Online Dynamic Programming Speedups*

Amotz Bar-Noy[1], Mordecai J. Golin[2], and Yan Zhang[2]

[1] Brooklyn College, 2900 Bedford Avenue Brooklyn, NY 11210
amotz@sci.brooklyn.cuny.edu
[2] Hong Kong University of Science and Technology, Kowloon, Hong Kong
{golin,cszy}@cse.ust.hk

Abstract. Consider the Dynamic Program $h(n) = \min_{1 \le j \le n} a(n,j)$ for $n = 1, 2, \ldots, N$. For arbitrary values of $a(n,j)$, calculating all the $h(n)$ requires $\Theta(N^2)$ time. It is well known that, if the $a(n,j)$ satisfy the *Monge property*, then there are techniques to reduce the time down to $O(N)$. This speedup is inherently static, i.e., it requires N to be known in advance.

In this paper we show that if the $a(n,j)$ satisfy a stronger condition, then it is possible, without knowing N in advance, to compute the values of $h(n)$ in the order of $n = 1, 2, \ldots, N$, in $O(1)$ amortized time per $h(n)$. This *maintains the DP speedup online*, in the sense that the time to compute all $h(n)$ is $O(N)$. A slight modification of our algorithm restricts the worst case time to be $O(\log N)$ per $h(n)$, while maintaining the amortized time bound. For $a(n,j)$ that satisfy our stronger condition, our algorithm is also simpler to implement than the standard Monge speedup.

We illustrate the use of our algorithm on two examples from the literature. The first shows how to make the speedup of the D-median on a line problem in an online settings. The second shows how to improve the running time for a DP used to reduce the amount of bandwidth needed when paging mobile wireless users.

1 Introduction

Consider the class of problems defined by

$$h(n) = \min_{1 \le j \le n} a(n,j), \quad \forall\ 1 \le n \le N \tag{1}$$

where the goal is to compute $h(n)$ for $1 \le n \le N$. In many applications, (1) is a Dynamic Program (DP), in the sense that the values of $a(n,j)$ depend upon $h(i)$, for some $1 \le i < n$. In this paper, we always assume any particular $a(n,j)$ can be computed in $O(1)$ time, provided that the values of $h(i)$ it depends on are known. For a generally defined function $a(n,j)$, it requires $\Theta(N^2)$ time to compute all the $h(n)$. It is well known, though [1], that if the values of $a(n,j)$ satisfy the *Monge property* (see Section 1.1), then the SMAWK algorithm [2] can compute all the $h(n)$, for $1 \le n \le N$, in $O(N)$ time. To be precise, if

* The research of the second and third authors was partially supported by Hong Kong RGC CERG grant HKUST6312/04E.

T. Erlebach and C. Kaklamanis (Eds.): WAOA 2006, LNCS 4368, pp. 43–54, 2006.

1. the value of N is known in advance;
2. and for any $1 \leq j \leq n \leq N$, the value of $a(n, j)$ can be computed in $O(1)$ time, i.e., $a(n, j)$ does not depend on $h(i)$;
3. and the values of $a(n, j)$ satisfy the Monge property defined by (4),

then the SMAWK algorithm [2] can compute all of the $h(n)$ for $1 \leq n \leq N$ in $O(N)$ time.

The main purpose of this paper is to consider the DP formula (1) in *online* settings. By this we mean that the values of $h(n)$ are computed in the order $n = 1, 2, \ldots, N$ *without* knowing the parameter N in advance, and the values of $a(n, j)$ are allowed to depend on *all* previously-computed values of $h(i)$ for $1 \leq i < n$. To be precise, our main result is

Theorem 1. *Consider the DP defined by (1). If*

1. $\forall\ 1 \leq j \leq n \leq N$, *the value of* $a(n, j)$ *can be computed in* $O(1)$ *time, provided that the values of* $h(i)$ *for* $1 \leq i < n$ *are known;*
2. *and* $\forall\ 1 \leq j < n \leq N$,

$$a(n, j) - a(n-1, j) = c_n + \delta_j \beta_n \tag{2}$$

where c_n, β_n *and* δ_j *are constants satisfying*
(a) $\forall\ 1 < n \leq N$, $\beta_n \geq 0$;
(b) and $\delta_1 > \delta_2 > \cdots > \delta_{N-1}$,

then, there is an algorithm that computes the values of $h(n)$ *in the order* $n = 1, 2, \ldots, N$ *in* $O(1)$ *amortized and* $O(\log N)$ *worst-case time per* $h(n)$. *The algorithm does not know the value of* N *until* $h(N)$ *has been computed.*

We call the Condition 2 in Theorem 1 (including Conditions (a) and (b)) the *online Monge property*. As we will see in Section 1.1, the online Monge property is a stronger Monge property. The SMAWK algorithm is a $\Theta(N)$ speedup of the computation of (1) when $a(n, j)$ satisfy the Monge property. Theorem 1 says that if $a(n, j)$ satisfy the online Monge property, then the same speedup can be maintained online, in the sense that the time to compute all $h(n)$ is still $O(N)$. Section 2 will give the main algorithm, which achieves the $O(1)$ amortized bound. In Section 2.3, we modify the algorithm a little bit to achieve the worst case $O(\log N)$ bound. Section 3 shows two applications of this technique.

Note that the online Monge property only says that c_n, β_n and δ_j exist. It does not say that c_n, β_n and δ_j are given. However, if δ_j is given, then the algorithm will be easier to understand. So, throughout this paper we will assume we have an extra condition:

– The values of δ_j can be computed in $O(1)$ time, provided that the values of $h(i)$ for $1 \leq i < j$ are known.

This condition is not really necessary. In Appendix A, we will show how it is implied by other conditions in Theorem 1.

As a final note we point out that there is a body of literature already discussing "online" problems of (1), e.g., [3,4,5,6,7]. We should clarify that the "online" in

those papers actually had a different meaning than the one used here. More specifically, the result they have is that if

1. the value of N is known in advance;
2. and for any $1 \leq j \leq n \leq N$, the value of $a(n, j)$ can be computed in $O(1)$ time, provided that the values of $h(i)$ for $1 \leq i < j$ are known;
3. and the values of $a(n, j)$ satisfy the Monge property defined by (4),

then both the Galil-Park algorithm [6] and the Larmore-Schieber algorithm [7] can compute all of the $h(n)$ for $1 \leq n \leq N$ in $O(N)$ time. As we can see, their definition of "online" is only that the $a(n, j)$ can depend upon part of the previously-computed values of $h(i)$, i.e., for $1 \leq i < j$. It does not mean that $h(n)$ can be computed without knowing the problem size N in advance.

1.1 Relations to Monge

In this section, we briefly introduce the definition of Monge property. See the survey [1] for more details. Consider an $N \times N$ matrix A. Denote by $R(n)$ the *index* of the rightmost minimum of row n of A, i.e.,

$$R(n) = \max\{j : A_{n,j} = \min_{1 \leq i \leq N} A_{n,i}\}.$$

A matrix A is *monotone* if $R(1) \leq R(2) \leq \cdots \leq R(N)$, A is *totally monotone* if all submatrices[1] of A are monotone. The SMAWK algorithm [2] says that if A is totally monotone, then it can compute all of the $R(n)$ for $1 \leq n \leq N$ in $O(N)$ time.

For our problem, if we set

$$A_{n,j} = \begin{cases} a(n, j) & 1 \leq j \leq n \leq N \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

then $h(n) = a(n, R(n))$. Hence, if we can show the matrix A defined by (3) is totally monotone, then the SMAWK algorithm can solve our problem (offline version) in $O(N)$ time. Totally monotone properties are usually established by showing a slightly stronger property, the Monge Property (also known as the *quadrangle inequality*). A matrix A is Monge if $\forall\ 1 \leq n < N$ and $\forall\ 1 \leq j < N$,

$$A_{n,j} + A_{n+1,j+1} \leq A_{n+1,j} + A_{n,j+1}.$$

It is easy to show that A is totally monotone if it is Monge. So, for the offline version of our problem, we only need to show that the matrix A defined by (3) is Monge, i.e., $\forall\ 1 \leq j < n < N$,

$$a(n, j) + a(n + 1, j + 1) \leq a(n + 1, j) + a(n, j + 1). \tag{4}$$

[1] In this paper, submatrices can take non-consecutive rows and columns from the original matrix, and are not necessarily square matrices.

By the conditions in Theorem 1,

$$a(n+1,j)+a(n,j+1)-a(n,j)-a(n+1,j+1)=(\delta_j-\delta_{j+1})\beta_{n+1}\geq 0.$$

So, the matrix A defined by (3) is Monge, and the SMAWK algorithm solves the offline problem.

Our problem is a special case of Monge. But how special a case? Referring to Section 2.2 of [1] for more details, we see that if we only consider the finite entries, then a matrix A is Monge if and only if $\forall\ A_{n,j}\neq\infty$,

$$A_{n,j}=P_n+Q_j+\sum_{k=n}^{N}\sum_{i=1}^{j}F_{ki} \tag{5}$$

where P and Q are vectors, and F is an $N\times N$ matrix, called the *distribution matrix*, whose entries are all nonnegative. For our problem, let $\delta_0=\delta_1$. Then

$$\begin{aligned}a(n,j)&=a(N,j)-\sum_{k=n+1}^{N}c_k-\delta_j\sum_{k=n+1}^{N}\beta_k\\&=a(N,j)-\sum_{k=n+1}^{N}c_k-\delta_0\sum_{k=n+1}^{N}\beta_k+(\delta_0-\delta_j)\sum_{k=n+1}^{N}\beta_k\end{aligned}$$

So, in our problem,

$$P_n=-\sum_{k=n+1}^{N}(c_k+\delta_0\beta_k),\qquad Q_j=a(N,j),\qquad F_{ki}=(\delta_{i-1}-\delta_i)\beta_{k+1},$$

where we define $\beta_{N+1}=0$. This shows that our problem is a special case of the Monge property where the distribution matrix has rank 1.

Conversely, if the distribution matrix F has rank 1, then the values of $a(n,j)$ satisfy the conditions of Theorem 1. So, Theorem 1 is really showing that the row minima of any Monge matrix defined by a rank 1 distribution matrix can be found online.

2 The Algorithm

In this section, we show the main algorithm that achieves the $O(1)$ amortized bound in Theorem 1. We will show the algorithm at step n, where the values of $h(i)$ have been computed for $1\leq i<n$, and we want to compute the value of $h(n)$. By the conditions in Theorem 1 and the extra condition, all the values $a(n,j)$ and δ_j for $1\leq j\leq n\leq N$ are known.

The key concept of the algorithm is a set of straight lines defined as follows.

Definition 2. *$\forall\ 1\leq j\leq n\leq N$, we define*

$$L_j^n(x)=a(n,j)+\delta_j\cdot x \tag{6}$$

So, $h(n) = \min_{1 \le j \le n} L_j^n(0)$. To compute $\min_{1 \le j \le n} L_j^n(x)$ at $x = 0$ efficiently, the algorithm maintains $\min_{1 \le j \le n} L_j^n(x)$ for the entire range $x \ge 0$, i.e., at step n, the algorithm maintains the *lower envelope* of the set of lines $\{L_j^n(x) : 1 \le j \le n\}$ in the range $x \in [0, \infty)$.

2.1 The Data Structure

The only data structure used is an array, called the *active-indices array*, $Z = (z_1, \ldots, z_t)$ for some length t. It will be used to represent the lower envelope. It stores, from left to right, the indices of the lines that appear on the lower envelope in the range $x \in [0, \infty)$. That is, at step n, if we walk along the lower envelope from $x = 0$ to the right, then we will sequentially encounter the lines $L_{z_1}^n(x), L_{z_2}^n(x), \ldots, L_{z_t}^n(x)$. Since $\delta_1 > \delta_2 > \cdots > \delta_n$, and by the properties of lower envelopes, we have $z_1 < z_2 < \cdots < z_t = n$, and no line can appear more than once in the active-indices array.

Once we have the active-indices array, computing $h(n)$ becomes easy as $h(n) = a(n, z_1)$. So, the problem is how to obtain the active-indices array. Inductively, when the algorithm enters step n from step $n-1$, it maintains an active-indices array for step $n-1$, which represents the lower envelope of the lines $\{L_j^{n-1}(x) : 1 \le j \le n-1\}$. So, the main part of the algorithm is to *update* the old active-indices array to the new active-indices array for $\{L_j^n(x) : 1 \le j \le n\}$.

Before introducing the algorithm, we introduce another concept, the *break-point* array, $X = (x_0, \ldots, x_t)$, where $x_0 = 0$, $x_t = \infty$ and x_i $(1 \le i < t)$ is the x-coordinate of the intersection point of lines $L_{z_i}^n(x)$ and $L_{z_{i+1}}^n(x)$. The break-point array is *not* stored explicitly, since for any i, the value of x_i can be computed in $O(1)$ time, given the active-indices array.

2.2 The Main Algorithm

In step n, we need to consider n lines $\{L_j^n(x) : 1 \le j \le n\}$. The algorithm will first deal with the $n-1$ lines $\{L_j^n(x) : 1 \le j \le n-1\}$, and then add the last line $L_n^n(x)$. Figure 1 illustrates the update process by an example. Figure 1(a) shows what we have from step $n-1$, Figure 1(b) shows the considerations for the first $n-1$ lines, and Figure 1(c) shows the adding of the last line.

Deal with the first $n-1$ lines. For the first $n-1$ lines $\{L_j^n(x) : 1 \le j \le n-1\}$, the key observation is the following lemma.

Lemma 3. *$\forall\ 1 < n \le N$ and $\forall\ x$,*

$$L_j^n(x) = L_j^{n-1}(x + \beta_n) + c_n, \qquad \forall\ 1 \le j \le n-1.$$

Proof. By (2) and (6),

$$\begin{aligned} L_j^n(x) &= [a(n, j) - \delta_j \beta_n] + \delta_j (x + \beta_n) \\ &= [a(n-1, j) + c_n] + \delta_j (x + \beta_n) \\ &= L_j^{n-1}(x + \beta_n) + c_n. \end{aligned}$$

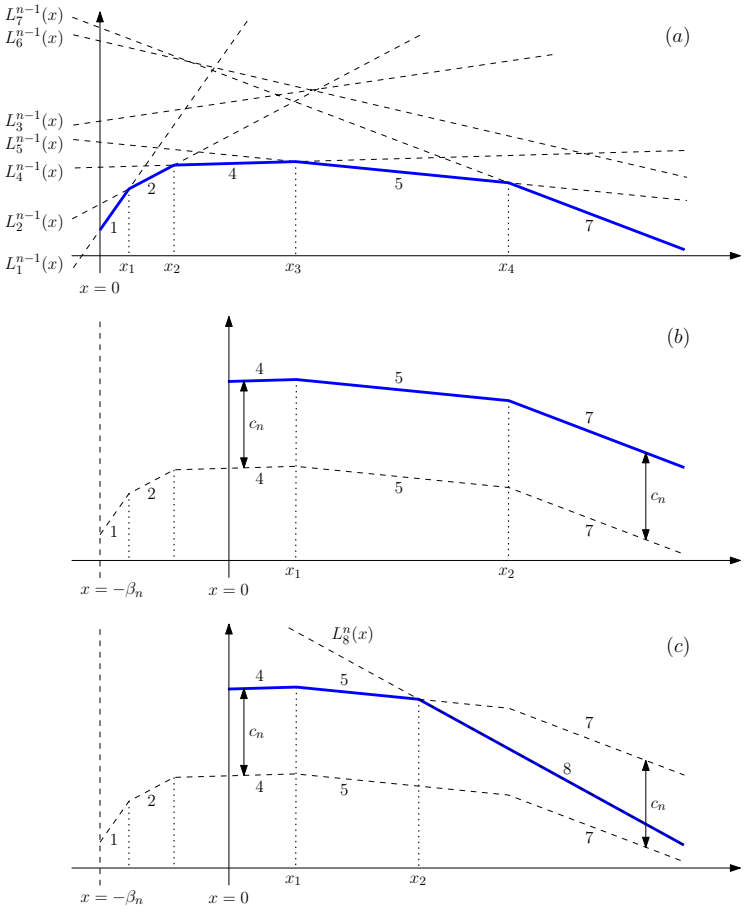

Fig. 1. The update of the active-indices array from Step $n-1$ to Step n, where $n = 8$. The thick solid chains are the lower envelopes. Figure (a) shows the lower envelope for the lines $\{L_j^{n-1}(x) : 1 \le j \le n-1\}$, Figure (b) shows the lower envelope for the lines $\{L_j^n(x) : 1 \le j \le n-1\}$, and Figure (c) shows the lower envelope for the lines $\{L_j^n(x) : 1 \le j \le n\}$. The numbers beside the line segments are the indices of the lines. The active-indices array changes from (a)$(1, 2, 4, 5, 7)$, to (b)$(4, 5, 7)$, then to (c)$(4, 5, 8)$.

Lemma 3 says that if we translate the line $L_j^{n-1}(x)$ to the left by β_n and upward by c_n, then we obtain the line $L_j^n(x)$. The translation is independent of j, for $1 \le j \le n-1$. So,

Corollary 4. *The lower envelope of the lines* $\{L_j^n(x) : 1 \le j \le n-1\}$ *is the translation of the lower envelope of* $\{L_j^{n-1}(x) : 1 \le j \le n-1\}$ *to the left by* β_n *and upward by* c_n.

As an example, see Figure 1, (a) and (b). From Figure 1(a) to 1(b), the entire lower envelope translates to the left by β_n and upward by c_n.

We call an active-index z_i *negative* if the part of $L^n_{z_i}(x)$ that appears on the lower envelope is completely contained in the range $x \in (-\infty, 0]$. By Corollary 4, to obtain the active-indices array for $\{L^n_j(x) : 1 \le j \le n-1\}$ from the old active-indices array, we only need to delete those active-indices who becomes negative due to the translation. This can be done by a simple sequential scan. We scan the old active-indices array from left to right, check each active-index whether it becomes negative. If it is, we delete it. As soon as we find the first active-index that is nonnegative, we can stop the scan, since the rest of the indices are all nonnegative.

To be precise, we scan the old active-indices array from z_1 to z_t. For each z_i, we compute x_i, the right break-point of the segment z_i. If $x_i < 0$, then z_i is negative. Let $z_{\min}$ be the first active-index that is nonnegative, then the active-indices array for $\{L^n_j(x) : 1 \le j \le n-1\}$ is $(z_{\min}, \ldots, z_t)$.

Adding the last line. We now add the line $L^n_n(x)$. Recall Condition (a) in Theorem 1. Since $L^n_n(x)$ has the smallest slope over all lines, it must be the rightmost segment on the lower envelope. And since no line can appear on the lower envelope more than once, we only need to find the intersection point between $L^n_n(x)$ and the lower envelope of $\{L^n_j(x) : 1 \le j \le n-1\}$. Assume they intersect on segment $z_{\max}$, then the new lower envelope should be $(z_{\min}, \ldots, z_{\max}, n)$. See Figure 1(c), in the example, $z_{\max} = 5$.

To find $z_{\max}$, we also use a sequential scan, but from right to left. We scan the active-indices array from z_t to $z_{\min}$. For each z_i, we compute x_{i-1}, the left break-point of segment z_i, and compare the values of $L^n_n(x_{i-1})$ and $L^n_{z_i}(x_{i-1})$. If $L^n_n(x_{i-1})$ is smaller, then z_i is deleted from the active-indices array. Otherwise, we find $z_{\max}$.

The running time. The two sequential scans use amortized $O(1)$ time per step, since each line can be added to or deleted from the active-indices array at most once.

2.3 The Worst-Case Bound

To achieve the worst-case bound, we can use binary search to find $z_{\min}$ and $z_{\max}$. Since for a given index z and value x the function $L^n_z(x)$ can be computed in $O(1)$ time, the binary search takes $O(\log N)$ time worst case.

To keep both the $O(1)$ amortized time and the $O(\log N)$ worst-case time, we run both the sequential search and the binary search in parallel, interleaving their steps, stopping when the first one of the two searches completes.

3 Applications

We will now see two applications. Both will require *multiple* applications of our technique, and both will be in the form

$$H(d,n) = \min_{d-1 \le j \le n-1} \left(H(d-1,j) + W^{(d)}_{n,j} \right), \tag{7}$$

where the value of $W^{(d)}_{n,j}$ can be computed in $O(1)$ time, and the values of $H(d,n)$ for $d = 0$ or $n = d$ are given. The goal is to compute $H(D,N)$. Setting

$$a^{(d)}(n,j) = H(d-1,j) + W^{(d)}_{n,j},$$

for each fixed d ($1 \le d \le D$), the values of $a^{(d)}(n,j)$ satisfy the online Monge property in Theorem 1, i.e.,

$$a^{(d)}(n,j) - a^{(d)}(n-1,j) = W^{(d)}_{n,j} - W^{(d)}_{n-1,j} = c^{(d)}_n + \delta^{(d)}_j \beta^{(d)}_n. \tag{8}$$

where $\delta^{(d)}_j$ decreases as j increases, and $\beta^{(d)}_n \ge 0$.

As before, we want to compute $H(d,n)$ in online fashion, i.e., as n increases from 1 to N, at step n, we want to compute the set $\mathcal{H}_n = \{H(d,n) \mid 1 \le d \le D\}$. By Theorem 1, this can be done in $O(D)$ amortized time per step. This gives a total of $O(DN)$ time to compute $H(D,N)$, while the naive algorithm requires $O(DN^2)$ time.

3.1 *D*-Medians on a Directed Line

The first application comes from [8]. It is the classic D-median problem when the underlying graph is restricted to a directed line. In this problem we have N points (users) $v_1 < v_2 < \cdots < v_N$, where we also denote by v_i the x-coordinate of the point. Each user v_i has a *weight*, denoted by w_i, representing the amount of requests. We want to choose a subset $S \subseteq V$ as servers (medians) to provide service to the users' requests. The line is *directed*, in the sense that the requests from a user can only be serviced by a server to its left. So, v_1 must be a server. Denote by $\ell(v_i, S)$ the distance from v_i to the nearest server to its left, i.e., $\ell(v_i, S) = \min\{v_i - v_l \mid v_l \in S, v_l \le v_i\}$. The objective is to choose D servers (not counting v_1) to minimize the *cost*, which is $\sum_{i=1}^{N} w_i \ell(v_i, S)$.

The problem can be solved by the following DP. Let $H(d,n)$ be the minimum cost of servicing $v_1, v_2, \ldots, v_n$ using exactly d servers (not counting v_1). Let $W_{n,j} = \sum_{l=j+1}^{n} w_l(v_l - v_{j+1})$ be the cost of servicing $v_{j+1}, \ldots, v_n$ by server v_{j+1}. Then

$$H(d,n) = \begin{cases} 0 & n = d \\ W_{n,0} & d = 0,\ n \ge 1 \\ \min\limits_{d-1 \le j \le n-1} (H(d-1,j) + W_{n,j}), & 1 \le d < n \end{cases}$$

The optimal cost we are looking for is $H(D,N)$.

To see the online Monge property, since

$$W_{n,j} - W_{n-1,j} = w_n(v_n - v_{j+1}),$$

we have $c_n = w_n v_n$, $\delta_j = -v_{j+1}$ and $\beta_n = w_n$, satisfying (8). So, Theorem 1 will solve the online problem in $O(D)$ amortized time per step. Hence, the total time to compute $H(D, N)$ is $O(DN)$.

[8] also gives an $O(DN)$ time algorithm, by observing the standard Monge property and applying the SMAWK algorithm. The algorithm in this paper has smaller constant factor in the $O(\cdot)$ notation, and hence is more efficient in practice. Further more, the online problem makes sense in this situation. It is known as the *one-sided* online problem. In this problem, a new user is added from right in each step. When a new user comes, our algorithm recomputes the optimal solution in $O(D)$ time amortized and $O(D \log N)$ time worst case.

We note that the corresponding online problem for solving the D-median on an *undirected* line was treated in [9], where a problem-specific solution was developed. The technique in this paper is a generalization of that one.

3.2 Wireless Mobile Paging

The second application comes from wireless networking [10]. In this problem, we are given N regions, called the *cells*, and there is a *user* somewhere. We want to find which cell contains the user. To do this, we can only query a cell whether the user is in or not, and the cell will answer yes or no. For each cell i, we know in advance the probability that it contains the user, denote it by p_i. We assume $p_1 \geq p_2 \geq \cdots \geq p_N$. We also approximate the real situation by assuming the cells are *disjoint*, so p_i is the probability that cell i contains the user *and* no other cell does.

There is a tradeoff issue between the delay and the bandwidth requirement. For example, consider the following two strategies. The first strategy queries all cells simultaneously, while the second strategy consists of N rounds, querying the cells one by one from p_1 to p_N, and stops as soon as the user is found. The fist strategy has the minimum delay, which is only one round, but has the maximum bandwidth requirement since it queries all N cells. The second strategy has the maximum worst case delay of N rounds, but the expected bandwidth requirement is the minimum possible, which is $\sum_{i=1}^{N} ip_i$ queries. In the tradeoff, we are given a parameter D, which is the worst case delay that can be tolerated, and we are going to find an optimal strategy that minimize the expected number of queries.

It is obvious that a cell with larger p_i should be queried no later than one with smaller p_i. So, the optimal strategy actually breaks the sequence $p_1, p_2, \ldots, p_N$ into D contiguous subsequences, and queries one subsequence in each round. Let $0 = r_0 < r_1 < \cdots < r_D = N$, and assume in round i, we query the cells from $p_{r_{i-1}+1}$ to p_{r_i}. Recall that the cells are disjoint. The expected number of queries, defined as the *cost*, is

$$\sum_{i=1}^{D} r_i \left(\sum_{l=r_{i-1}+1}^{r_i} p_l \right). \tag{9}$$

[10] developed a DP formulation to solve the problem. It is essentially the following DP. Let $H(d,n)$ be the optimal cost for querying cells $p_1,\ldots,p_n$ using exactly d rounds. Denote $W_{n,j} = n\sum_{l=j+1}^{n} p_l$ the contribution to (9) of one round that queries $p_{j+1},\ldots,p_n$. Then

$$H(d,n) = \begin{cases} \sum_{l=1}^{n} lp_l & n = d \\ \infty & d = 0,\ n \geq 1 \\ \min\limits_{d-1\leq j\leq n-1} (H(d-1,j) + W_{n,j}), & 1 \leq d < n \end{cases}$$

[10] applied the naive approach to solve the DP in $O(DN^2)$ time. Actually, this DP satisfies the online Monge property. Since

$$W_{n,j} - W_{n-1,j} = np_n + \sum_{l=j+1}^{n-1} p_l,$$

we can set $c_n = np_n + \sum_{l=1}^{n-1} p_l$, $\delta_j = -\sum_{l=1}^{j} p_l$ and $\beta_n = 1$, satisfying (8). So, the DP can be solved in $O(DN)$ time, using either the SMAWK algorithm or the technique in this paper. However, in this problem, there is no physical interpretation to the meaning of the online situation. But, due to the simplicity of our algorithm, it runs faster than the SMAWK algorithm in practice, as suggested by the experiments in [11], and is therefore more suitable for real time applications.

References

1. Burkard, R.E., Klinz, B., Rudolf, R.: Perspectives of Monge properties in optimization. Discrete Applied Mathematics **70**(2) (1996) 95–161
2. Aggarwal, A., Klawe, M.M., Moran, S., Shor, P.W., Wilber, R.E.: Geometric applications of a matrix-searching algorithm. Algorithmica **2** (1987) 195–208
3. Wilber, R.: The concave least-weight subsequence problem revisited. Journal of Algorithms **9**(3) (1988) 418–425
4. Eppstein, D., Galil, Z., Giancarlo, R.: Speeding up dynamic programming. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science. (1988) 488–496
5. Galil, Z., Giancarlo, R.: Speeding up dynamic programming with applications to molecular biology. Theoretical Computer Science **64**(1) (1989) 107–118
6. Galil, Z., Park, K.: A linear-time algorithm for concave one-dimensional dynamic programming. Information Processing Letters **33**(6) (1990) 309–311
7. Larmore, L.L., Schieber, B.: On-line dynamic programming with applications to the prediction of RNA secondary structure. Journal of Algorithms **12**(3) (1991) 490–515
8. Woeginger, G.J.: Monge strikes again: Optimal placement of web proxies in the Internet. Operations Research Letters **27**(3) (2000) 93–96
9. Fleischer, R., Golin, M.J., Zhang, Y.: Online maintenance of k-medians and k-covers on a line. Algorithmica **45**(4) (2006) 549–567

10. Krishnamachari, B., Gau, R.H., Wicker, S.B., Haas, Z.J.: Optimal sequential paging in cellular wireless networks. Wireless Networks **10**(2) (2004) 121–131
11. Bar-Noy, A., Feng, Y., Golin, M.J.: Efficiently paging mobile users under delay constraints. Unpublished manuscript (2006)

A Dropping the Extra Condition

This appendix will show how to drop the condition that

– the values of δ_j can be computed in $O(1)$ time, provided that the values of $h(i)$ for $1 \leq i < j$ are known.

In real applications, this doesn't seem to be an issue. For example, in both of the applications in Section 3, the value of δ_j can easily be computed in $O(1)$ time when needed, and in neither of the applications does δ_j depend on the previously-computed values of $h(i)$ for $1 \leq i < j$. It is a theoretical issue, though, so in this appendix, we will show how to dispense with the condition.

Recall (2) from Theorem 1. It is true that we cannot compute δ_n from other values available at step n, since the constraints containing δ_n will only appear from step $n+1$. However, it suffices to compute δ_n at step $n+1$, since we can modify the algorithm a little bit. The only place that uses δ_n in step n of the algorithm is in the addition of new line $L_n^n(x)$ to the lower envelope. After that, the algorithm computes $h(n)$ by evaluating the value of the lower envelope at $x = 0$, and then precedes to step $n+1$. So, we can postpone the addition of line $L_n^n(x)$ to the beginning of step $n+1$, after we compute δ_n. To compute $h(n)$ at step n, we can evaluate the value of the lower envelope *without* $L_n^n(x)$ at $x = 0$, compare it with $L_n^n(0) = a(n, n)$, and take the smaller of the two. Hence, what is left is to show

Lemma 5. *A feasible value of δ_n can be computed in $O(1)$ time at step $n+1$.*

Proof. We will show an algorithm that computes c_n and β_n at step n, and computes δ_n at step $n+1$. There are actually many feasible solutions of c_n, β_n and δ_j for (2). Consider a particular solution c_n, β_n and δ_j. If we set $c'_n = c_n + x\beta_n$, $\beta'_n = \beta_n$ and $\delta'_j = \delta_j - x$ for some arbitrary value x, then the new solution c'_n, β'_n and δ'_j still satisfies (2). This gives us the degree of freedom to choose δ_1. We choose $\delta_1 = 0$ and immediately get

$$c_n = a(n, 1) - a(n-1, 1), \qquad \forall\ 1 < n \leq N.$$

So, we can compute c_n at step n.

What is left is to compute β_n and δ_j. The constraints (2) become $\forall\ 1 < j < n \leq N$,

$$\delta_j \beta_n = a(n, j) - a(n-1, j) - c_n. \tag{10}$$

β_2 does not show up in the constraints (10). In fact, the value of β_2 will not affect the algorithm. So, we can choose an arbitrary value for it, e.g. $\beta_2 = 0$. All other values, β_n $(3 \leq n \leq N)$ and δ_j $(2 \leq j \leq N)$, appear in the constraints (10),

but we still have one degree of freedom. Consider a particular solution β_n and δ_j to the constraints (10). If we set $\beta'_n = \beta_n/x$, and $\delta'_j = \delta_j \cdot x$ for some $x > 0$, then we obtain another feasible solution. So, we can choose δ_2 to be an arbitrary negative value, e.g. $\delta_2 = -1$. The rest is easy. In step n, we can compute β_n by

$$\beta_n = [a(n,2) - a(n-1,2) - c_n]/\delta_2,$$

and in step $n+1$, we compute δ_n by

$$\delta_n = [a(n+1,n) - a(n,n) - c_{n+1}]/\beta_{n+1}.$$

Hence, the lemma follows.