Theory Comput Syst (2009) 45: 429–445
DOI 10.1007/s00224-009-9166-x

Online Dynamic Programming Speedups

Amotz Bar-Noy · Mordecai J. Golin · Yan Zhang

Published online: 21 January 2009

Abstract Consider the dynamic program $h(n) = \min_{1 \le j \le n} a(n, j)$, where $a(n, j)$ is some formula that may (online) or may not (offline) depend on the previously computed $h(i)$, for $i < n$. The goal is to compute all $h(n)$, for $1 \le n \le N$. It is well known that, if $a(n, j)$ satisfy the *Monge* property, then the SMAWK algorithm (Aggarwal et al., Algorithmica 2(1):195–208, 1987) can solve the offline problem in $O(N)$ time; a $\Theta(N)$ speedup over the naive algorithm.

In this paper we extend this speedup to the online case, that is, to compute $h(n)$ in the order $n = 1, 2, \ldots, N$ when (i) we do not know the values of $a(n', j)$ for $n' > n$ before $h(n)$ has been computed and (ii) do not know the problem size N in advance. We show that if $a(n, j)$ satisfy a stronger, but sometimes still natural, property than the Monge one, then each $h(n)$ can be computed in online fashion in $O(1)$ amortized time. This maintains the speedup online, in the sense that the total time to compute all $h(n)$ is $O(N)$. We also show how to compute each $h(n)$ in the worst case $O(\log N)$ time, while maintaining the amortized time bound.

For $a(n, j)$ satisfying our stronger property, our algorithm is also simpler than the standard SMAWK algorithm for solving the offline case. We illustrate our technique

The research of the first author was partially supported by the NSF program award CNS-0626606; the research of the second and third authors was partially supported by Hong Kong RGC CERG grant HKUST6312/04E.

A. Bar-Noy
Department of Computer and Information Science, Brooklyn College, 2900 Bedford Avenue Brooklyn, New York, NY 11210, USA
e-mail: amotz@sci.brooklyn.cuny.edu

M.J. Golin (✉) · Y. Zhang
Department of Computer Science, Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong
e-mail: golin@cse.ust.hk

Y. Zhang
e-mail: cszy@cse.ust.hk

on two examples from the literature; the first is the D-median problem on a line, and the second comes from mobile wireless paging.

Keywords Dynamic programming · Monge property

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 $a(n, j)$ is some formula that might depend upon the values of $h(i)$, for $1 \le i < n$. Our goal is to compute all $h(n)$ for $1 \le n \le N$.

In many applications, the values of $a(n, j)$ do depend[1] on the previously computed values $\{h(i) : 1 \le i < n\}$ so, (1) essentially represents a simple dynamic program (DP). A simple example would be DPs in the form

$$h(n) = \min_{1 \le j \le n} \big\{h(j-1) + w(n, j)\big\}, \quad \forall\, 1 \le n \le N, \tag{2}$$

i.e., $a(n, j) = h(j-1) + w(n, j)$ where $h(0)$ is given and $w(n, j)$ is some function that does *not* depend upon any of the $h(\cdot)$ values.

A naive calculation of the $h(n)$ as defined by (1) would require $\Theta(N^2)$ time. It is well known that if the $a(n, j)$ satisfy some special conditions, e.g., the *Monge* property, then this calculation can be reduced down to $\Theta(N)$ time. These speedups require that the problem be static and do not allow online computation of the $h(n)$. The main result of this paper is an algorithm that, in the presence of a stronger version of the Monge property, does permit maintaining the speedup in an online setting.

In the remainder of this section we define our terms and then state our results. In Sect. 2 we quickly review the Monge property and place our new stronger property in context. We also discuss a possible confusion that might occur due to the fact that the word "online" appeared previously in the Monge literature with a different meaning. Section 3 presents our new algorithm and Sect. 4 some modifications and generalizations. Section 5 presents two applications to problems in the literature. We conclude in Sect. 6 with an open question.

1.1 Definition of Online

For arbitrary functions $a(n, j)$, it would require $\Theta(N^2)$ time to compute all the $h(n)$. We can do better if the $a(n, j)$ possess special properties.

Definition 1 The values of $a(n, j)$ satisfy the *Monge property* [4], if for all $1 \le j < n < N$,

$$a(n, j) + a(n+1, j+1) \le a(n+1, j) + a(n, j+1). \tag{3}$$

[1] In this paper we follow the standard practice of assuming that any particular $a(n, j)$ can be computed in $O(1)$ time when needed, provided that the values of $h(i)$ upon which it depends are known.

From our perspective, the major result on such functions is the SMAWK[2] algorithm [1] which permits finding the $h(n)$ in linear time.

Theorem 2 (SMAWK [1]) *Consider the DP defined by* (1). *If* $a(n, j)$ *satisfy the Monge property, and*

1. *for any* n *and* j, *the value of* $a(n, j)$ *can be computed in* $O(1)$ *time, i.e.,* $a(n, j)$ *does not depend on any* $h(i)$;
2. *and the value of* N *is known in advance,*

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

The SMAWK algorithm therefore provides a $\Theta(N)$ speedup to the naive algorithm, in the offline case. Section 2.1 provides more background on the Monge property and the SMAWK algorithm.

The main purpose of this paper is to consider the DP problem in online settings. In applications, "online" means that some sort of "data" arrives one at a time, and, after each arrival, we recompute the optimal solution. Translating to mathematics, at each step, say step n, the values of $a(n, j)$ for $1 \leq j \leq n$ become available, and we need to compute $h(n)$. We will see some applications in Sect. 5.

An online algorithm would allow both of

(C1) the value of $a(n, j)$ may depend on any (or all) $h(i)$, for $1 \leq i < n$,
(C2) and the value of N is not known in advance.

(C1) violates condition 1 of Theorem 2 and (C2) violates condition 2; the SMAWK algorithm therefore can't work in the presence of either. We call (C1) and (C2) the *online conditions*.

The online condition (C2) is straightforward, but to understand (C1) we need to clarify the meaning of "depend". There can actually be two types of dependencies:

1. Explicit dependency. The formula $a(n, j)$ contains some $h(i)$ explicitly. For example, in (2), we have $a(n, j) = h(j - 1) + w(n, j)$.
2. Implicit dependency. The value of $a(n, j)$ depends on $h(i)$ because the problem is physically online, i.e., before computing $h(i)$ for $i < n$, the value of $a(n, j)$ is simply not available due to the problem setting.

Our algorithm will physically calculate all of the $h(i)$, for $i < n$, before calculating $h(n)$, so it does not need to distinguish between these two types.

In Sect. 2.3 we briefly mention the literature solving the case in which $a(n, j)$ may depend on some of the values in $\{h(i) : i < j\}$ (as opposed to $\{h(i) : i < n\}$), which we call the "semi-online" problem, and discuss how it differs from this one.

[2]SMAWK is an acronym of the first letters of the last names of the authors of [1].

1.2 The Results

Definition 3 The values of $a(n, j)$ satisfy the *online Monge property*, if for all $1 \leq j < n \leq N$,

$$a(n, j) - a(n-1, j) = c_n + \delta_j \beta_n, \tag{4}$$

where c_n, β_n and δ_j are constants satisfying

1. for all $2 \leq n \leq N$, $\beta_n \geq 0$,
2. and $\delta_1 \geq \delta_2 \geq \cdots \geq \delta_{N-1}$.

The main result of this paper is

Theorem 4 *Consider the DP defined by* (1). *If $a(n, j)$ satisfy the online Monge property, and*

1. *for any n and j, 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,*

then there is an algorithm (*Sects.* 3 *and* 4) *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 for each $h(n)$.*

Note that, from the statement of the theorem, the algorithm does not need to know the value of N in advance.

It is easy to see that the online Monge property implies that

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

i.e., it implies that $a(n, j)$ satisfy the standard Monge property. However, the online Monge property seems quite artificial. In Sect. 2.2, we will see that it actually has a very natural interpretation in that it is equivalent to a Monge property with rank one matrices in the standard decomposition.

As mentioned before, the SMAWK algorithm provides a $\Theta(N)$ speedup in the calculation of the $h(n)$ when $a(n, j)$ satisfy the Monge property and the problem is offline. Theorem 4 says that if $a(n, j)$ satisfy a stronger version of the Monge property, then this same speedup can be maintained online, in the sense that the time to compute all $h(n)$ is still $O(N)$.

Note that the online Monge property only requires that c_n, β_n and δ_j *exist*. It does not require that c_n, β_n and δ_j be given or computable in $O(1)$ time. But, if δ_j is given, the algorithm will be much easier to develop and understand. So, in what follows we will start by assuming we have an extra condition:

(C3) For any j, the value 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 and in Sect. 4.3, we will show how to remove it.

We point out that Auletta et al. [2] and Fleisher et al. [6] model the problem of placing K medians on an undirected line with N nodes by a DP that looks as if it

requires $\Theta(KN^2)$ time to solve. Fleischer et al. [6] also noted that this problem has a Monge property that permits reducing the running time to $\Theta(KN)$ in the offline case. Both references then provide special purpose algorithms that show how to solve the problem online, i.e., adding the nodes one at a time to the right of the line and recomputing the medians after each addition, without losing the DP speedup. After deconstruction, the algorithms provided there can be seen as a very special case of the general algorithm given in this paper.

2 More Background

2.1 The Monge Property and the SMAWK Algorithm

We start with a brief introduction to the Monge property and the SMAWK algorithm. The survey [4] provides many more details. Consider an $N \times N$ matrix M. Denote by $R(n)$ the *index* of the rightmost minimum of row n of M, i.e.,

$$R(n) = \max\left\{ j : M_{n,j} = \min_{1 \le i \le N} M_{n,i} \right\}.$$

A matrix M is *monotone* if $R(1) \le R(2) \le \cdots \le R(N)$, M is *totally monotone* if all submatrices[3] of M are monotone. The SMAWK algorithm says that if M is totally monotone, then the set of all of the $R(n)$ for $1 \le n \le N$ can be computed in $O(N)$ time.

For our problem, if we set

$$M_{n,j} = \begin{cases} a(n,j), & 1 \le j \le n \le N; \\ \infty, & \text{otherwise}, \end{cases} \tag{5}$$

then $h(n) = a(n, R(n))$. Hence, if we can show that the matrix M defined by (5) is totally monotone, then the SMAWK algorithm can solve our problem (offline version) in $O(N)$ time.

Total monotonicity is quite difficult to demonstrate directly. In practice, it is usually established by demonstrating that the matrix possesses the Monge property which is essentially what we introduced in Definition 1 if we consider the matrix as a two-variable function.

Definition 5 An $N \times N$ matrix M is Monge, if for all $1 \le n < N$ and $1 \le j < N$,

$$M_{n,j} + M_{n+1,j+1} \le M_{n+1,j} + M_{n,j+1}. \tag{6}$$

Note that if the $a(n, j)$ are Monge as in Definition 1, then the associated M given in (5) is a Monge matrix as in Definition 5.

It is easy to show [4] that if M is Monge, then it is totally monotone. So, if $a(n, j)$ are Monge, then the SMAWK algorithm can calculate all of the $h(n)$, in the offline case, in $O(N)$ time.

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

2.2 Decompositions and the Online Monge Property

As already shown, the online Monge property is a special case of the Monge property. We now make this more formal. Monge matrices can be decomposed (Sect. 2.2 of [4]) as follows.

Lemma 6 *An $N \times N$ matrix M is Monge if and only if for all $M_{n,j} \neq \infty$,*

$$M_{n,j} = P_n + Q_j + \sum_{k=n}^{N} \sum_{i=1}^{j} F_{ki} \tag{7}$$

where P and Q are vectors of length N, and F is an $N \times N$ matrix, whose entries are all nonnegative.

The matrix F is called the *density matrix*. We now show that the matrices satisfying the online Monge property are exactly those that have rank-one density matrices. Recall the definition of online Monge property. Let $\delta_0 = \delta_1$, then

$$\begin{aligned} a(n,j) &= a(n+1,j) - c_{n+1} - \delta_j \beta_{n+1} \\ &= a(n+2,j) - (c_{n+2} + c_{n+1}) - \delta_j(\beta_{n+2} + \beta_{n+1}) \\ &= \cdots \\ &= a(N,j) - \sum_{k=n+1}^{N} c_k - \delta_j \sum_{k=n+1}^{N} \beta_k \\ &= a(N,j) - \sum_{k=n+1}^{N} c_k - \delta_0 \sum_{k=n+1}^{N} \beta_k + (\delta_0 - \delta_j) \sum_{k=n+1}^{N} \beta_k. \end{aligned}$$

So, for the online Monge property,

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

where we set $\beta_{N+1} = 0$. So, the online Monge property is a special case of the Monge property where the density matrix F is of rank 1.

Conversely, if $\operatorname{rank}(F) = 1$, then $F_{ki} = U_k V_i$ where U, V are nonnegative vectors of length N. From (7),

$$a(n,j) - a(n-1,j) = P_n - P_{n-1} - U_{n-1} \sum_{i=1}^{j} V_i.$$

That is, the values of $a(n,j)$ satisfy the online Monge property with

$$c_n = P_n - P_{n-1}, \qquad \beta_n = U_{n-1}, \qquad \delta_j = -\sum_{i=1}^{j} V_i.$$

So, what we are calling the online Monge property is exactly the Monge property with rank one density matrices.

2.3 Semi-Online Problems

There is a series of papers discussing another type of "online" version of (1), e.g., [5, 7–9, 11, 12]. The final 'best' result in this line is given by

Theorem 7 (LARSCH [11]) *Consider the DP defined by* (1). *If* $a(n, j)$ *satisfy the Monge property, and*

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

then the LARSCH algorithm can compute all of the $h(n)$, *for* $1 \le n \le N$, *in* $O(N)$ *time*.

We call this type of problem "semi-online" because the $a(n, j)$ are only allowed to depend on $\{h(i) : i < j\}$, and not on all of the $\{h(i) : i < n\}$ as in the online condition (C1). Also, the LARSCH algorithm does not support the online condition (C2) since it requires the problem size N to be fixed in advance.

3 The Main Algorithm

In this section, we develop the online algorithm that achieves the $O(1)$ amortized bound in Theorem 4. For the purposes of this section, we assume the slightly simpler condition $\delta_1 > \delta_2 > \cdots > \delta_{N-1}$ for simplicity. We postpone the extension to the case where, for some i, $\delta_i = \delta_{i+1}$, the analysis of the worst-case bound, and other details to Sect. 4.

We will show the algorithm at step n, where the values of $\{h(i) : 1 \le i < n\}$ have been computed, and we want to compute $h(n)$. By the conditions in Theorem 4 and the extra condition (C3), all the values $a(n, j)$ and δ_j for $1 \le j \le n \le N$ are known.

3.1 The Lower Envelope

The key to the algorithm is the following set of straight lines:

Definition 8 For all $1 \le j \le n \le N$, we define

$$L_j^n(x) = a(n, j) + \delta_j \cdot x. \tag{8}$$

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 $x \in [0, \infty)$.

3.1.1 The Data Structure

The only data structure used is an array, called the *active-indices array*, $Z = (z_1, \ldots, z_t)$ for some $t \le n$. 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 $L^n_{z_1}(x), L^n_{z_2}(x), \ldots, L^n_{z_t}(x)$. The slopes of the line segments forming the lower envelope of a set of lines decreases as one sweeps from left to right. Since $\delta_1 > \delta_2 > \cdots > \delta_n$, we have $z_1 < z_2 < \cdots < z_t = n$, and no line can appear more than once in the active-indices array.

Given the active-indices array, computing $h(n)$ is a constant time operation since

$$h(n) = L^n_{z_1}(0) = a(n, z_1).$$

So, the problem is how to obtain and maintain the active-indices array. Inductively, at the time that the algorithm enters step n from step $n-1$, it maintains the active-indices array for the step $n-1$, which represents the lower envelope of the lines $\{L^{n-1}_j(x) : 1 \le j \le n-1\}$. So, the main part of the algorithm is to *update* the old active-indices array to become the new active-indices array for $\{L^n_j(x) : 1 \le j \le n\}$.

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 $L^n_{z_i}(x)$ and $L^n_{z_{i+1}}(x)$. 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. That is, x_i is the unique solution to the equation

$$a(n, z_i) + \delta_{z_i} \cdot x = L^n_{z_i}(x) = L^n_{z_{i+1}}(x) = a(n, z_{i+1}) + \delta_{z_{i+1}} \cdot x,$$

or

$$x = -\frac{a(n, z_{i+1}) - a(n, z_i)}{\delta_{z_{i+1}} - \delta_{z_i}}. \tag{9}$$

3.2 Updating the Lower Envelope

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

3.2.1 Updating the first $n-1$ lines

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

Lemma 9 *For all $1 < n \le N$ and for all x,*

$$L^n_j(x) = L^{n-1}_j(x + \beta_n) + c_n, \quad \forall\, 1 \le j \le n-1.$$

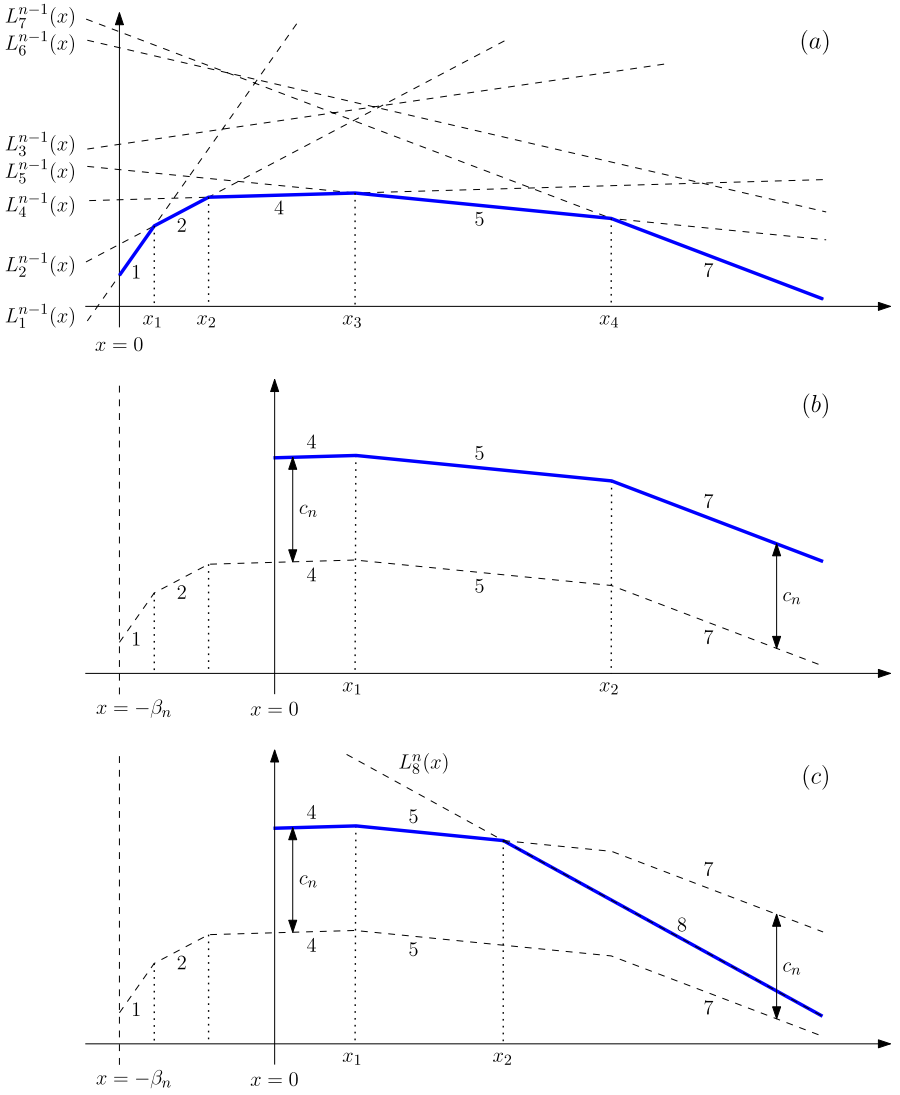

Fig. 1 The update of the active-indices array from Step $n-1$ to Step n, where $n = 8$. The *thick solid chains* are the lower envelopes. (**a**) Shows the lower envelope for the lines $\{L_j^{n-1}(x) : 1 \le j \le n-1\}$, (**b**) shows the lower envelope for the lines $\{L_j^n(x) : 1 \le j \le n-1\}$, and (**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**) $(4, 5, 8)$

Proof By (4) and (8),

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

□

Lemma 9 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 $1 \le j \le n-1$.

Corollary 10 *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 Fig. 1(a) and 1(b). From Fig. 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 $L_{z_i}^n(x)$ that appears on the lower envelope is completely contained in the range $x \in (-\infty, 0)$. By Corollary 10, to obtain the active-indices array for $\{L_j^n(x) : 1 \le j \le n-1\}$ from the old active-indices array, we only need to delete those active-indices that become 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, and check each active-index to see whether it becomes negative. If it does, 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 $\{L_j^n(x) : 1 \le j \le n-1\}$ is $(z_{\min}, \ldots, z_t)$. See Part 2.1 of Fig. 2.

3.2.2 Adding the Last Line

We now add the line $L_n^n(x)$. Recall that we assume $\delta_1 > \delta_2 > \cdots > \delta_{N-1}$. Since $L_n^n(x)$ has a smaller slope than any line currently in the lower envelope, it must be the rightmost segment on the lower envelope. There are now two mutually exclusive possibilities; either (i) $L_n^n(x)$ becomes the *only* line on the new lower envelope or (ii) $L_n^n(x)$ will not become the only line on the new lower envelope.

Possibility (i) occurs if and only if $L_n^n(0) \le L_{z_{\min}}^n(0)$, which can be checked in constant time.

If (i) does not occur, then (ii) does. In this case, 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 $\{L_j^n(x) : 1 \le j \le n-1\}$. Assume they intersect on segment $z_{\max}$; then the new lower envelope should be $(z_{\min}, \ldots, z_{\max}, n)$. See Fig. 1(c); in the example, $z_{\max} = 5$.

To find $z_{\max}$, we also use a sequential scan, but now from right to left. We scan the active-indices array starting from z_t, stepping down 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 $L_{z_i}^n(x_{i-1})$. If $L_n^n(x_{i-1})$ is smaller, then z_i is deleted from the active-indices array. Otherwise, we stop and let $z_{\max}$ be z_i. See Part 2.2 of Fig. 2.

3.3 Running Time

The sequential scans use $O(1)$ time for each insertion and deletion in the active-indices array. Since each line can be inserted or deleted at most once, the algorithm uses $O(1)$ amortized time per step.

The Amortized $O(1)$ Time Algorithm

1. // Initialize.
   Set $Z = (z_1) = (1)$;
   Output $h(1) = a(1, z_1)$;

2. **While** (a new step begins) // step n
   {
   2.1. // Deal with $\{L_j^n(x) : 1 \le j \le n-1\}$.
   // This starts with $Z = (z_1, \ldots, z_t)$.
   For i from 1 to t
   {
   Compute x_i;
   If $(x_i \le 0)$
   Remove z_i from array Z;
   Else
   End the "For" loop immediately;
   }
   2.2. // Adding the line $L_n^n(x)$.
   // This starts with $Z = (z_{\min}, \ldots, z_t)$.
   If $L_n^n(0) \le L_{z_{\min}}^n(0)$ // L_n^n is only line on envelope
   $Z = (n)$; $z_{\min} = n$;
   Else
   For i from t down to "min"
   {
   Compute x_{i-1};
   If $\left(L_{z_i}^n(x_{i-1}) \ge L_n^n(x_{i-1})\right)$
   Remove z_i from array Z;
   Else
   End the "For" loop immediately; // $z_{\max} = z_i$.
   }
   $Z \leftarrow (z_{\min}, \ldots, z_{\max}, n)$
   2.3. Output $h(n) = a(n, z_{\min})$;
   }

Fig. 2 The pseudo-code for the amortized $O(1)$ time algorithm. Part 2.1 of this figure corresponds to Sect. 3.2.1. Part 2.2 corresponds to Sect. 3.2.2

4 Extensions to the Analysis and the Algorithm

4.1 $\delta_1 \ge \delta_2 \ge \cdots \ge \delta_{N-1}$

In Sect. 3, we assumed $\delta_1 > \delta_2 > \cdots > \delta_{N-1}$, which, for example, is what occurs in the applications in Sect. 5. But in general, the values of δ_j may not be distinct.

The only place that is affected is the adding of the last line $L_n^n(x)$. This line may no longer be the rightmost segment of the lower envelope when the values of δ_j can be the same. To be precise, consider the case $\delta_t = \delta_n$, where t is the index of the rightmost segment before $L_n^n(x)$ is considered. Then, according to (8), $L_n^n(x)$ is the rightmost segment if and only if $a(n,t) \geq a(n,n)$. So it is only necessary to modify the algorithm as follows: (i) If $a(n,t) \geq a(n,n)$ proceed as in Sect. 3.2.2. (ii) If $a(n,t) < a(n,n)$, leave the lower envelope unchanged (throwing away $L_n^n(x)$).

4.2 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 any real number x, the y-coordinate of $L_z^n(x)$ at x can be evaluated in $O(1)$ time, the binary search takes $O(\log N)$ time in the worst case.

To keep *both* the $O(1)$ amortized and the $O(\log N)$ worst-case time bounds per step, we can run both the sequential search and the binary search in parallel, interleaving their steps, stopping when the first one of the two searches completes. Then the total search time is at most 2 times the minimum of the two and we maintain both time bounds.

4.3 Dropping the Extra Condition (C3)

In this section, we will show how to drop the condition

(C3) For any j, the value of δ_j can be computed in $O(1)$ time, provided that the values of $h(i)$ for $1 \leq i < j$ are known.

The algorithm uses the δ_j to define the lines $L_j^n(x)$ using (8) and to calculate the break-points using (9). Most of the calculations in step n only require using δ_j for $j < n$ and we can assume, inductively, that these have already been previously calculated and stored for our use. The only place that uses δ_n, i.e., the line $L_n^n(x)$, in step n is the right-to-left scan when inserting line $L_n^n(x)$ into the lower envelope.

The difficulty is that, if we are not somehow explicitly given the value δ_n, we cannot compute δ_n from other values available at step n, since the constraints containing δ_n will only first appear from step $n+1$. So, we will not know the line $L_n^n(x)$ during the end of step n when we need it.

The idea is to postpone the computation of δ_n and the addition of $L_n^n(x)$ to the lower envelope, until the beginning of step $n+1$. To compute $h(n)$ at step n, we can evaluate the lower envelope — now without $L_n^n(x)$ — at $x = 0$, compare this value with $L_n^n(0) = a(n,n)$, and return the smaller of the two.

What is left is to show

Lemma 11 *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 (4). 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 (4). This gives us the degree of freedom to choose δ_1. We choose $\delta_1 = 0$ and immediately get

$$c_n = a(n,1) - a(n-1,1), \quad \forall 1 < n \le N. \tag{10}$$

So, we can compute c_n at step n.

To compute β_n and δ_j, we substitute (10) into (4):

$$\delta_j \beta_n = a(n,j) - a(n-1,j) - c_n, \quad \forall 1 < j < n \le N. \tag{11}$$

β_2 does not show up in (11). In fact, the value of β_2 cannot affect the algorithm. So, we can choose an arbitrary value for it, e.g., $\beta_2 = 0$. All other values, β_n $(3 \le n \le N)$ and δ_j $(2 \le j < N)$, appear in (11), but we still have one degree of freedom:

Consider a particular set of solutions β_n and δ_j to (11). If we set $\beta'_n = \beta_n / x$, and $\delta'_j = \delta_j \cdot x$ for some $x > 0$, then it is still a feasible solution. So, we can choose δ_2 to be an arbitrary negative value, say $\delta_2 = -1$. 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

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

The lemma follows. □

4.4 Removing Conditions on δ_j and β_j

After developing the algorithm it is interesting to go back and ask whether all of the conditions we imposed are necessary. In particular, consider the case that the values of $a(n,j)$ only satisfy (4), i.e., β_n and δ_j can be arbitrary real values without the nonnegative or the nonincreasing constraints. Note that in this case the values of $a(n,j)$ are no longer Monge.

In this case, we keep the entire lower envelope for the range $x \in (-\infty, \infty)$. The sequential search will fail, but the binary search still works. So, we get a worst case $O(\log N)$ time algorithm.

Corollary 12 *Consider the DP defined by (1). If for all $1 \le j < n \le N$, there exists c_n, β_n and δ_j such that (4) is satisfied and*

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

then there is an algorithm that computes the values of $h(n)$ in the order $n = 1, 2, \ldots, N$ in $O(\log N)$ worst-case time for each $h(n)$.

5 Applications

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

$$H(d,n) = \min_{d-1 \le j \le n-1} \left(H(d-1,j) + W^{(d)}_{n,j} \right), \tag{12}$$

where the values of $H(d,n)$ for $d = 0$ or $n = d$ are given, the values of $W^{(d)}_{n,j}$ can be computed in $O(1)$ time and, for each fixed d $(1 \le d \le D)$, the $W^{(d)}_{n,j}$ satisfy the online Monge property in Theorem 4, i.e.,

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

where $\delta^{(d)}_j$ decreases as j increases, and $\beta^{(d)}_n \ge 0$. The goal is to compute $H(D,N)$. Setting

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

it is easy to see that for each fixed d $(1 \le d \le D)$, the values of $a^{(d)}(n,j)$ satisfy the online Monge property as well since

$$a^{(d)}(n,j) - a^{(d)}(n-1,j) = W^{(d)}_{n,j} - W^{(d)}_{n-1,j} = c^{(d)}_n + \delta^{(d)}_j \beta^{(d)}_n. \tag{13}$$

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 4, for each d, the value of $H(d,n)$ can be computed in $O(1)$ amortized time. So, the set of values $\mathcal{H}_n$ can be computed in $O(D)$ amortized time. This gives a total of $O(DN)$ time to compute $H(D,N)$, while the naive algorithm requires $O(DN^2)$ time.

5.1 D-Medians on a Directed Line

The first application comes from [13]. 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 on the line. Each user v_i has a *weight*, denoted by w_i, representing the amount of service it 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.

The cost of a server at point v_l servicing w units of request by point v_i is $w_i(v_i - v_l)$; to minimize this a user will always be serviced by the nearest server to its left. 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 minimum cost of servicing v_i will then be $w_i \ell(v_i, S)$.

The objective is to choose D servers (not counting v_1) to minimize the total service *cost*, that is

$$\min_{|S|=D+1} \sum_{i=1}^{N} w_i \ell(v_i, S).$$

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

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

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

Note that

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

which gives the online Monge property with $c_n = w_n v_n$, $\delta_j = -v_{j+1}$ and $\beta_n = w_n$, satisfying (13). So, Theorem 4 will solve the online problem in $O(D)$ amortized time per step, and the total time to compute $H(D,N)$ is $O(DN)$.

Woeginger [13] gives an offline $O(DN)$ time algorithm for this problem, by observing that the standard Monge property holds and applying the SMAWK algorithm (D times).

The online problem has a physical interpretation here. Consider the users arriving one by one, each new user arriving to the *right* of the previous users (we therefore call this the *one-sided* online problem). After a new user arrives we want to calculate the new set of D medians that minimizes the cost. Our algorithm permits doing this in $O(D)$ amortized and $O(D \log N)$ worst case time.

We note that the corresponding online problem for solving the D-median on an *undirected* line was treated in [6], where a problem-specific solution was developed. As previously mentioned, the technique in this paper can be regarded as a generalization of that algorithm.

5.2 Wireless Mobile Paging

The second application comes from wireless networking [10]. In this problem, we are given N regions, called *cells*, and a *user* located in one of them. We want to find which cell contains the user. To do this, we can only query a cell whether the user is located in it; the cell will answer yes or no. For each cell i, we know in advance the probability, denoted by p_i, that it contains the user. Without loss of generality, 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 exactly one cell contains the user.

Such a search exhibits a tradeoff between delay and (expected) 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, stopping as soon as the user is found. The first 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 can be calculated to be $\sum_{i=1}^{N} ip_i$ queries, which can be shown to be the minimum possible.

In the tradeoff, we are given a parameter D, which is the worst case delay that can be tolerated, and our goal is to find an optimal strategy that minimizes 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

$$\sum_{i=1}^{D} r_i \left(\sum_{l=r_{i-1}+1}^{r_i} p_l \right). \tag{14}$$

Krishnamachari et al. [10] used this equation to develop a DP formulation to solve this 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. Let $W_{n,j} = n \sum_{l=j+1}^{n} p_l$ denote the contribution to (14) of one round that queries $p_{j+1}, \ldots, p_n$. Then

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

Krishnamachari et al. [10] applied the naive approach to solve the DP in $O(DN^2)$ time. But, 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 (13). This DP therefore satisfies the online Monge property and can thus be solved in $O(DN)$ time, using either the SMAWK algorithm or the online technique in this paper.

In this problem, we know of no physical interpretation to the online problem. However due to the simplicity of our algorithm (performing simple scans), it seems to run faster than the SMAWK algorithm in practice, as suggested by the experiments in [3], and therefore might be more suitable for real time applications.

6 Open Problems

It is well known that the solution of Dynamic Programs can be sped up if they possess a *Monge* property. This speedup is inherently restricted to offline problems. In this

paper, we showed (Theorem 4) how, if the problem possesses what we call an *online Monge* property, we can maintain the speedup in an online setting.

In Sect. 2.2 we showed that our online Monge property was equivalent to a Monge property with rank 1 density matrices. This raises the question of how tight our results are. Is it possible to show that there are online algorithms that maintain the speedup for all Monge properties with rank $\leq k$ density matrices for some $k > 1$? Or, might it be possible to show that no such general algorithm exists for $k > 1$?

References

1. Aggarwal, A., Klawe, M.M., Moran, S., Shor, P.W., Wilber, R.E.: Geometric applications of a matrix-searching algorithm. Algorithmica **2**(1), 195–208 (1987). A preliminary version appeared in Proceedings of the 2nd Annual Symposium on Computational Geometry, pp. 285–292 (1986)
2. Auletta, V., Parente, D., Persiano, G.: Placing resources on a growing line. J. Algorithms **26**(1), 87–100 (1998)
3. Bar-Noy, A., Feng, Y., Golin, M.J.: Paging mobile users efficiently and optimally. In: Proceedings of the 26th Annual IEEE Conference on Computer Communications (Infocom'07), pp. 1910–1918 (2007)
4. Burkard, R.E., Klinz, B., Rudolf, R.: Perspectives of Monge properties in optimization. Discrete Appl. Math. **70**(2), 95–161 (1996)
5. Eppstein, D., Galil, Z., Giancarlo, R.: Speeding up dynamic programming. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science, pp. 488–496 (1988)
6. Fleischer, R., Golin, M.J., Zhang, Y.: Online maintenance of k-medians and k-covers on a line. Algorithmica **45**(4), 549–567 (2006). A preliminary version appeared in Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, pp. 102—113 (2004)
7. Galil, Z., Giancarlo, R.: Speeding up dynamic programming with applications to molecular biology. Theor. Comput. Sci. **64**(1), 107–118 (1989)
8. Galil, Z., Park, K.: A linear-time algorithm for concave one-dimensional dynamic programming. Inf. Process. Lett. **33**(6), 309–311 (1990)
9. Klawe, M.M.: A simple linear time algorithm for concave one-dimensional dynamic programming. Technical Report 89-16, Department of Computer Science, University of British Columbia (1989)
10. Krishnamachari, B., Gau, R.-H., Wicker, S.B., Haas, Z.J.: Optimal sequential paging in cellular wireless networks. Wirel. Netw. **10**(2), 121–131 (2004)
11. Larmore, L.L., Schieber, B.: On-line dynamic programming with applications to the prediction of RNA secondary structure. J. Algorithms **12**(3), 490–515 (1991). A preliminary version appeared in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 503—512 (1990)
12. Wilber, R.: The concave least-weight subsequence problem revisited. J. Algorithms **9**(3), 418–425 (1988)
13. Woeginger, G.J.: Monge strikes again: Optimal placement of web proxies in the Internet. Oper. Res. Lett. **27**(3), 93–96 (2000)