A SIMPLE LINEAR TIME ALGORITHM FOR CONCAVE ONE-DIMENSIONAL DYNAMIC PROGRAMMING

by

Maria M. Klawe

Technical Report 89-16

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5 Canada

A Simple Linear Time Algorithm for Concave One-Dimensional Dynamic Programming

by
Maria M. Klawe
The University of British Columbia

Following [KK89] we will say that an algorithm for finding the column minima of a matrix is **ordered** if the algorithm never evaluates the (i,j) entry of the matrix until the minima of columns $1,2,\ldots,i$ are known. This note presents an extremely simple linear time ordered algorithm for finding column minima in triangular totally monotone matrices. Analagous to [KK89] this immediately yields a linear time algorithm for the concave one-dimensional dynamic programming problem. Wilber [W88] gave the first linear time algorithm for the concave one-dimensional dynamic programming problem, but his algorithm was not ordered and hence could not be applied in some situations. Examples of these situations are given in [GP89] and [L89]. Galil and Park [GP89] and Larmore [L89] independently found quite different ordered linear time algorithms. All of these algorithms, and ours as well, rely on the original linear-time algorithm known as SMAWK for finding column minima in totally monotone matrices [AKMSW87]. The constant in our algorithm is essentially the same of that of the Galil-Park algorithm, and since our algorithm is so simple to program, we expect it to be the algorithm of choice in implementations.

Let $w(i,j)$ for integers $0 \leq i \leq j \leq n$ be a real-valued function such that for all $a \leq b \leq c \leq d$ we have $w(a,c)+w(b,d) \leq w(b,c)+w(a,d)$. The concave one-dimensional programming problem is to compute, given $E[0]$, the values $E[j] = \min_{0\leq i<j} D[i]+w(i,j)$ for $j = 1,\ldots,n$, where $D[i]$ can be computed from $E[i]$ in constant time. Let $M(i,j)$ for $1 \leq j \leq n$, $0 \leq i < j$ be the upper triangular matrix defined by $M(i,j) = D[i] + w(i,j)$. This matrix is shown in Figure 1. We will call M an upper-triangular matrix of size n. Clearly computing the $E[j]$ is equivalent to finding the column minima of M, and the ordering condition for a column-minima finding algorithm is simply a condition to guarantee that we can always evaluate an entry of M in constant time.

Following [GP89] we say that an element $M(i,j)$ is **available** if the column minima of columns $1,2,\ldots,i$ are already known. It is well-known (see [Y82] for example) and easy to check that the property that $w(a,c)+w(b,d) \leq w(b,c)+w(a,d)$ for all $a \leq b \leq c \leq d$ implies that M has the property that whenever $0 \leq h < i < j$, if $M(h,j) < M(i,j)$ then $M(i,k)$ is not a column minima for $i+1 \leq k \leq j$, and if $M(h,j) \geq M(i,j)$ then $M(h,k)$ is not a column minima for $j \leq k \leq n$. This is essentially (up to reflection) the total monotonicity property introduced in [AKMSW87]. Figure 2 shows the regions eliminated from containing column minima in each case.

Clearly an algorithm to find column minima is ordered if it only evaluates elements of the matrix which are available. We will say that an element of M has been **treated** if it has either been evaluated or has been shown not to be a column-minima. Similarly, a region of the matrix has been treated if every element in that region has been treated.

It is easy to see that any ordered algorithm which treats the entire matrix M can trivially be converted to an ordered algorithm which finds the column minima of M with at most a doubling of the time. Thus we will concentrate on giving a simple algorithm to treat the matrix M which only evaluates an entry $M(i,j)$ if the region consisting of columns $0,\ldots,i$ has already been treated. At the end of this note we indicate how our "treatment" algorithm can be converted to a column minima finding algorithm with only a small additive increase in the multiplicate constant.

The overall structure of the algorithm is that there is a fixed constant c, such that for some q with $0 < q \leq n/2$, in cq time the algorithm will treat the region consisting of rows $0,\ldots,q-1$. Since the remaining untreated region of the matrix is an upper-triangular matrix of size $n-q$, by applying the algorithm recursively we obtain an algorithm running in time cn. The algorithm functions by building two types of treated regions in the matrix, squares and slices. The algorithm will interleave the building of these two types of regions.

For $k \geq 0$ we define the k-square to be the region $M(0 : 2^k - 1,\ 2^k : 2^{k+1} - 1)$. Figure 3(a) shows the k squares for $k = 0, 1, 2$ and 3. For $i \neq 2^{\lceil \log i \rceil} - 1$ we define the i-slice to be the region $M(i,\ i+1 : 2^{\lceil \log i \rceil} - 1)$. Figure 3(b) shows the i-slices for $i = 2, 4, 5$ and 6. It is easy to verify that for any s the union of the i slices for $i \leq s$ with the k-squares for $k \leq \lfloor \log(s+1) \rfloor$ is exactly the region $M(0 : s,\ 1 : 2^{\lfloor \log(s+1) \rfloor + 1} - 1)$. In particular, this region covers the columns $1,\ldots,s$ so after we have treated the i-slices for $i \leq s$ and the k-squares for $k \leq \lfloor \log(s+1) \rfloor$, we may evaluate any entry in rows $0,\ldots,s+1$ of the matrix.

We now give the algorithm, and then explain how it works.

The Algorithm
$i \leftarrow 0; r \leftarrow 0;$ *initialization*

While $i < 2^{\lceil \log(n+1) \rceil}$ and $r < i$ do;
  If $i = 2^{\lceil \log i \rceil} - 1$
    then do;
      SMAWK $M(0 : i,\ i+1 : \min(2(i+1)-1, n))$;
      * a square has been treated*
      $i \leftarrow i + 1$;
      end;
    else if $M(r, 2^{\lfloor \log(i+1) \rfloor + 1} - 1) < M(i, 2^{\lfloor \log(i+1) \rfloor + 1} - 1)$
      then $i \leftarrow i + 1$;
      * a slice has been treated*
      else $r \leftarrow r + 1$;
      *the region $M(r,\ 2^{\lfloor \log(i+1) \rfloor + 1} - 1 : n)$ has been treated *
  end;

The algorithm has now treated rows $0,\ldots,i-1$.

Before considering the correctness and timing of the algorithm, we start with an

intuitive explanation of what is going on. The basic idea of the algorithm is depending on whether $i = 2^{\lceil \log i \rceil} - 1$ or not, the algorithm either treats the k square where $k = \lceil \log i \rceil$ or attempts to treat the $i - slice$ by comparing $M(r, 2^{\lfloor \log(i+1) \rfloor + 1} - 1)$ with $M(i, 2^{\lfloor \log(i+1) \rfloor + 1} - 1)$. If $M(r, 2^{\lfloor \log(i+1) \rfloor + 1} - 1) < M(i, 2^{\lfloor \log(i+1) \rfloor + 1} - 1)$ then we succeed in treating the i-slice and go on to the next value of i. If not, then we know that the region $M(r,\ 2^{\lfloor \log(i+1) \rfloor + 1} - 1 : n)$ contains no column minima so we increment r. It is easy to prove that by induction that at the beginning of each iteration of the while loop we have treated the regions $M(0 : i-1,\ 1 : \min(2^{\lfloor \log i \rfloor + 1} - 1, n))$ and $M(0 : r-1,\ 1 : n)$. Since the algorithm only evaluates elements in rows with indices at most i it clearly is an ordered algorithm. Thus in order to prove its correctness it suffices to show that when it exits the while loop every element in rows $0, \ldots, i-1$ has been treated. This is obvious from the previous observation if $r = i$ so we may assume $i = 2^{\lceil \log(n+1) \rceil}$. However, now it is easy to check that this implies $2^{\lfloor \log i \rfloor + 1} - 1 \geq n$, and hence by the previous observation we are done.

We now analyze the timing of the algorithm. Let a be a constant such that the SMAWK algorithm uses at most an time to find the column minima of an $n \times n$ totally monotone matrix. The amount of time used in applying the SMAWK algorithm is at most $\sum_{k=0}^{\lfloor \log i \rfloor} a2^k \leq a2^{\lfloor \log i \rfloor + 1} \leq 2ai$. The additional work consists of incrementing the variables i and r and performing the comparisons. The number of increments is clearly bounded by $2i$ and the number of comparisons is simply the number of passes through the while loop. Since either i or r is incremented on each pass through the while loop, the total number of comparisons is also at most $2i$. Thus the total amount of additional work is bounded by $4i$. Combining all this we see that the amount of time required by the algorithm is at most $i(4 + 2a)$ which is obviously linear in i.

Comparing the constants.

In order to fairly compare our algorithm with the Galil-Park algorithm we must convert it to a column minima finding algorithm rather than simply a "treatment algorithm". The only change necessary is that after the algorithm treats the rows $0, \ldots, i-1$, before applying it to the remaining upper triangular matrix of size $n-i$ we must "merge" the column minima found in the last square which we SMAWKed with the values in the corresponding columns in row i of the matrix. One possibility for this merging is as follows. For each column in the last square to be SMAWKed we replace the value in row i with the minimum of that value and the column minimum in the SMAWKed square. This adds at most an additional i comparisons and assignments. A solution which does better on average, at least with respects to the effect on the constants, is to add a dummy top row to the remaining matrix containing the column minima for columns in the most recently SMAWKed square, and infinity in the columns to the right of most recently SMAWKed square.

Since neither our algorithm nor the Galil-Park algorithm has been completely optimized with respect to constants, it only seems to make sense to compare the dominant cast which is the application of the SMAWK algorithm to square submatrices of the

original triangular matrix. In both cases, in the worst case the sum of the number of rows in the square submatrices is approximately $2n$, so with respect to the use of the SMAWK algorithm the two algorithms are essentially equivalent.

References

[AKMSW87] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix searching algorithm, Algorithmica 2(1987), pp. 195-208.

[AK87] A. Aggarwal and M. Klawe, Applications of generalized matrix searching to geometric algorithms, to appear in Discrete Applied Math.

[AP88] A. Aggarwal and J. Park, Notes on searching in multidimensional arrays, Proc. 29th Ann. IEEE Symposium on Found. Comp. Sci. (1988), pp.497-512.

[EGG88] D. Eppstein, Z. Galil and R Giancarlo, Speeding up dynamic programming, Proc. 29th Ann. IEEE Symposium on Found. Comp. Sci. (1988), pp.488-496.

[GP89] Z. Galil and K. Park, A linear-time algorithm for concave one-dimensional dynamic programming, preprint 1989.

[HL87] D.S. Hirschberg and L.L. Larmore, The least weight subsequence problem, SIAM J. Computing 16, 1987, pp. 628-638.

[L89] L. Larmore, On-line dynamic programming and application to Waterman's RNA problem, preprint 1989.

[W88] R. Wilber, The concave least weight subsequence revisited, J. Algorithms 9 (1988), pp.418-425.

[Y82] F. Yao, Speed-up in Dynamic Programming, SIAM J. Alg. Disc. Methods 3, 1982, pp. 532-540.

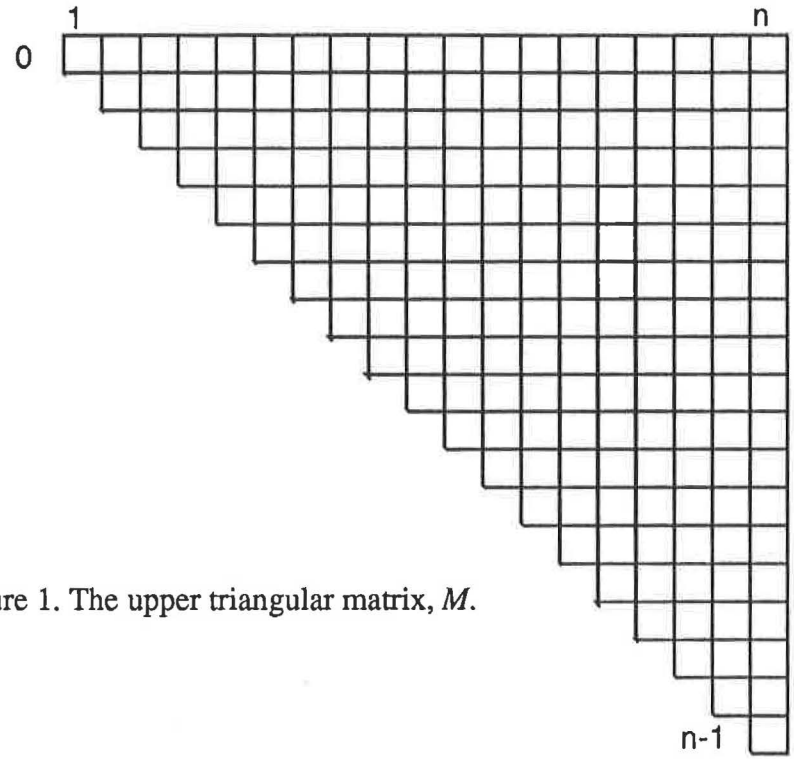

Figure 1. The upper triangular matrix, M.

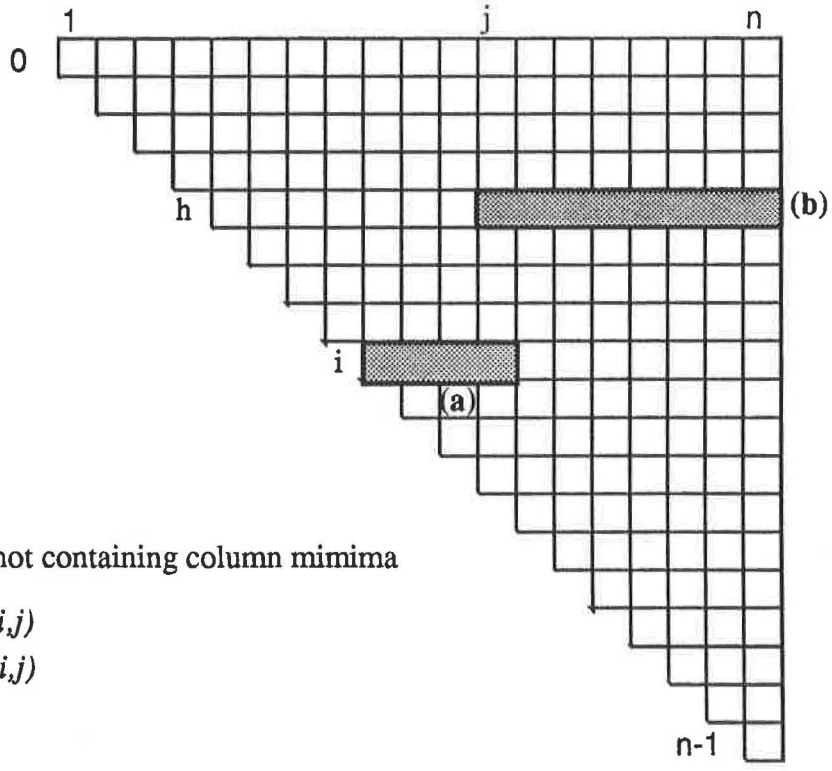

Figure 2. Regions not containing column mimima

(a) If $M(h,j) < M(i,j)$

(b) If $M(h,j) \leq M(i,j)$

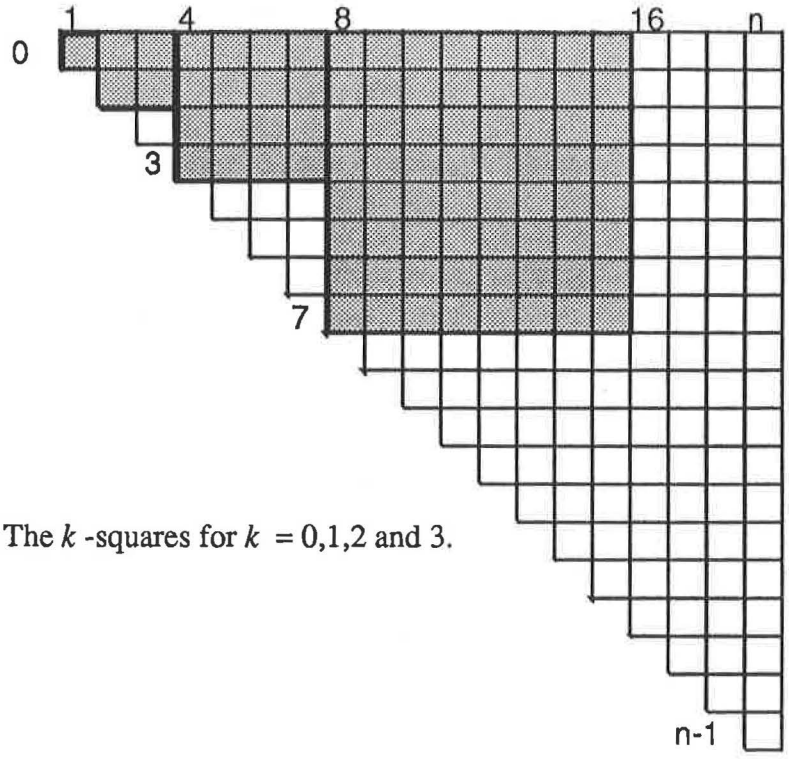

Figure 3(a). The k -squares for k = 0,1,2 and 3.

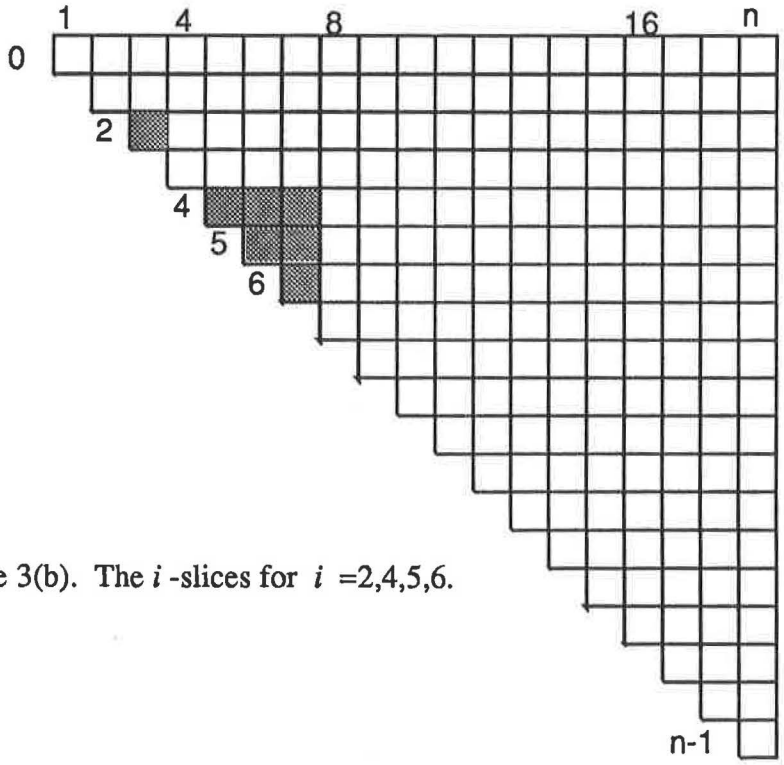

Figure 3(b). The i -slices for i =2,4,5,6.