

## A LINEAR-TIME ALGORITHM FOR CONCAVE ONE-DIMENSIONAL DYNAMIC PROGRAMMING \*

Zvi GALIL

*Department of Computer Science, 450 Computer Science Building, Columbia University, New York, NY 10027, USA*  
*Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel*

Kunsoo PARK

*Department of Computer Science, 450 Computer Science Building, Columbia University, New York, NY 10027, USA*

Communicated by David Gries

Received 2 May 1989

Revised 1 September 1989

**Keywords:** Dynamic programming, quadrangle inequality, total monotonicity

The one-dimensional dynamic programming problem is defined as follows: given a real-valued function  $w(i, j)$  for integers  $0 \leq i \leq j \leq n$  and  $E[0]$ , compute

$$E[j] = \min_{0 \leq i < j} \{ D[i] + w(i, j) \} \quad \text{for } 1 \leq j \leq n,$$

where  $D[i]$  is computed from  $E[i]$  in constant time. The least weight subsequence problem [4] is a special case of the problem where  $D[i] = E[i]$ . The modified edit distance problem [3], which arises in molecular biology, geology, and speech recognition, can be decomposed into  $2n$  copies of the problem.

Let  $A$  be an  $n \times m$  matrix.  $A[i, j]$  denotes the element in the  $i$ th row and the  $j$ th column.  $A[i:i', j:j']$  denotes the submatrix of  $A$  that is the intersection of rows  $i, i+1, \dots, i'$  and columns  $j, j+1, \dots, j'$ . We say that the cost function  $w$  is *concave* if it satisfies the quadrangle inequality [7]

$$w(a, c) + w(b, d) \leq w(b, c) + w(a, d),$$

for  $a \leq b \leq c \leq d$ .

\* Work supported in part by NSF Grants CCR-86-05353 and CCR-88-14977.

In the concave one-dimensional dynamic programming problem  $w$  is concave as defined above. A condition closely related to the quadrangle inequality was introduced by Aggarwal et al. [1]. An  $n \times m$  matrix  $A$  is *totally monotone* if for all  $a < b$  and  $c < d$ ,

$$A[a, c] > A[b, c] \Rightarrow A[a, d] > A[b, d].$$

Let  $r(j)$  be the smallest row index such that  $A[r(j), j]$  is the minimum value in column  $j$ . Then total monotonicity implies

$$r(1) \leq r(2) \leq \dots \leq r(m). \quad (*)$$

That is, the minimum row indices are nondecreasing. We say that an element  $A[i, j]$  is *dead* if  $i \neq r(j)$ . A submatrix of  $A$  is dead if all of its elements are dead. Note that the quadrangle inequality implies total monotonicity, but the converse is not true. Aggarwal et al. [1] show that the row maxima of a totally monotone  $n \times m$  matrix  $A$  can be found in  $O(n + m)$  time if  $A[i, j]$  for any  $i, j$  can be computed in constant time. Their algorithm is easily adapted to find the column minima. We will refer to their algorithm as the SMAWK algorithm.

Let  $B[i, j] = D[i] + w(i, j)$  for  $0 \leq i < j \leq n$ . We say that  $B[i, j]$  is *available* if  $D[i]$  is known and



therefore  $B[i, j]$  can be computed in constant time. Then the problem is to find the column minima in the upper triangular matrix  $B$  with the restriction that  $B[i, j]$  is available only after the column minima for columns  $1, 2, \dots, i$  have been found. It is easy to see that when  $w$  satisfies the quadrangle inequality,  $B$  also satisfies the quadrangle inequality. For the concave problem Hirschberg and Larmore [4] and later Galil and Giancarlo [3] gave  $O(n \log n)$  algorithms using queues. Wilber [6] proposed an  $O(n)$  time algorithm when  $D[i] = E[i]$ . However, his algorithm does not work if the availability of matrix  $B$  must be obeyed, which happens when many copies of the problem proceed simultaneously (i.e., the computation is interleaved among many copies) as in the modified edit distance problem [3] and the mixed convex and concave cost problem [2]. Eppstein [2] extended Wilber's algorithm for interleaved computation. Our algorithm is more general than Eppstein's; it works for any totally monotone matrix  $B$  (we use only relation (\*)), whereas Eppstein's algorithm works only when  $B[i, j] = D[i] + w(i, j)$ . Our algorithm is also simpler than both Wilber's and Eppstein's. Recently, Larmore and Schieber [5] reported another linear-time algorithm, which is quite different from ours.

The algorithm consists of a sequence of iterations. Fig. 1 shows a typical iteration. We use  $N[j]$ ,  $1 \leq j \leq n$ , to store interim column minima

before row  $r$ ;  $N[j] = B[i, j]$  for some  $i < r$  (the usage will be clear shortly). At the beginning of each iteration the following invariants hold:

- (a)  $0 \leq r$  and  $r < c$ .
- (b)  $E[j]$  for all  $1 \leq j < c$  have been found.
- (c)  $E[j]$  for  $j \geq c$  is  $\min(N[j], \min_{i \geq r} B[i, j])$ .

Invariant (b) means that  $D[i]$  for all  $0 \leq i < c$  are known, and therefore  $B[i, j]$  for  $0 \leq i < c$  and  $c \leq j \leq n$  is available. Initially,  $r = 0$ ,  $c = 1$ , and all  $N[j]$  are  $+\infty$ .

Let  $p = \min(2c - r, n)$ , and let  $G$  be the union of  $N[c:p]$  and  $B[r:c-1, c:p]$ ,  $N[c:p]$  as its first row and  $B[r:c-1, c:p]$  as the other rows.  $G$  is a  $(c - r + 1) \times (c - r + 1)$  matrix unless  $2c - r > n$ . Let  $F[j]$ ,  $c \leq j \leq p$ , denote the column minima of  $G$ . Since matrix  $G$  is totally monotone, we use the SMAWK algorithm to find the column minima of  $G$ . Once  $F[c:p]$  are found, we compute  $E[j]$  for  $j = c, c+1, \dots$  as follows. Obviously,  $E[c] = F[c]$ . For  $c+1 \leq j \leq p$ , assume inductively that  $B[c:j-2, j:p]$  ( $\beta$  in Fig. 1) is dead and  $B[j-1, j:n]$  is available. It is trivially true when  $j = c+1$ . By the assumption  $E[j] = \min(F[j], B[j-1, j])$ .

- (1) If  $B[j-1, j] < F[j]$ , then  $E[j] = B[j-1, j]$ , and by relation (1)  $B[r:j-2, j:n]$  ( $\alpha, \beta, \gamma$ , and the part of  $G$  above  $\beta$  in Fig. 1) and  $N[j:n]$  are dead. We start a new iteration with  $c = j+1$  and  $r = j-1$ .
- (2) If  $F[j] \leq B[j-1, j]$ , then  $E[j] = F[j]$ . We compare  $B[j-1, p]$  with  $F[p]$ .
  - (2.1) If  $B[j-1, p] < F[p]$ ,  $B[r:j-2, p+1:n]$  ( $\alpha$  and  $\gamma$  in Fig. 1) is dead by relation (\*).  $B[c:j-2, j:p]$  ( $\beta$  in Fig. 1) is dead by the assumption. Thus only  $F[j+1:p]$  among  $B[0:j-2, j+1:n]$  may become column minima in the future computation. We store  $F[j+1:p]$  in  $N[j+1:p]$  and start a new iteration with  $c = j+1$  and  $r = j-1$ .
  - (2.2) If  $F[p] \leq B[j-1, p]$ ,  $B[j-1, j:p]$  ( $\delta$  in Fig. 1) is dead by relation (\*) in submatrix  $B[r:j-1, j:p]$  ( $\beta, \delta$ , and the part of  $G$  above  $\beta$ ). Since  $B[j, j+1:n]$  is available from  $E[j]$ , the assumption holds at  $j+1$ . We go on to column  $j+1$ .

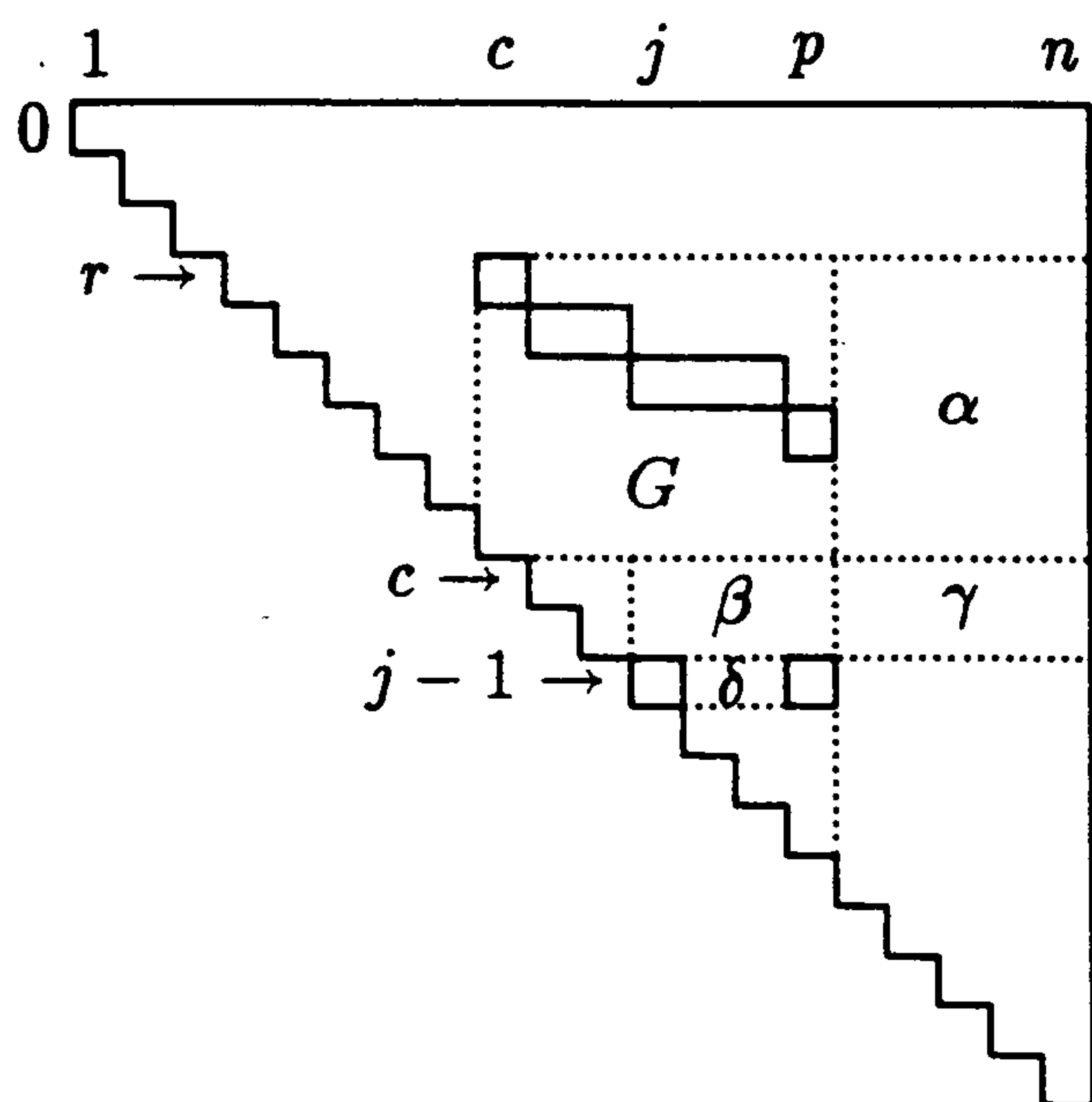


Fig. 1. Matrix  $B$  at a typical iteration.



```

procedure concave 1D
   $c \leftarrow 1$ ;
   $r \leftarrow 0$ ;
   $N[1:n] \leftarrow +\infty$ ;
  while  $c \leq n$  do
     $p \leftarrow \min(2c - r, n)$ ;
    use SMAWK to find column minima  $F[c:p]$  of  $G$ ;
     $E[c] \leftarrow F[c]$ ;
    for  $j \leftarrow c + 1$  to  $p$  do
      if  $B[j-1, j] < F[j]$  then
         $E[j] \leftarrow B[j-1, j]$ ;
        break
      else
         $E[j] \leftarrow F[j]$ ;
        if  $B[j-1, p] < F[p]$  then
           $N[j+1:p] \leftarrow F[j+1:p]$ ;
          break
        end if
      end if
    end for
    if  $j \leq p$  then
       $c \leftarrow j + 1$ ;
       $r \leftarrow j - 1$ 
    else
       $c \leftarrow p + 1$ ;
       $r \leftarrow \max(r, \text{row of } F[p])$ 
    end if
  end while
end

```

Fig. 2. The algorithm for concave 1D dynamic programming.

If case (2.2) is repeated until  $j = p$ , we have found  $E[j]$  for  $c \leq j \leq p$ . We start a new iteration with  $c = p + 1$ . If the row of  $F[p]$  is greater than  $r$ , it becomes the new  $r$  (it may be smaller than  $r$  if it is the row of  $N[p]$ ). Note that the three invariants hold at the beginning of new iterations. Figure 2 shows the algorithm, where the **break** statement causes the innermost enclosing loop to be exited immediately.

Each iteration takes time  $O(c - r)$ . If either case (1) or case (2.1) happens, we charge the time to rows  $r, \dots, c - 1$  because  $r$  is increased by  $(j - 1) - r \geq c - r$ . If case (2.2) is repeated until  $j = p$ , there are two cases. If  $p < n$ , we charge the time to columns  $c, \dots, p$  because  $c$  is increased by  $(p + 1) - c \geq c - r + 1$ . If  $p = n$ , we have finished the whole computation, and rows  $r, \dots, c - 1$  ( $< n$ ) have not been charged yet; we charge the time to the rows. Since  $c$  and  $r$  never decrease, only constant time is charged to each row or column. Thus the total time of the algorithm is linear in  $n$ .

## References

- [1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* 2 (1987) 195–208.
- [2] D. Eppstein, Sequence comparison with mixed convex and concave costs, *J. Algorithms*, to appear.
- [3] Z. Galil and R. Giancarlo, Speeding up dynamic programming with applications to molecular biology, *Theoret. Comput. Sci.* 64 (1989) 107–118.
- [4] D.S. Hirschberg and L.L. Larmore, The least weight subsequence problem, *SIAM J. Comput.* 16 (4) (1987) 628–638.
- [5] L.L. Larmore and B. Schieber, On-line dynamic programming with applications to the prediction of RNA secondary structure, to be presented at 1st Ann. ACM–SIAM Symp. on Discrete Algorithms.
- [6] R. Wilber, The concave least-weight subsequence problem revisited, *J. Algorithms* 9 (1988) 418–425.
- [7] F.F. Yao, Speed-up in dynamic programming, *SIAM J. Algebraic Discrete Methods* 3 (1982) 532–540.