]. Eppstein [4] extended Wilber's algorithm for the on-line case. Our algorithm is more general than Eppstein's algorithm; it works for a general on-line searching in a totally monotone triangular matrix, while Eppstein's algorithm works only for the on-line one dimensional dynamic programming problem. Recently, independent to our work, Galil and Park [6] give a linear time algorithm for the on-line one dimensional concave problem. Also, Klawe [8] gives a linear time algorithm for the same problem.

Searching in totally monotone matrices was first considered by Aggarwal et al. in [1]. They consider the off-line problem for full (i.e., non triangular) matrices. Aggarwal et al. give an $O(n)$ time algorithm for that problem, now nicknamed the SMAWK algorithm, and also show various applications of this algorithm to solving problems in computational geometry and VLSI. It is not difficult to see that the same algorithm applies also for off-line searching in an upper triangular concave totally monotone matrix. Klawe and Kleitman [9] give an $O(n\alpha(n))$ time algorithm for off-line searching in an upper triangular convex totally monotone matrix. Their algorithm can be used to solve the respective on-line problem. We use this algorithm to achieve our algorithm for the two dimensional convex case.

The off-line algorithms for both the one dimensional and two dimensional dynamic programming problems are recursive. These algorithms are not suitable for the on-line problems since the recursive processes access inputs before they become available. These inputs become available only after the recursive processes are terminated. To overcome this problem we implement the recursion in a different way. When one process invokes another recursive process it does not wait until the recursive process terminates. Instead, both processes are active and intermediate values are communicated between the two. We believe that this technique is general, and can be used in other algorithms for on-line problems.

The rest of the extended summary is organized as follows. In the next section we describe the algorithm for the on-line two dimensional dynamic programming. In Section 3 we overview the linear time algorithm for the on-line one dimensional concave dynamic programming. The full description of this algorithm can be found in the full paper [10].

2. The two dimensional dynamic programming algorithm

In this section we describe an algorithm for solving the recurrence (1). The algorithm works for both the concave and the convex cases, the only difference is in the algorithm for the one dimensional problem that is used as a module. To make

the presentation clearer we assume in this section that $n+1$ is a power of two. Our algorithm can be easily modified for the case where $n+1$ is not a power of two.

We start with some definitions.

Consider the $(n+1) \times (n+1)$ grid with set of points $\{p = (i,j) \mid 0 \leq i,j \leq n\}$. Define the natural partial order on the points of the grid. That is, the point $p' = (i',j') \prec p = (i,j)$ if $i' < i$ and $j' < j$. The point p' is called a *predecessor* of p.

Let $p = (i,j)$ be a point. Define i to be the *row index* of p, j to be the *column index* of p, and $i+j$ to be the *diagonal index* of p, denoted $d(p)$.

For $0 \leq l \leq \log_2(n+1)$, define a *square of level* l to be any $2^l \times 2^l$ square of points whose upper left corner has both row and column indices that are multiples of 2^l. Let $S^l_{i,j}$ be the square of level l whose upper left corner is at the point $(i2^l, j2^l)$. Let $S^l_{i,*}$ be the set of squares of level l whose upper left corner is in row $i2^l$. Similarly, let $S^l_{*,j}$ be the set of squares of level l whose upper left corner is in column $j2^l$. Notice that each square $S^l_{i,j}$ consists of four squares of level $l-1$. These are: the upper left sub square $S^{l-1}_{2i,2j}$, the upper right sub square $S^{l-1}_{2i,2j+1}$, the lower left sub square $S^{l-1}_{2i+1,2j}$, and the lower right sub square $S^{l-1}_{2i+1,2j+1}$. Every point p lies within exactly one square of level l. Let $S^l(p)$ be the square of level l that contains the point p.

We extend the relation "$\prec$" to be defined over the squares. For two squares S and S' (of any levels), $S' \prec S$ if every point in S' precedes every point in S. That is, if the lower right corner of S' is a predecessor of the upper left corner of S.

For $0 \leq l \leq \log_2(n+1)$, define

$$E_l[p] = \min\{F[p'] + W(d(p'), d(p)) \mid S^l(p') \prec S^l(p)\}.$$

Notice that $E_0[p] = E[p]$ and that $E_{\log_2(n+1)}[p] = \infty$, for all points p. Also, for any l and for all points p in squares that are in either the set $S^l_{0,*}$ or $S^l_{*,0}$, $E_l[p] = \infty$.

We now turn to the description of our algorithm. First, we describe the algorithm for the off-line problem. Then, we show how to modify the algorithm for the on-line problem.

2.1. The off-line algorithm

Suppose that all the entries of the matrix F are available at all times. We describe an algorithm for computing the recurrence (1). The algorithm consists of $\log_2(n+1)$ phases. Starting from $l = \log_2(n+1)-1$ down to $l = 0$, in phase l we compute the matrix E_l given the matrix E_{l+1}. Below, we describe phase l of the algorithm.

Consider a point p, and let $S^{l+1}_{i,j} = S^{l+1}(p)$. We distinguish between four cases depending on the position of the square $S^l(p)$ in the square $S^{l+1}(p)$.

Case 1. The square $S^l(p)$ is the upper left sub square of $S^{l+1}(p)$; that is, $S^l(p) = S^l_{2i,2j}$. In this case it is easy to see that $E_l(p) = E_{l+1}(p)$.

Case 2. The square $S^l(p)$ is the upper right sub square of $S^{l+1}(p)$; that is, $S^l(p) = S^l_{2i,2j+1}$. In this case there are entries of F that have to be considered when computing $E_l(p)$ and were not considered in computing $E_{l+1}(p)$. These entries correspond to the points in the squares in $S^l_{*,2j}$ that precede the square $S^l_{2i,2j+1}$.

For the points in the squares in $S^l_{*,2j+1}$, define the following recurrence, called the *column recurrence*

$$COL^l_j[p] = \min\{F[p'] + W(d(p'), d(p)) \mid p' \in S^l_{*,2j} \text{ and } S^l(p') \prec S^l(p)\}. \quad (3)$$

Then, $E_l[p] = \min\{E_{l+1}[p], COL^l_j[p]\}$.

Case 3. The square $S^l(p)$ is the lower left sub square of $S^{l+1}(p)$; that is, $S^l(p) = S^l_{2i+1,2j}$. In this case there are entries of F that have to be considered when computing $E_l(p)$ and were not considered in computing $E_{l+1}(p)$. These entries correspond to the points in

the squares in $S^l_{2i,*}$ that precede the square $S^l_{2i+1,2j}$.

For the points in the squares in $S^l_{2i+1,*}$, define the following recurrence, called the *row recurrence*

$$ROW^l_i[p] = \min\{F[p'] + W(d(p'), d(p)) \mid p' \in S^l_{2i,*} \text{ and } S^l(p') \prec S^l(p)\}. \quad (4)$$

Then, $E_l[p] = \min\{E_{l+1}[p], ROW^l_i[p]\}$.

Case 4. The square $S^l(p)$ is the lower right sub square of $S^{l+1}(p)$; that is, $S^l(p) = S^l_{2i+1,2j+1}$. Similar to the previous cases we get that in this case $E_l[p] = \min\{E_{l+1}[p], ROW^l_i[p], COL^l_j[p]\}$.

We conclude that to compute E_l given E_{l+1} we have to solve the $(n+1)/2^{l+1}$ row recurrences and $(n+1)/2^{l+1}$ column recurrences. It follows from Observation 1 below that the entries of E_l have only $O((n+1)^2/2^l)$ distinct values. Thus, overall we need to compute only $O(n^2)$ values. Below, we show how to solve each row and column recurrence by executing four instances of the one dimensional algorithm.

The row and column recurrences. We show how to compute COL^l_j. The algorithm for solving the row recurrences is analogous.

We make the following two observations that are implied by the fact that the weight function $W(.,.)$ depends only on the diagonal index of its two argument points.

OBSERVATION 1: The value of $COL^l_j[p]$ (and of $E_l[p]$) is the same for all points p with the same diagonal index that are in the same square of level l.

OBSERVATION 2: To compute $COL^l_j[p]$ (and $E_l[p]$) it is sufficient to consider only a single value for all points p' with the same diagonal index that are in the same square of level l that precedes $S^l(p)$. This value is the minimum $F[p']$ among all these points p'.

Consider the diagonals that intersect the set of squares in $S^l_{*,2j}$. Note that the indices of these diagonals are in the range: $j2^{l+1}, \ldots, j2^{l+1} + 2^l + n - 1$. Each such diagonal intersects at most two squares in the set, one square with an even row index, i.e., a square $S^l_{2i,2j}$, and one with an odd row index. For $j2^{l+1} \le k' \le j2^{l+1} + n - 1$, let $F_{even}[k']$ be the minimal value of $F[p]$ among all points p on the diagonal k' that intersect the (unique) square with an even row index from the set $S^l_{*,2j}$. Similarly, for $j2^{l+1} + 2^l \le k' \le j2^{l+1} + 2^l + n - 1$, let $F_{odd}[k']$ be the minimal value of $F[p]$ among all points p on the diagonal k' that intersect the (unique) square with an odd row index from the set $S^l_{*,2j}$.

Let k be the index of a diagonal that intersects squares in $S^l_{*,2j+1}$. Let S be the (unique) square in this set with an even row index that intersects the diagonal k. Consider all the points p in this intersection. By Observation 1 the value of $COL^l_j[p]$ is the same for all these points. We compute $COL^l_j[p]$ for all these points in two stages. First, we compute the influence on $COL^l_j[p]$ of points from squares in $S^l_{*,2j}$ with even row indices, then, we compute the influence of points from squares in $S^l_{*,2j}$ with odd row indices.

Let $D_{even}[k]$ denote the influence of points from squares in $S^l_{*,2j}$ with even row indices. To compute $D_{even}[k]$ it is sufficient to consider only $F_{even}[k']$, for all diagonals k' that intersect squares with even row indices in $S^l_{*,2j}$ that precede the square S. A simple calculation shows that these diagonals are in the range $j2^{l+1}, \ldots, \left\lfloor \frac{k-2^l}{2^{l+1}} \right\rfloor 2^{l+1} - 2$. Thus, we get

$$D_{even}[k] = \min\{F_{even}[k'] + W(k', k) \mid j2^{l+1} \le k' < \left\lfloor \tfrac{k-2^l}{2^{l+1}} \right\rfloor 2^{l+1} - 2\},$$

Similarly, let $D_{odd}[k]$ denote the influence of points from squares in $S^l_{*,2j}$ with odd row indices. Again, it is sufficient to consider only $F_{odd}[k']$, for all diagonals k' that intersect squares with odd row indices in $S^l_{*,2j}$ that precede the square S. We get

$$D_{odd}[k] = \min\{F_{odd}[k'] + W(k', k) \mid$$

$j2^{l+1} + 2^l \leq k' < 2^l + \left\lfloor \frac{k-2^l}{2^{l+1}} \right\rfloor 2^{l+1} - 2\}$.

Clearly, $COL^l_j[p] = \min\{D_{even}[k], D_{odd}[k]\}$, for all points p on the diagonal k that intersect a square with an even row index from the set $S^l_{*,2j}$.

Both recursions can be translated into one dimensional matrix searching problems as shown in the full paper.

In a similar way we can define two one dimensional problems to compute $COL^l_j[p]$ for all points p on the diagonal k that intersect the (unique) square with an odd row index from the set $S^l_{*,2j}$.

We conclude

Theorem 2.1: *An instance of the off-line two dimensional dynamic programming of size n can be solved by solving at most $8n$ instances of the off-line one dimensional dynamic programming of size n.*

Corollary 2.2: *The off-line two dimensional concave dynamic programming can be solved in $O(n^2)$ time. The off-line two dimensional convex dynamic programming can be solved in $O(n^2\alpha(n))$ time.*

2.2. The on-line algorithm

In the on-line algorithm we compute the same matrices E_l; however, we do not do it serially. Instead, we advance along the rows and at each time we compute the top part of all the matrices E_l. In this process we use the off-line one dimensional algorithm for computing the row recurrences and the on-line one dimensional algorithm for computing the column recurrences.

The algorithm computes the entries of $E = E_0$ row by row. Suppose that the r rows indexed $0, \ldots, r-1$ of E have been computed already. This implies that rows $0, \ldots, r-1$ of F are available. The invariant property maintained by the algorithm at this point is as follows.

INVARIANT PROPERTY FOR r.

1. *For each $0 \leq l \leq \log_2(n+1)$, the $\left\lceil r/2^l \right\rceil 2^l$ first rows of E_l have been computed.*
2. *For each $0 \leq l \leq \log_2(n+1)$ and for each square $S \in S^l_{i,*}$, $0 \leq i < \left\lfloor r/2^l \right\rfloor$, the minimum value of $F[p]$ among the points p on diagonal k in S has been computed, for each diagonal k in S.*

Notice that the Invariant Property for $r = n+1$ implies that the whole matrix $E = E_0$ has been computed.

We prove the Invariant Property by induction. The Invariant Property is clearly satisfied for $r = 1$. Suppose that the Invariant Property holds for r. We describe the computation required in order to advance to $r+1$.

MAINTAINING PART (1) OF THE INVARIANT PROPERTY. Let k be the maximum integer such that 2^k divides r. Note that for $l \leq k$, $\left\lceil r/2^l \right\rceil = \left\lceil (r+1)/2^l \right\rceil - 1$, and for $l > k$, $\left\lceil r/2^l \right\rceil = \left\lceil (r+1)/2^l \right\rceil$. Thus, to maintain part (1) of the Invariant Property for $r+1$ we have to compute $E_l[p = (i,j)]$, for $r \leq i \leq r + 2^l - 1$ and $0 \leq j \leq n$, for all $0 \leq l \leq k$. Since 2^l divides r, these points are exactly the points in the squares $S^l_{r/2^l,*}$, and hence, we have to compute $E_l[p]$, for all p in $S^l_{r/2^l,*}$.

The computation consists of two steps.

Step 1. Compute $E_k[p]$, for all p in $S^k_{r/2^k,*}$.

For $0 \leq j \leq (n+1)/2^{k+1}$, consider the squares $S^k_{r/2^k,2j}$ and $S^k_{r/2^k,2j+1}$. These squares are the lower left sub square and the lower right sub square of $S^{k+1}_{(r-2^k)/2^{k+1},j}$. (Notice that 2^{k+1} divides $r - 2^k$.) By Case 3 above, $E_k[p]$, for all p in $S^k_{r/2^k,2j}$, $0 \leq j \leq (n+1)/2^{k+1}$, is given by computing the row recurrence $ROW^k_{(r-2^k)/2^{k+1}}[p]$. The Invariant Property for r implies that this recurrence can be computed off-line. By Case 4 above, $E_k[p]$, for all p in $S^k_{r/2^k,2j+1}$, $0 \leq j \leq (n+1)/2^{k+1}$, is given by computing the same row recurrence and the en-

tries $COL^k_j[p]$. These entries can be computed by activating the on-line algorithms for computing each of these column recurrences. The Invariant Property for r implies that the entries $F[p']$, for $p' \in S^k_{i',2j}$, $0 \leq i' < r/2^k$ that are needed for the computation are available.

Step 2. For $l = k-1$ downto 0, compute $E_l[p]$, for all p in $S^l_{r/2^l,*}$.
For $0 \leq j \leq (n+1)/2^{l+1}$, consider the squares $S^k_{r/2^l,2j}$ and $S^k_{r/2^l,2j+1}$. These squares are the upper left sub square and the upper right sub square of $S^{l+1}_{r/2^{l+1},j}$. (Notice that 2^{l+1} divides r.) By Case 1 above, $E_l[p] = E_{l+1}[p]$, for all p in $S^l_{r/2^l,2j}$, $0 \leq j \leq (n+1)/2^{l+1}$. By Case 2 above, $E_l[p]$, for all p in $S^l_{r/2^l,2j+1}$, $0 \leq j \leq (n+1)/2^{l+1}$, is given by computing the entries $COL^l_j[p]$. These entries can be computed by activating the on-line algorithms for computing each of these column recurrences. Again, the Invariant Property for r implies that the entries $F[p']$, for $p' \in S^l_{i',2j}$, $0 \leq i' < r/2^l$ that are needed for the on-line computation are available.

All the computations can be done by the one dimensional dynamic programming algorithm using the minimum elements in each diagonal computed to maintain part (2) of the Invariant Property for r.

MAINTAINING PART (2) OF THE INVARIANT PROPERTY. Let k be the maximum integer such that 2^k divides $r+1$. Note that for $l \leq k$, $\lfloor r/2^l \rfloor = \lfloor (r+1)/2^l \rfloor - 1$, and for $l > k$, $\lfloor r/2^l \rfloor = \lfloor (r+1)/2^l \rfloor$. Thus, to maintain part (2) of the Invariant Property for $r+1$ we have to compute the minimal value of $C[p]$ in each diagonal of each of the squares in $S^l_{i,*}$, for $0 \leq l \leq k$ and $i = (r+1)/2^l - 1$. The computation can be done in constant time per diagonal, starting from $l = 0$, and using the precomputed minimum elements.

3. The one dimensional dynamic programming algorithm

In this section we describe a linear time algorithm for solving the on-line recurrence (2) under the assumption that $W(.,.)$ is concave. Instead of solving the recurrence we solve the following equivalent on-line matrix searching problem.

THE MATRIX SEARCHING PROBLEM. Let M be an $n \times m$ generalized upper triangular concave totally monotone matrix. The rows of M are indexed in the range $1, \ldots, n$, and the columns are indexed in the range $0, \ldots, m-1$. For each $1 \leq i \leq n$ there is a column index $C_{i-1} \leq C_i < m$ such that $M[i,j] = \infty$ for all $j > C_i$. Find the minimum element in each row of the matrix M, under the constraint that, for $i > 1$, the value of C_i and the elements of columns $C_{i-1}+1, \ldots, C_i$ of M (that are not defined to be ∞) are available only after the minimum element in row $i-1$ has been found.

We show that the Matrix Searching Problem can be solved in linear time. We give two different reductions of an instance of the Matrix Searching Problem to a smaller instance of the same problem. Using just one of these reductions does not give a linear bound for the problem, but if they are interleaved, linearity is achieved. The technique is similar to that used for the off-line SMAWK algorithm [1]. The two reductions are as follows:

1. An instance of size $n \times m$ of the Matrix Searching Problem can be reduced to an instance of size $\lfloor n/2 \rfloor \times m$ of the same problem in $O(n+m)$ time.

2. An instance of size $n \times m$ of the Matrix Searching Problem can be reduced to an instance of size $n \times n$ of the same problem in $O(n+m)$ time.

Interleaving the reductions and solving the resulting recurrence, we get that the Matrix Searching Problem can be solved in $O(n+m)$ time.

Reduction 1

Let M be an $n \times m$ input matrix for the Matrix Searching Problem. Define M' to be the sub-matrix consisting of all entries $M[2i, j]$ for which $0 \leq j \leq C_{2i-1}$; that is, M' consists of all entries in each even row of M that lie under a non-infinity element in the previous odd row. It is easy to see that M' is an input matrix for an instance of the Matrix Searching Problem of size $\lfloor n/2 \rfloor \times m'$, for some $m' \leq m$. The sequence $C'_1, \ldots, C'_{\lfloor n/2 \rfloor}$, that corresponds to M', is given by $C'_i = C_{2i-1}$, for $i = 1, \ldots, \lfloor n/2 \rfloor$. (See Figure 1.)

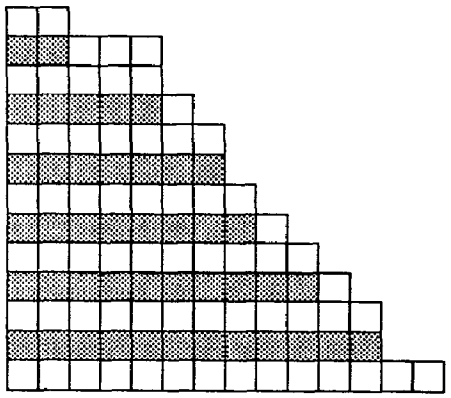

Figure 1: The result from Reduction 1.

Suppose that we are given an algorithm $\mathcal{A}'$ that computes on-line the row minima of M'. We describe an algorithm that computes on-line the row minima of M by interleaving its computations with the computations of the algorithm $\mathcal{A}'$.

Let $j_M(i)$ be the column index of the minimum element in row i of M. Similarly, Let $j_{M'}(i)$ be the column index of the minimum element in row i of M'.

When the algorithm starts columns $0, \ldots C_1$ are available, and hence row 1 of M and row 1 of M' are available. Algorithm $\mathcal{A}'$ is activated to compute $j_{M'}(1)$. Recall that the first row of M' consists of all entries in the second row of M that lie under a non-infinity element in the first row. From this and from the total monotonicity of M it follows that $0 \leq j_M(1) \leq j_{M'}(1)$. Thus, $j_M(1)$ is that $j \in \{0, \ldots j_{M'}(1)\}$ for which $M[1, j]$ is minimum. After computing $j_M(1)$ columns $C_1 + 1, \ldots, C_2$, and hence row 2 of M become available. We set $j_M(2)$ to be that $j \in \{j_{M'}(1), C_1 + 1, \ldots C_2\}$ for which $M[2, j]$ is minimum. After computing $j_M(2)$ columns $C_2 + 1, \ldots, C_3$, and hence row 3 of M and row 2 of M' become available. We continue in a similar manner; that is, for $i = 2, \ldots, \lfloor n/2 \rfloor$, algorithm $\mathcal{A}'$ is activated to compute $j_{M'}(i)$. Then, $j_M(2i-1)$ is set to be that $j \in \{j_M(2i-2), \ldots j_{M'}(i)\}$ for which $M[2i-1, j]$ is minimum, and $j_M(2i)$ is set to be that $j \in \{j_{M'}(i), C_{2i-1}+1, \ldots C_{2i}\}$ for which $M[2i, j]$ is minimum.

In the full paper, we give a formal description of the algorithm.

Reduction 2

Let M be an $n \times m$ input matrix for the Matrix Searching Problem. We show how to reduce the Matrix Searching Problem for M to the Matrix Searching Problem for an $n \times m'$ sub-matrix M' of M, for some $m' \leq n$. Notice that this reduction is meaningful only if $m > n$, since otherwise no advancement is made.

The sub-matrix M' is defined by *headers*. Each header is an entry of M. Each column of M' consists of one header together with all entries below it in the same column; that is, it contains the header together with all entries in the same column with higher row index. Headers are staggered down and to the right, i.e., no two headers are in the same row, and a header with a higher row index has also a higher column index. Figure 2 shows an example of a sub-matrix M' within a matrix M.

The sub-matrix M' is constructed in such a way that the minimum element in each row of M lies in the same row in M'. Thus, the row minima of M can be computed from the row minima of M', simply by translating the column index of these minima in M' to the corresponding column index in M. The hard part of the reduction is the construction of the sub-matrix M'; that is, the computation of the headers. This construction is interleaved with the execution of an algorithm $\mathcal{A}'$

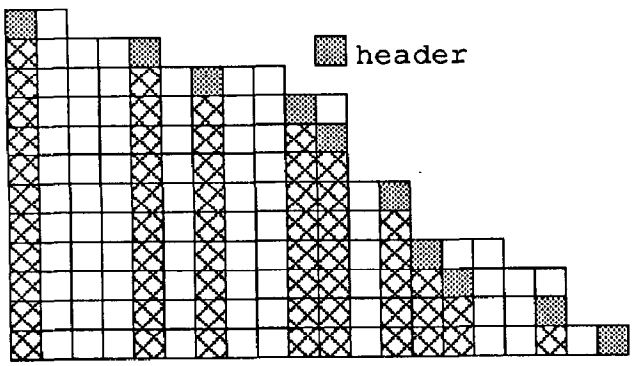

Figure 2: The result from Reduction 2.

that computes on-line the row minima of M'.

The computation of the headers is done by processing the columns of M one by one. In the computation we use an array S to store the tentative headers. The array S can be viewed as a stack, similar to the algorithms [5, 7]. The entry $S(i)$ is either undefined or consists of the row and column index of the i-th tentative header. We refer to these indices as $S_{row}(i)$ and $S_{col}(i)$. As the computation proceeds a header may either be popped out from the stack S, or become a permanent header. In case it becomes a permanent header it defines a column in M'. This column can be passed to algorithm $\mathcal{A}'$.

Informal description of the algorithm: The stack is initially empty. Columns are processed one at a time, from 0 to $m-1$. When column j is processed, this column "challenges" tentative headers on the top of the stack S. Column j "wins" the challenge if its entry in the same row as the top tentative header is smaller than the header. In other words, let t denote the index of the top of the stack S. Column j "wins" the challenge if $M[S_{row}(t), j] < M[S_{row}(t), S_{col}(t)]$. In this case the stack is popped (and t is decremented by one). We show in the full paper that when the stack is popped none of the elements in column $S_{col}(t)$ are candidates for the minimum elements in their rows.

Challenges continue until the stack is empty or until column j "loses" a challenge. If the stack is empty then the header $(1, j)$ is pushed into the stack. Otherwise, i.e., in case $M[S_{row}(t), j] > M[S_{row}(t), S_{col}(t)]$, implying that column j "lost" the challenge with the header on the top of the stack, the entry (i', j) is pushed onto the stack, where i' is the row index of the highest entry just below $(S_{row}(t), j)$ that consists of a non-infinity element. (Notice that in case $M[S_{row}(t), j] = \infty$, i' is not necessarily $S_{row}(t)+1$.) If $S_{row}(t) = n$, i.e., $S_{row}(t)$ is the last row in M then i' is undefined (since there are no rows below row n), in this case no entry is pushed onto the stack.

From the above description it can be seen that a tentative header (i, j) may be replaced only by tentative headers whose column index is less or equal to C_i. This is, since for $k > C_i$, $M[i, k] = \infty$, and thus the tentative header will always "win" the challenge with entries from these columns. This implies that after C_i columns of M have been processed the tentative header in row i, if such exists, becomes *permanent.* At this time the headers that define all the entries in row i of M' are permanent. Thus, we can set C'_i to be the number of these permanent headers, and activate algorithm $\mathcal{A}'$ to report the minimum of row i of M'. It is important to note that tentative headers become permanent headers at the last possible moment before they are needed by algorithm $\mathcal{A}'$. Thus, the columns of M' become visible to algorithm $\mathcal{A}'$ just when it needs them.

A formal description of the algorithm and its correctness proof are given in the full paper.

The complexity of the whole algorithm

We solve the one-dimensional on-line dynamic programming problem by alternation of the two reductions, just as in the SMAWK algorithm [1].

Let $C(n, m)$ denote the number of comparisons required for a problem of size $n \times m$. Similarly, let $F(n, m)$ denote the number of fetches required for a problem of size $n \times m$.

Reduction 1 gives the recurrences:

$$C(n, m) \leq 2m + C(\lfloor n/2 \rfloor, m)$$

$$F(n,m) \leq 2m + n/2 + F(\lfloor n/2 \rfloor, m).$$

While Reduction 2 gives the recurrences:

$$C(n,m) \leq 2m + C(n,n)$$

$$F(n,m) \leq 3m + F(n,n).$$

Using these reductions we get:
(1) For $n < m$, $C(n,m) \leq 2m+8n$, and $F(n,m) \leq 3m + 11n$.
(2) For $m \leq n < 2m$, $C(n,m) \leq 4m + 4n$ and $F(n,m) \leq 5m + 6n$.
(3) For $2m \leq n$, $C(n,m) \leq 12m + 2n$ and $F(n,m) \leq 17m + 3n$.

The Regular Problem

In order to more fairly compare our algorithm with the (off-line) SMAWK algorithm and with other existing on-line algorithms by Galil and Park [6], and Klawe [8], we give in the full paper an accurate count of the comparisons and fetches in what we call the *Regular Problem*. The Regular Problem is the special case of the Matrix Searching Problem where $n = m$ and $C_i = i - 1$ for all i. We show that the number of comparisons needed by our algorithm in that case is only $6n$, and the number of fetches is $8.5n$. These constants are superior to the constants achieved by other linear time on-line algorithms for the problem. For comparison, the (off-line) SMAWK algorithm, when applied to an off-line Regular Problem, performs $5n$ comparisons and $8n$ fetches.

References

[1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:209–233, 1987.

[2] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 497–512, October 1988.

[3] K.Q. Brown. Dynamic programming in computer science. Technical Report CMU-CS-79-106, Dept. of Computer Science, Carnegie-Mellon University, February 1979.

[4] D. Eppstein. Sequence comparison with mixed convex and concave costs, October 1988. To appear in *Journal of Algorithms*.

[5] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 488–496, October 1988.

[6] Z. Galil and K. Park. A linear time algorithm for concave one dimensional dynamic programming. Manuscript, Dept. of Computer Science, Columbia University, NY, 1989.

[7] D.S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16:628–638, 1987.

[8] M.M. Klawe. A simple linear time algorithm for concave one-dimensional dynamic programming. Technical Report 89-16, The University of British Columbia, Vancouver, BC, 1989.

[9] M.M. Klawe and D.J. Kleitman. An almost linear time algorithm for generalized matrix searching. Technical Report RJ 6275, IBM - Research Division, Almaden Research Center, 1988.

[10] L.L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure, 1989. Submitted for journal publication.

[11] D. Sankoff and J.B. Kruskal, editors. *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Ma, 1983.

[12] D. Sankoff, J.B. Kruskal, S. Mainville, and R.J. Cedergren. Fast algorithms to determine RNA secondary structures containing multiple loops. In D. Sankoff and J.B. Kruskal, editors, *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 95–120. Addison-Wesley, Reading, Ma, 1983.

[13] M.S. Waterman and T.F. Smith. Rapid dynamic programming algorithms for RNA secondary structure. *Advances in Applied Math.*, 7:455–464, 1986.

[14] R. Wilber. The concave least weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, September 1988.

[15] F.F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proc. of the 12th ACM Symp. on Theory of Computing*, pages 429–435, May 1980.