Advanced Techniques for Dynamic Programming

Wolfgang Bein

Contents

W. Bein
Department of Computer Science, University of Nevada, Las Vegas, Las Vegas, NV, USA
e-mail: beinw@unlv.nevada.edu

P.M. Pardalos et al. (eds.), *Handbook of Combinatorial Optimization*,
DOI 10.1007/978-1-4419-7997-1_28, © Springer Science+Business Media New York 2013

Abstract

This is an overview over dynamic programming with an emphasis on advanced methods. Problems discussed include path problems, construction of search trees, scheduling problems, applications of dynamic programming for sorting problems, server problems, as well as others. This chapter contains an extensive discussion of dynamic programming speedup. There exist several general techniques in the literature for speeding up naive implementations of dynamic programming. Two of the best known are the Knuth-Yao quadrangle inequality speedup and the SMAWK/LARSCH algorithm for finding the row minima of totally monotone matrices. The chapter includes "ready to implement" descriptions of the SMAWK and LARSCH algorithms. Another focus is on dynamic programming, online algorithms, and work functions.

1 Introduction

Dynamic programming, formally introduced by Richard Bellman (August 26, 1920–March 19, 1984) at the Rand Corporation in Santa Monica in the 1940s, is a versatile method to construct efficient algorithms for a broad range of problems. As with many tools which evolved over time, the original paper sounds antiquated. In his monograph "Dynamic Programming," Bellman [25] describes the principle of optimality, which is central to dynamic programming: "An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." Not exactly what is found in textbooks today. But then in those days the focus was more on stochastic processes and not so much on combinatorial optimization. The term "programming" is outdated as well, as it does not refer to programming in a modern programming language. Instead, programming means planning by filling in tables. The term "linear programming" derives in the same way. Over the decades, dynamic programming has evolved and is now one of the "killer" techniques in algorithmic design. And, of course, the laptop computer on which this chapter was written is more powerful than anyone in Bellman's days could have imagined.

Today, the emphasis is on how to organize dynamic programming in a way that makes it possible to solve massive problems (which – again – would have never been considered in Bellman's days) in reasonable time. The focus is on dynamic programming speedup. Such speedup comes from carefully observing what values are essential and need to be computed and which might be unnecessary. Keeping

proper look-up tables on the side can accomplish this sometimes. But there are many situations where there are inherent monotonicity properties which can be exploited to only calculate a fraction of what is necessary in a simple-minded approach.

The goal of this chapter is to briefly introduce dynamic programming, show some of the diversity of problems that can be tackled efficiently with dynamic programming, and – centrally – focus on the issue of dynamic programming speedup. Clearly, the chapter does not cover all of dynamic programming, but there is a section on recommended reading, Sect. 8, which covers some ground.

The chapter is organized as follows: Sect. 2 starts with a simple introduction to dynamic programming using a few standard examples, described more tersely than in a textbook and augmented with a general few themes. A reader altogether unfamiliar with dynamic programming might utilize some of the resources given in Sect. 8.

Section 3 contains "open-ended dynamic programming," where a process is updated continuously. This is closely related to the theory of online optimization. In online computation, an algorithm must make decisions without knowledge of future inputs. Online problems occur naturally across the entire field of discrete optimization, with applications in areas such as robotics, resource allocation in operating systems, and network routing. Dynamic programming plays an important role in this kind of optimization. Notably, work functions replace tables or matrices used in offline optimization.

Section 4 contains an example from sorting. The intent of this section is twofold: First, lesser-known dynamic programming techniques for sorting are highlighted. Second, here is an example where a straightforward simple-minded implementation might give an algorithm of cubic complexity or worse, and a more intricate setup solves the problem in quadratic time, thus achieving substantial dynamic programming speedup.

Section 5 gives an example from scheduling (a batching problem) to illustrate important properties for dynamic programming speedup: the Monge property and total monotonicity. Techniques exploiting these techniques are now standard, and the reader might consult Sect. 8 to see how prolific such techniques are.

Speedup is based on two important and intricate algorithms: SMAWK and LARSCH. Use of these is essential for many fast dynamic programming algorithms. However, these algorithms are currently only accessible through the original publications. Section 6 contains "ready to implement" descriptions of the SMAWK and LARSCH algorithms.

Another type of speedup is based in the Knuth-Yao quadrangle inequality; this dynamic programming speedup works for a large class of problems. Even though both the SMAWK algorithm and the Knuth-Yao (KY) speedup use an implicit quadrangle inequality in their associated matrices, on second glance, they seem quite different from each other. Section 7 discusses the relation between these kinds of speedup in greater detail.

Finally, Sect. 8 is a cross-reference list with other chapters, and Sect. 8 gives concluding remarks.

2 A Few Standard Introductory Examples

As mentioned earlier, this section has a few introductory examples.

2.1 A Brief Introduction to Dynamic Programming: Fibonacci, Pascal, and Making Change

Many problems can be solved recursively by dividing an instance into subinstances. But a direct implementation of a recursion is often inefficient because subproblems overlap and are recomputed numerous times. The calculation of the Fibonacci numbers provides a simple example:

$$f_n = \begin{cases} f_{n-1} + f_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & n = 0. \end{cases} \tag{1}$$

Recursively, the Fibonacci numbers can be calculated in the following way:

function $fib(n)$
if $n = 0$ or $n = 1$ **return** n
else return $fib(n-1) + fib(n-2)$.

This is extremely inefficient; many values will be recalculated; see Fig. 1. Instead, one could use an array and would simply fill in values from "left to right," starting with F_0 and F_1 and the using Eq. 1 to continue. This way every value is only calculated once.

The Pascal triangle for calculating binomial coefficients provides a good example for the use of *tables* in dynamic programming. Recall the definition of the binomial coefficient:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \\ 0 & \text{else.} \end{cases} \tag{2}$$

Clearly, the coefficients can be calculated using the following recursive program:

function $c(n,k)$
if $k = 0$ or $k = n$ **return** 1
else return $c(n-1,k-1) + c(n-1,k)$.

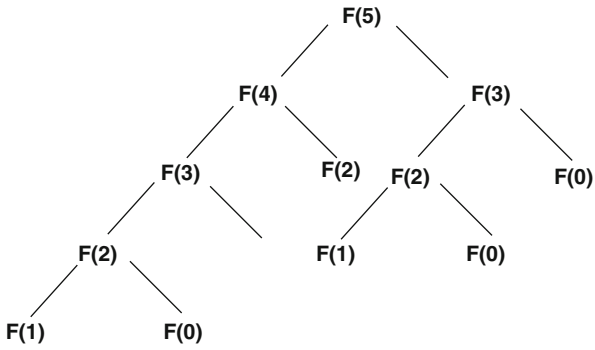


Fig. 1 Recursive calculation of the fifth Fibonacci number

But again this is inefficient, as terms are recalculated over and over. Indeed, since the solution is ultimately made up of terms of value 1. the run time of this algorithm is $\Omega(\binom{n}{k})$. This problem consists of *overlapping subproblems*, which often makes it inefficient to use recursions directly. Instead, one can calculate the coefficients *bottom up* using a table to store intermediate results. In the case of the binomial coefficient, this is the Pascal triangle Fig. 2. With it the run time is $\Theta(nk)$ with space requirement $\Theta(k)$ (as only two rows at a time need to be stored).

Most often dynamic programming is used for optimization problems. As an example consider the problem of making change with the minimum number of coins. Given are n denominations of value $\{d_1, \ldots d_n\}$, an unlimited supply of these coins, and the problem is to make change for amount A. For example, if the denominations are $1, 5, 10, 12$, and $A = 21$, then the minimum number of coins is 3. Note that the greedy algorithm uses 6 coins. What is important here is that the principle of optimality holds: If optimal change for amount C involves making change into amounts A and B, $C = A + B$, then change for A and B is also optimal. More generally, the principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal.

Let

$$C[i, j] = \text{minimum number of coins required to make change for amount } j \text{ using only coins of denomination } \{1, \ldots, i\},$$

where $i = 1, \ldots n$ and $j = 1, \ldots A$. The principle of optimality implies the following recurrence:

$$C[i, j] = \min\{C[i-1, j], 1 + C[i, j - d_i]\}, \tag{3}$$

where out-of-bounds values are set to ∞.

Just as before with Fibonacci and Pascal, one uses a table to calculate values "bottom-up." The table is initialized by setting all $C[i, 0]$ to 0. Then the table is filled row by row using Eq. 3. An example is in Table 1.

Fig. 2 The Pascal triangle

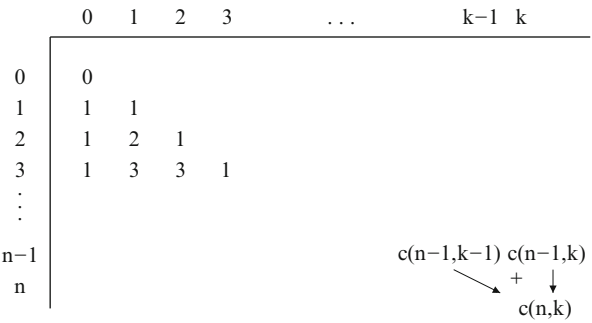


Table 1 An instance of the change-making problem: There are four types of coins with denominations $d_1 = 1, d_2 = 5, d_3 = 10$, and $d_4 = 12$. The amount is $A = 21$. The last entry in the table gives the minimum number of coins, which is 3. Note that the greedy algorithm uses six coins

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
d_1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
d_2	0	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6	7	4	5
d_3	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6	2	3
d_4	0	1	2	3	4	1	2	3	4	5	1	2	1	2	3	2	3	2	3	4	2	3

2.2 Chained Matrix Multiplication Problem

Given is a sequence of matrices $M_1, \ldots, M_n$ to be multiplied. The dimensions are $d_0 \times d_1, \ldots d_{n-1} \times d_n$, denoted by $(d_0, d_1, \ldots, d_n)$, for short. The question is what parenthesization gives the minimum number of multiplications. For example, given three matrices, A, B, C, with dimensions $(12, 5, 90, 2)$, the calculation $A(BC)$ requires $5 \times 90 \times 2$ multiplications to produce BC and then $12 \times 5 \times 2$ to multiply the result by A, for a total of 1,020. The order $(AB)C$ is much worse: 7500 multiplications. For general n, exhaustive search is prohibitive for the following reason: Let

$$T(n) = \text{number of ways to parenthesize a product of } n \text{ matrices},$$

then

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) \tag{4}$$

with $T(2) = T(1) = 1$. The solution to recurrence 4 is

$$T(n) = \frac{1}{n}\binom{2n-2}{n-1}, \tag{5}$$

which is $\Omega(4^n/n^2)$.

Clearly the principle of optimality applies. If the parenthesization is optimal for $M = M_1, \ldots, M_n$ and there are two parts $A = M_1, \ldots, M_k$ and $B = M_{k+1}, \ldots, M_n$ such that $M = AB$, then the parenthesizations for A and B are optimal. Let

$$M[i, j] = \text{optimal number of multiplications to compute } M_i \cdots M_j.$$

Thus, the recurrence is

$$M[i, j] = \min_{i \le k < j} \{M[i, k] + M[k + 1, j] + d_{i-1} d_k d_j\} \text{ for } 1 \le i < j \le n, \quad (6)$$

with $M[i, i] = 0$.

The dynamic program fills a table with entries for $i \le j$, starting with the main diagonal, which is set to $M[i, i] = 0$. The computation then progresses to $M[i, i+1]$ and so forth until the element in the north-east corner $M[1, n]$ is reached. In order to be able to recover the solution, the index k which gives the minimum in 6 is also stored. When the algorithm reaches $(1, n)$, this index gives the index of the first cut. One then proceeds recursively on both sides to construct the actual parenthesization. Figure 3 gives an example.

2.3 Shortest Paths

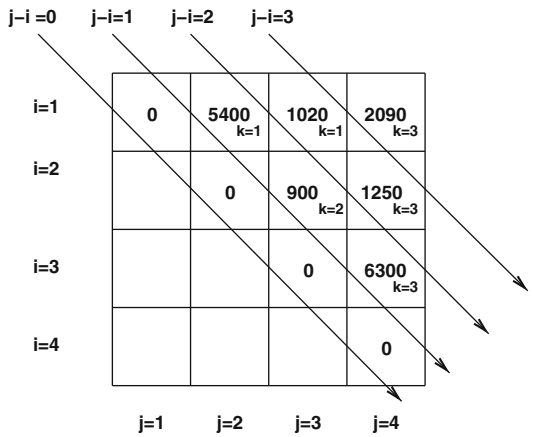
Consider a directed graph on nodes $V = \{1, \ldots, n\}$ with distance matrix $D[i, j] \ge 0$, where $D[i, j] = \infty$ if there is no edge between node i and node j. The object is to calculate the shortest path between each pair of nodes. To use dynamic programming, one defines

$$D_k[i, j] = \text{length of a shortest path from } i \text{ to } j \text{ which only uses nodes } \{1, \ldots, k\}.$$

Clearly $D_0 = D$. Starting with D_0 one calculates $D_1, D_2, \ldots D_n$ using the following recursion:

$$D_k[i, j] = \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}. \quad (7)$$

The algorithm, which computes this series of tables, is called the Floyd algorithm. The reader is invited to follow the calculations in Table 2, which gives an example for the graph in Fig. 4. The principle of optimality is at work here: If a path from node i to node j is optimal and passes through node k, then the path from i to k, as well as the path from k to j, is optimal. This is not true for the longest simple path problem. (A simple path is a path without repeated nodes.) In graph of Fig. 4, the longest simple path from node 2 to node 4 is $2, 1, 3, 4$, but the path $2, 1$ is not a longest simple path from 2 to 1. Indeed the problem "longest simple path" is $\mathcal{NP}$-hard. This does not contradict the fact that the Floyd algorithm works for



$$
\begin{aligned}
& M[1,1] = M[2,2] = M[3,3] = 0 \\
M[1,2] &= \mathbf{M[1,1]} + \mathbf{M[2,2]} + \mathbf{12 \times 5 \times 90} = 5400, \text{ store } k = 1 \\
M[2,3] &= \mathbf{M[2,2]} + \mathbf{M[3,3]} + \mathbf{5 \times 90 \times 2} = 900, \text{ store } k = 2 \\
M[3,4] &= M[3,3] + M[4,4] + 90 \times 2 \times 35 = 6300, \text{ store } k = 3 \\
M[1,3] &= \min\{\mathbf{M[1,1]} + \mathbf{M[2,3]} + \mathbf{12 \times 5 \times 2}, M[1,2] + M[3,3] + 12 \times 90 \times 2\} \\
&= 1020, \text{ store } k = 1 \\
M[2,4] &= \min\{M[2,2] + M[3,4] + 5 \times 90 \times 35, \mathbf{M[2,3]} + \mathbf{M[4,4]} + \mathbf{5 \times 2 \times 35}\} \\
&= 1250, \text{ store } k = 3 \\
M[1,4] &= \min\{M[1,1] + M[2,4] + 12 \times 5 \times 35, M[1,2] + M[3,4] + 12 \times 90 \times 35, \\
&\qquad \mathbf{M[1,3]} + \mathbf{M[4,4]} + \mathbf{12 \times 2 \times 35}\} \\
&= 2090, \text{ store } k = 3
\end{aligned}
$$

Optimal Parenthesization: $(M_1(M_2M_3)M_4)$

Fig. 3 An example for the dynamic program for the problem "chained matrix multiplication." In the example, $n = 4$ and the dimensions are $(12, 5, 90, 2, 35)$. The calculation proceeds by first filling the table for indices with $j - i = 0$ (i.e., initialization of $M[i, i]$) and then continues along the diagonals $j - i = 1, 2, 3$. Calculations which give the minima for each cell in Eq. 6 are shown in bold type, and the corresponding index k is stored. Once the value of $M[1, 4]$ is known, then the solution can be recovered using these indices: $(M_1M_2M_3)M_4$ can be concluded from the index $k = 3$ in cell $(1, 4)$. Next, look up cell $(1, 3)$ to find the parenthesization $M_1(M_2M_3)$

negative distances if there are no negative cycles. (Longest simple path cannot be reduced to shortest path with negative distances because of the cycle restriction.) The Floyd algorithm can be used to detect negative cycles by checking for negative elements in the main diagonal of D_n.

The time complexity of Floyd algorithm is $O(n^3)$. The calculation can be performed by keeping only D_{k-1} and D_k at each iteration, and thus the space

Table 2 The Floyd algorithm for the example of Fig. 4. One proceeds from matrix D_{k-1} to matrix D_k by applying the rule $D_k[i,j] = \min\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$ for all i and j. In the lower right corner of the table, the matrix of pointers P is shown. For example, the length of a shortest path from node 3 to node 2 is 4 since $D_4[3,2] = 4$. To construct the path look up $P[3,2] = 4$ to find that the path goes through node 4. Recursively, look up $P[3,4] = P[4,2] = 0$. Thus the shortest path is 3, 4, 2

D_0	0	2	4	3	D_1	0	2	4	3
	3	0	∞	3		3	0	7	3
	5	∞	0	3		5	7	0	3
	∞	1	4	0		∞	1	4	0
D_2	0	2	4	3	D_3	0	2	4	3
	3	0	7	3		3	0	7	3
	5	7	0	3		5	7	0	3
	4	1	4	0		4	1	4	0
D_4	0	2	4	3	P	0	0	0	0
	3	0	7	3		0	0	1	0
	5	4	0	3		0	4	0	0
	4	1	4	0		2	0	0	0

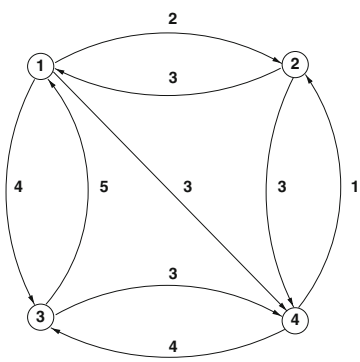


Fig. 4 A directed graph with distances

requirement is $O(n^2)$. In order to retrieve the actual path (and not only its length), a matrix P of pointers is used, where $P[i,j]$ contains the number of the last iteration k that causes a change in $D_k[i,j]$ ($P[i,j]$ is initialized to 0). To construct the path between node i and j, look up $P[i,j]$ at the end. If $P[i,j] = 0$, then there was never a change for any D_k and the shortest path is the edge (i,j), else the shortest path goes through k. Recursively examine $P[i,k]$ and $P[k,j]$.

The resulting algorithm is named after Floyd (It appeared as a one-page note in the Communications of the ACM – together with other algorithms of the time. Therefore, the antiquated title is "Algorithm 97: Shortest Path," Floyd [55]).

Another algorithm of the time is the matrix algorithm, originally given in the context of the transitive closures by Warshall. The paper is a short two-pager; see Warshall [92].

Define

$$D^{(k)}[i,j] = \text{length of a shortest path from } i \text{ to } j \text{ containing at most } k \text{ edges}$$

and set

$$D^{(0)} == \begin{cases} 0 & \text{if } i = j \\ \infty & \text{else.} \end{cases}$$

Clearly by the principle of optimality,

$$D^{(k)}[i,j] = \min\{D^{(k-1)}[i,j], \min_{1\le\ell\le n, \ell\neq j}\{D^{(k-1)}[i,\ell] + D[l,j]\}\} \tag{8}$$

$$= \min_{1\le\ell\le n}\{D^{(k-1)}[i,\ell] + D[l,j]\}. \tag{9}$$

The previous equation defines a matrix multiplication where the usual multiplication means "+" and the operation "+" means "min." Table 3 gives an example. In other words,

$$D^{(k)} = D^k.$$

The solution to the shortest path problem is therefore obtained by calculating D^k with the operators properly replaced. The run time of this algorithm is at first glance $O(n^4)$, but it can be improved by "repeated squaring": Calculate

- $D^2 = D \cdot D$
- then $D^4 = D^2 \cdot D^2$
- then $D^8 = D^4 \cdot D^4$, and so forth

If $n-1$ is not a power of 2, then D^{n-1} can be obtained by going up to the highest power of 2 smaller than $(n-1)$ and multiplying the powers of the binary representation of $(n-1)$. As a result the number of matrix multiplications is only logarithmic, giving a run time of $O(n^3 \log n)$.

2.4 The Knapsack Problem

This is a classical problem in combinatorial optimization: Given are n items $\{1,\ldots,n\}$ with weights $w_i > 0$ and profits $p_i > 0$, and there is a knapsack of weight capacity $W > 0$. One is to fill the knapsack in such a way that the profit of the items chosen is maximized while obeying that the total weight be less than W. Let

$$P[i,j] = \text{maximum profit, which can be obtained if the}$$
$$\text{weight limit is } j \text{ and only items from } \{1,\ldots i\} \text{ may be included,}$$

Table 3 Example of the progression of the matrix algorithm for the all shortest path problem given in Fig. 4. The operations in the "matrix multiplication" are not the usual "+" and "·" but rather "min" and "+"

$$I \cdot D = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{pmatrix} = D^{(1)}$$

$$D^{(1)} \cdot D = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 4 & 0 & 3 \\ 4 & 1 & 4 & 0 \end{pmatrix} = D^{(2)}$$

$$D^{(2)} \cdot D = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 4 & 0 & 3 \\ 4 & 1 & 4 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & 3 \\ \infty & 1 & 4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 4 & 0 & 3 \\ 4 & 1 & 4 & 0 \end{pmatrix} = D^{(3)}$$

where $1 \leq i \leq n$ and $0 \leq j \leq W$. The following recursion is valid:

$$P[i, j] = \max\{P[i-1, j], P[i-1, j-w_i + p_i\}, \tag{10}$$

where $P[0, j] = 0$ and $P[i, j] = -\infty$ when $j < 0$. Equation 10 says that for a new element i one does not include it (left of max) or one does include it (right of max). If one includes the element, then by the principle of optimality the problem with capacity reduced by the weight of element i on the earlier elements $\{1, \ldots, i-1\}$ must be optimal.

The table $P[i, j]$ can be filled in either row by row or column by column fashion. The run time of this algorithm is $\Theta(nW)$. Note that this is not a polynomial algorithm for the knapsack problem. The input size is $O(n \log \max_{i=1,\ldots,n} w_i + \log W)$, which means that $\Theta(nW)$ is exponential in the input size. This is, of course, to be expected as the knapsack problem is $\mathcal{NP}$-hard. The algorithm's complexity is called *pseudo-polynomial*. Many dynamic programs give pseudo-polynomial algorithms.

Pseudo-polynomial, Strongly Polynomial. Let s be the input to some decision problem Π. Let $|s|_{\log}$ be the length of the input – that is, the length of the binary encoding – and let $|s|_{\max}$ be the magnitude of the largest number in s. A problem Π is pseudo-polynomially solvable if there is an algorithm for Π with run time bounded by a polynomial function in $|s|_{\max}$ and $|s|_{\log}$. A decision problem is a *number problem* if there exists no polynomial p such that $|s|_{\max}$ is bounded by $p(|s|_{\log})$ for all input s. (For example, the decision version of the knapsack problem is a number problem.) Note that by these definitions it immediately follows that an $\mathcal{NP}$-complete *non*-number problem (such as HAMILTONIAN CIRCUIT, for

Table 4 An example of the pseudo-polynomial dynamic programming algorithm for the knapsack problem on nine items. The weights are $w[1] = 1, w[2] = 2, w[3] = 3, w[4] = 4, w[5] = 5, w[6] = 6, w[7] = 7, w[8] = 8, w[9] = 9$ and the profits are $p[1] = 1, p[2] = 2, p[3] = 5, p[4] = 10, p[5] = 15, p[6] = 16, p[7] = 21, p[8] = 22, p[9] = 35$. The size of the knapsack is 15. The table shows in position (i, j) the maximum profit, which can be obtained if the weight limit is j and only items from $\{1, \ldots i\}$ may be included. Bold entries indicate that in Eq. 10 the second choice is the maximizer, i.e., the new item is considered for inclusion. From the regular-bold information the actual solution can be reconstructed: The last entry in the table is 50. Because of the bold type face item $i = 9$ is included. Thus one looks $w_9 = 9$ many cells to the left in the previous row. The regular type face of 16 in cell $(7, 6)$ means that item 8 is not included, looking at the cells above neither are items 7 and 6. Item 5 is included, next look-up cell $(4, 1)$ to see that no more items are included. The solution is $\{5, 9\}$ with a profit of 50

W	0	1	2	3	4	5	6	7	8	9	11	12	13	14	15
i=1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=2	0	1	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
i=3	0	1	2	**5**	**6**	**7**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
i=4	0	1	2	5	**10**	**11**	**12**	**15**	**16**	**17**	**18**	**18**	**18**	**18**	**18**
i=5	0	1	2	5	10	**15**	**16**	**17**	**20**	**25**	**26**	**27**	**30**	**31**	**32**
i=6	0	1	2	5	10	15	16	17	20	25	26	**31**	**32**	**33**	**36**
i=7	0	1	2	5	10	15	16	**21**	**22**	25	26	31	**36**	**37**	**38**
i=8	0	1	2	5	10	15	16	21	22	25	26	31	36	37	38
i=9	0	1	2	5	10	15	16	21	22	**35**	**36**	**37**	**40**	**45**	**50**

example) cannot be solved by a pseudo-polynomial algorithm (unless $\mathcal{P} = \mathcal{NP}$). For polynomial p let Π_p denote the subproblem which is obtained by restricting Π to instances with $|s|_{\max} \leq p(|s|_{\log})$. Problem Π is called *strongly $\mathcal{NP}$-complete* if P is in $\mathcal{NP}$and there exists a polynomial p for which Π_p is $\mathcal{NP}$-complete. It is easy to see (Table 4):

Theorem 1 *A strongly $\mathcal{NP}$-complete problem cannot have a pseudo-polynomial algorithm unless $\mathcal{P} = \mathcal{NP}$.*

2.5 Binary Search Tress

The construction of optimal binary search trees is a classic optimization problem. One is interested in constructing a search tree, in which elements can be looked up as quickly as possible. The first dynamic program for this problem was given by Gilbert and Moore [59] in the 1950s. More formally, given are n search keys with known order $\text{Key}_1 < \text{Key}_2 < \cdots < \text{Key}_n$. The input consists of $2n + 1$ probabilities $p_1, \ldots, p_n$ and $q_0, q_1, \ldots, q_n$. The value of p_l is the probability that a search is for the value of Key_l; such a search is called *successful*. The value of q_l is the probability that a search is for a value between Key_l and Key_{l+1} (set $\text{Key}_0 = -\infty$

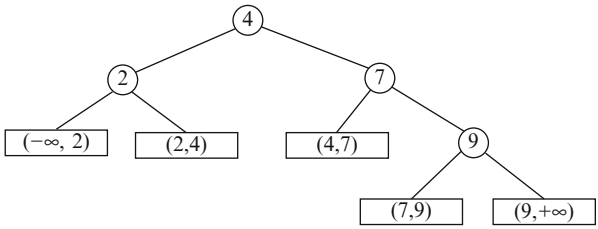


Fig. 5 A binary search tree with successful (*round*) and unsuccessful (*rectangular*) nodes

and $\text{Key}_{n+1} = \infty$); such a search is called *unsuccessful*. Note that in the literature the problem is sometimes presented with *weights* instead of *probabilities*, in that case the p_l and q_l are not required to add up to 1.

The binary search tree constructed will have n internal nodes corresponding to the successful searches, and $n+1$ leaves corresponding to the unsuccessful searches. The *depth* of a node is the number of edges from the node to the root. Denote $d(p_l)$ the depth of the internal node corresponding to p_l and $d(q_l)$ the depth of the leaf corresponding to q_l. A successful search requires $1 + d(p_l)$ comparisons, and an unsuccessful search requires $d(q_l)$ comparisons. See Fig. 5. So, the expected number of comparisons is

$$\sum_{1 \le l \le n} p_l \, (1 + d(p_l)) + \sum_{0 \le l \le n} q_l \, d(q_l). \tag{11}$$

The goal is to construct an *optimal binary search tree* that minimizes the expected number of comparisons carried out, which is (11).

Let $B_{i,j}$ be the expected number of comparisons carried out in a optimal subtree containing the keys $\text{Key}_{i+1} < \text{Key}_2 < \cdots < \text{Key}_j$. Observing that in a search the probability to search in the region between Key_{i+1} and Key_j is $\sum_{l=i+1}^{j} p_l + \sum_{l=i}^{j} q_l$ it is clear that the following recurrence holds:

$$B_{i,j} = \begin{cases} 0, & \text{if } i = j; \\ \displaystyle\sum_{l=i+1}^{j} p_l + \sum_{l=i}^{j} q_l + \min_{i<t\le j} \left(B_{i,t-1} + B_{t,j} \right), & \text{if } i < j, \end{cases} \tag{12}$$

where the cost of the optimal binary search tree is $B_{0,n}$. Calculating $B_{i,j}$ requires $O(j-i)$ time, thus calculating all of the $B_{i,j}$ requires $O(n^3)$ time.

2.6 Pyramidal Tours for the Traveling Salesman Problem

In the traveling salesman problem one is given a set of n "cities" $V = \{1, \ldots, n\}$ as well as a distance matrix $d[i, j]$. Find a permutation t ("the tour"), such that

$$f(t) = d(t(n), t(1)) + \sum_{i=1}^{n-1} d(t(i), t(i+1))$$

is minimized. This problem is well studied and it is known to be $\mathcal{NP}$-hard. (A good resource on the Traveling Salesman Problem is the "guided tour book" by Lawler, Lenstra, Rinnoy Kan, and Shmoys [79]). A tour t is said to be *pyramidal* if, t is of the form $1, i_1, i_2, i_3, \ldots n, j_1, \ldots, j_{n-r-2}$, where $1 < i_1 < i_2 < i_3 < \ldots n$ and $j_1 > j_2 > j_3 > \ldots > j_{n-r-2}$. A pyramidal tour can be found by dynamic programming in $\Theta(n^2)$. To this end, let $H[i, j]$ be the length of a shortest Hamiltonian path from i to j subject to the condition that the path goes from i to 1 in descending order followed by the rest in ascending order from 1 to j. The reader is encouraged to verify that by the principle of optimality the following recursion holds:

$$H[i,j] = \begin{cases} H[i, j-1] + d[j-1, j] & \text{for } i < j-1, \\ \min_{k<i}\{H[i,k] + d[k,j]\} & \text{for } i = j-1, \\ H[i-1, j] + d[i, i-1] & \text{for } i < j+1, \\ \min_{k<j}\{H[k,j] + d[j,k]\} & \text{for } i = j+1. \end{cases} \tag{13}$$

Then the cost of shortest pyramidal tour is

$$\min\{H[n, n-1] + d[n-1, n], H[n-1, n] + d[n, n-1]\}.$$

A matrix d is *Monge* if for all $i < i'$ and $j < j'$, $d[i, j] + d[i', j'] \leq d[i', j] + d[i, j']$. If the matrix d is a Monge matrix then it is easy to see that there exists an optimal tour, which is pyramidal.

3 Open-ended Dynamic Programming: Work Functions

Dynamic programs are useful in decision making for problems where new request are constantly added, and updates need to be performed. The *k-server problem*, originally given by Manasse, McGeoch, and Sleator [82], is defined as follows: one is given $k \geq 2$ mobile servers which reside in a metric space M. A sequence of

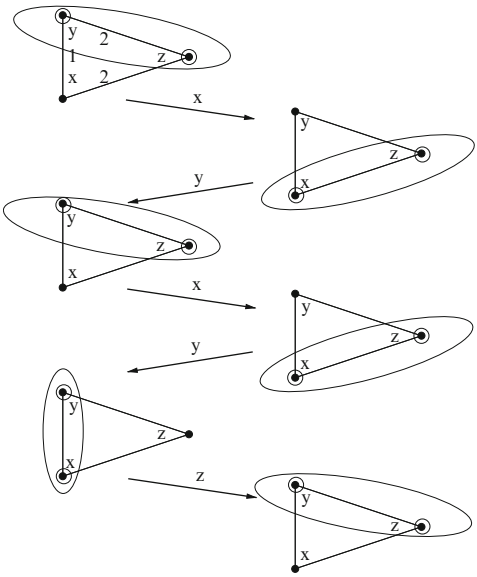


Fig. 6 The two points x and y are at distance 1 and a point z is at distance 2 from x and y. The request sequence is $xyxyz$. The solution, i.e., the positions of the servers for each request, are circled. The solution has a cost of 7, which is not optimal since it is better to move the server at point z for the first and last request

requests is issued, where each request is specified by a point $r \in M$. To "satisfy" this request, one of the servers must be moved to r, at a cost equal to the distance from its current location to r. (If a request is to a point that already has a server the cost is zero.) The goal is to minimize the total service cost. An algorithm $\mathcal{A}$ for the k-server problem computes a solution which determines which server is moved at each step. Figure 6 gives a very simple example for the 2-server problem, i.e., the server problem for $k = 2$ in a metric space with only three points. Interestingly, a very similar metric space is powerful enough to model the noted "ski rental problem", see Karlin, Manasse, Rudolph, and Sleator [68] for details.

This problem can be solved by dynamic programming calculating *work functions*, when the length of the request sequence and all the requests are known in advance. This standard version of the problem is also called the offline version of the k-server problem. Credit for introducing work functions goes to Larmore and Chrobak [42].

Work functions provide information about the optimal cost of serving the past request sequence. For a request sequence ϱ, by $\omega_\varrho(X)$ one denotes the minimum cost of serving ϱ and ending in configuration X – an unordered k-tuples of points. The function ω_ϱ is called the work function after request sequence ϱ. The notation ω is used to denote any work function ω_ϱ, for some request sequence ϱ. Immediately from the definition of work functions it can be concluded that the optimal cost to service ϱ is $opt(\varrho) = \min_X \omega_\varrho(X)$.

For given ϱ, the work function ω_ϱ can be computed using dynamic programming. Initially, $\omega_\epsilon(X) = S^0 X$, for each configuration X (ϵ is the empty request sequence). For a non-empty request sequence ϱ, if r is the last request in ϱ, write $\varrho = \sigma r$.

Then ω_ϱ can be computed recursively as $\omega_\varrho = \omega_\sigma \wedge r$, where "$\wedge$" is the *update operator* defined as follows:

$$\omega \wedge r(X) = \min_{Y \ni r} \{\omega(Y) + \text{dist}(Y, X)\} \tag{14}$$

Here $\text{dist}(Y, X)$ denotes the minimum-matching distance between X and Y. Note that $|\omega(X) - \omega(Y)| \leq XY$ for any work function ω and any configurations X and Y. This inequality is called the *Lipschitz property*. A set of configurations $S = \{X_1, X_2, \dots\}$ is said to support a work function ω if, for any configuration Y, there exists some $X \in S$ such that $\omega(Y) = \omega(X) + \text{dist}(X, Y)$. If ω is supported by a finite set (which it usually is), then there is a unique minimal set S which supports ω, which is called the work function support of ω. Note the following: If $r \in X$, then $\omega \wedge r(X) = \omega(X)$. If $r \notin X$, let Y be the configuration that contains r and minimizes $\omega(Y) + YX$, and let x be the point in X that is matched to r in the minimum matching between X and Y. Then $\omega(Y) + YX = \omega(Y) + rx + Y(X - x + r) \geq \omega(X - x + r) + rx$. Thus the update formula Eq. 14 can be rewritten as $\omega \wedge r(X) = \min_{x \in X} \{\omega(X - x + r) + rx\}$. Also note that in calculating the functions ω_ϱ one need only keep track of the value of the function at their support. This is important as the number of configurations in the domain of ω_ϱ grows as requests grows. Figure 7 shows how to calculate an optimal solution, the progression of ω_ϱ and support for the example in Fig. 6.

In practice, requests might be given one at a time and the algorithm then has to make a decision about which server to move before future requests are known. An algorithm $\mathcal{A}$ is said to be *online* if its decisions are made without the knowledge of future requests. It is unlikely that such an algorithm would achieve optimality. Similar to approximation algorithms, the quality of the algorithm is measured by comparing against the offline cost: $\mathcal{A}$ is C*-competitive* if the cost incurred by $\mathcal{A}$ to service each request sequence ϱ is at most C times the optimal (offline) service cost for ϱ, plus possibly an additive constant independent of ϱ. The *competitive ratio of* $\mathcal{A}$ is the smallest C for which $\mathcal{A}$ is C-competitive. The competitive ratio is frequently used to study the performance of online algorithms for the k-server Problem, as well as other optimization problems. The reader is referred to the book of Borodin and El-Yaniv [30] for a comprehensive discussion of competitive analysis.

It is interesting to note that the work functions ω_ϱ play a central role in the algorithm with current best competitiveness for the k-server problem: Work Function Algorithm. The Work Function Algorithm chooses its service of the request sequence ϱ as follows: Suppose that WFA is in configuration S, and that the current work function is ω. On request r, WFA chooses some $x \in S$ which minimizes $xr + \omega \wedge r(S - x + r)$, and moves the server from x to r. If there is more than one choice which minimizes that quantity, the choice is arbitrary. WFA can be seen as a "linear combination" of two greedy strategies. The first one, a *short-sighted greedy*, minimizes the cost xr in the given step. The second, a *retrospective greedy*, chooses the optimal configuration after r, that is, the configuration $S - x + r$ that minimizes $\omega \wedge r(S - x + r)$. The short-sighted greedy strategy is not competitive for any k. The retrospective greedy strategy is not competitive for $k \geq 3$, and its

		$\omega_\varrho(\{y,z\})$	$\omega_\varrho(\{x,z\})$	$\omega_\varrho(\{x,y\})$
initial	ω_ϵ:	$\underline{0}$	1	2
request x	ω_x:	2	$\underline{1}$	$\underline{2}$
request y	ω_{xy}:	$\underline{2}$	3	$\underline{2}$
request x	ω_{xyx}:	4	$\underline{3}$	$\underline{2}$
request y	ω_{xyxy}:	4	4	$\underline{2}$
request z	ω_{xyxyz}:	$\underline{4}$	$\underline{4}$	6

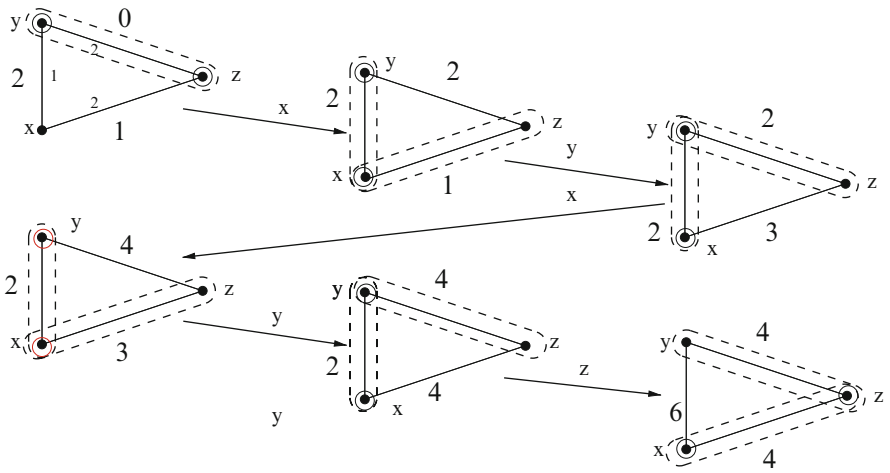


Fig. 7 Calculating the optimal solution using dynamic programming for the example in Fig. 6. The work functions ω_ϱ are shown in the figure as values adjacent to the corresponding pairs of points. Equivalently, the work functions can be written into the traditional dynamic programming table (*top*). The support is marked by *ovals* in the figure and is *underlined* in the table. The minimum value of the last row is 4 – the optimal value

competitive ratio for $k = 2$ is at least 3. The reader might verify that the service sequence depicted in Fig. 6 shows the steps of WFA for that example.

Manasse, McGeoch, and Sleator [82], have proved the following:

Theorem 2 *No online algorithm for k servers has competitive ratio smaller than k if a metric space has at least $k + 1$ points.*

They also give the *k-server conjecture* which states that, for each k, there exists an online algorithm for k servers which is k-competitive in any metric space. For $k > 2$, this conjecture has been settled only in a number of special cases, including trees and spaces with at most $k+2$ points. (cf. the work of Chrobak, Karloff, Payne, and Vishwanathan [44], the work of Chrobak and Larmore [41] and the work of Koutsoupias and Papadimitriou [72].) Koutsoupias and Papadimitriou have shown:

Theorem 3 *The Work Function Algorithm is* $(2k-1)$*-competitive for* k *servers in arbitrary metric spaces.*

Thus a wide gap remains. Even some simple-looking special cases remain open, for example the 3-server problem on the circle, in the plane, or in 6-point spaces. Chrobak and Larmore [42] (see also [43]) prove:

Theorem 4 *The Work Function Algorithm is 2-competitive for* $k = 2$*.*

Bein, Chrobak, and Larmore [15] show:

Theorem 5 *The Work Function Algorithm is 3-competitive for* $k = 3$ *if the metric space* M *is the Manhattan plane.*

4 Intricate Dynamic Programming: Block Deletion in Quadratic Time

Sorting problems under various operations have been studied extensively, including work on sorting with prefix reversals, transpositions and block moves. This section contains an example from this realm and shows that intricate setup of dynamic programming can speed up dynamic programming schemes.

4.1 Preliminaries

Define a *permutation* of length n to be a list $x = (x_1, \ldots, x_n)$ consisting of the integers $\{1, \ldots, n\}$ where each number occurs exactly once. For any $1 \leq i \leq j \leq n$ denote the sublist of x that starts at position i and ends with position j by $x_{i\ldots j}$. A list y is a *subsequence* of x if y is obtained from x by deleting any number of elements. For example, $(2, 3)$ is a subsequence of $(2, 4, 3, 1)$, but not a sublist. Since x has no duplicate symbols, a subsequence of x is uniquely characterized by its set of items. By a slight abuse of notation, one identifies a subsequence with the set of its items. Define the *closure* of a subsequence of x to be the smallest sublist of x which contains it. For example, the closure of the subsequence $(2, 3)$ of $(2, 4, 3, 1)$ is the sublist $(2, 4, 3)$. If A and A' are subsequences of a list x, say that A and A' are *separated* if the closures of A and A' are disjoint.

A block deletion sequence for a subsequence y of x consists of a sequence $A_1, \ldots, A_m$ of disjoint non-empty subsequences of y such that
1. for all $i = 2, \ldots, m$, A_i is a block in $y - \bigcup_{u=1}^{i-1} A_u$, and
2. $y - \bigcup_{u=1}^{m} A_u$ is a monotone increasing list.

For example, a minimum length block deletion sequence for the list $(1, 4, 2, 5, 3)$ consists of two steps. First delete the block (2), obtaining $(1, 4, 5, 3)$, then delete the block $(4, 5)$, obtaining the sorted list $(1, 3)$. Figure 8 shows another example of a block deletion sequence. A complete block deletion sequence for a subsequence y

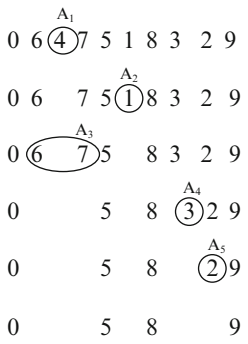


Fig. 8 A block deletion sequence. There are five steps: $A_1, \ldots, A_5$

of x consists of a block deletion sequence $A_1, \ldots, A_m$ of y such that $y - \bigcup_{u=1}^{m} A_u$ is the empty list.

4.2 A Dynamic Program for Complete Block Deletion

Consider first the complete block deletion problem for all sublists of x, which will be solved in quadratic time by dynamic programming. Once the $O(n^2)$ answers to this problem are obtained, the original block deletion problem can be solved in quadratic time. The following three lemmas are used:

Lemma 1 *If $A_1, \ldots, A_m$ is a block deletion sequence for a sublist y of x, and $1 \leq u < v \leq m$, then either A_u and A_v are separated, or A_u is a subsequence of the closure of A_v.*

Proof The closure of A_u cannot contain any item of A_v, since otherwise A_u could not be deleted before A_v. If *all* items of A_v are before A_u or *all* items of A_v are after A_u, then A_u and A_v are separated. If some items of A_v are before A_u and some items are after A_u, then A_u is a subsequence of the closure of A_v. □

Lemma 2 *If $A_1, \ldots, A_t, A_{t+1}, \ldots, A_m$ is a block deletion sequence for a sublist y of x, and A_t and A_{t+1} are separated, then A_t and A_{t+1} may be transposed, i.e., $A_1, \ldots, A_{t+1}, A_t, \ldots, A_m$ is a block deletion sequence for y.*

Proof For any u, let $y_u = x - \bigcup_{v<u} A_v$. By definition, A_t is a block of y_t, and A_{t+1} is a block of $y_{t+1} = y_t - A_t$. Since A_t and A_{t+1} are separated, A_{t+1} is also a block of y_t. Thus, A_{t+1} can be deleted before A_t. □

Lemma 3 *For any $1 \leq i \leq j \leq n$, if there is a complete block deletion sequence for $x_{i\ldots j}$ of length m, then there is a complete block deletion sequence for $x_{i\ldots j}$ of length m such that x_i is deleted in the last step.*

Proof Let $A_1, \ldots, A_m$ be a complete block deletion sequence of $x_{i\ldots j}$. Suppose that $x_i \in A_t$ for some $t < m$. Since x_i is deleted in the t^{th} move of the sequence, all deletions after that must involve blocks whose first symbol occurs to the right of x_i in x. That is, for any $v > t$, x_i cannot be an item of the closure of A_v, hence, by Lemma 1, A_t and A_v must be separated. By Lemma 2, one can transpose A_t with A_v for each $v > t$ in turn, moving A_t to the end of the deletion sequence. □

Next is given a recurrence to compute the minimum length of a complete block deletion sequence. This recurrence will be used in Algorithm 1 .

Theorem 6 *Given a permutation x, let $t_{i,j}$ denote the minimum length of any complete block deletion sequence for the sublist $x_{i\ldots j}$. The values of $t_{i,j}$ can be computed inductively by the following: Set $t_{i,i} = 1$, $t_{i,j} = 0$ for $i > j$, and for $i < j$ set*

$$t_{i,j} = \begin{cases} \min\{1 + t_{i+1,j},\ t_{i+1,\ell-1} + t_{\ell,j}\}, & \text{if } \exists\ \ell \in \{i+1, \ldots, j\} \text{ such that } x_\ell = x_i + 1, \\ 1 + t_{i+1,j}, & \text{otherwise.} \end{cases} \tag{15}$$

Proof The proof is by induction on $j - i$ to show that $t_{i,j}$ is computed correctly. The base case is trivial, namely $t_{i,i} = 1$, because it takes one block deletion to delete a sublist of length 1.

For the inductive step, assume that $t_{i,j}$ is the length of a minimum block deletion sequence for $x_{i\ldots j}$ when $j - i < k$. For $j - i = k$, let $m = t_{i,j}$ and let $A_1, \ldots, A_m$ be a corresponding minimum length complete block deletion sequence of $x_{i\ldots j}$. By Lemma 3, one can insist that $x_i = a$ is an item in A_m. Consider now two cases based on whether there is an ℓ with $i < \ell \leq j$ such that $x_\ell = a + 1$ (The reader might also consult Fig. 9).

If the element $a + 1$ does not occur in the interval $\{i + 1, \ldots, j\}$, then the element a is not part of a block in this interval and must be deleted by itself. So, $A_1, \ldots, A_{m-1}$ is a complete block deletion sequence of $x_{i+1\ldots j}$. Note that it must be of optimum length, for if $B_1, \ldots, B_r$ were a shorter complete block deletion sequence of $x_{i+1\ldots j}$, then $B_1, \ldots, B_r, \{a\}$ would be shorter than $A_1, \ldots, A_m$. Thus, as $j - (i + 1) = k - 1$, by the induction hypothesis $t_{i+1,j} = m - 1$. So $t_{i,j} = 1 + t_{i+1,j}$, which is optimum.

Now for the case that the element $x_\ell = a + 1$ does occur in the interval $\{i + 1, \ldots, j\}$. This means that $a + 1$ and possibly other larger values can be included in A_m when element a is deleted. If element $a + 1$ is not included in A_m then the same argument used in the previous paragraph shows that $m = 1 + t_{i+1,j}$. If on the other hand $a + 1$ is included in A_m, then by Lemma 1, for any t, $(1 \leq t \leq m)$, A_t is either completely before or completely after the element $a + 1$, since A_m is deleted after A_t. By Lemma 2, one can permute the indices so that, for some $u < m$,

1. if $t \leq u$, then A_t is a subsequence of $x_{i+1\ldots\ell-1}$, and
2. if $u < t \leq m$, then A_t is a subsequence of $x_{\ell+1\ldots j}$.

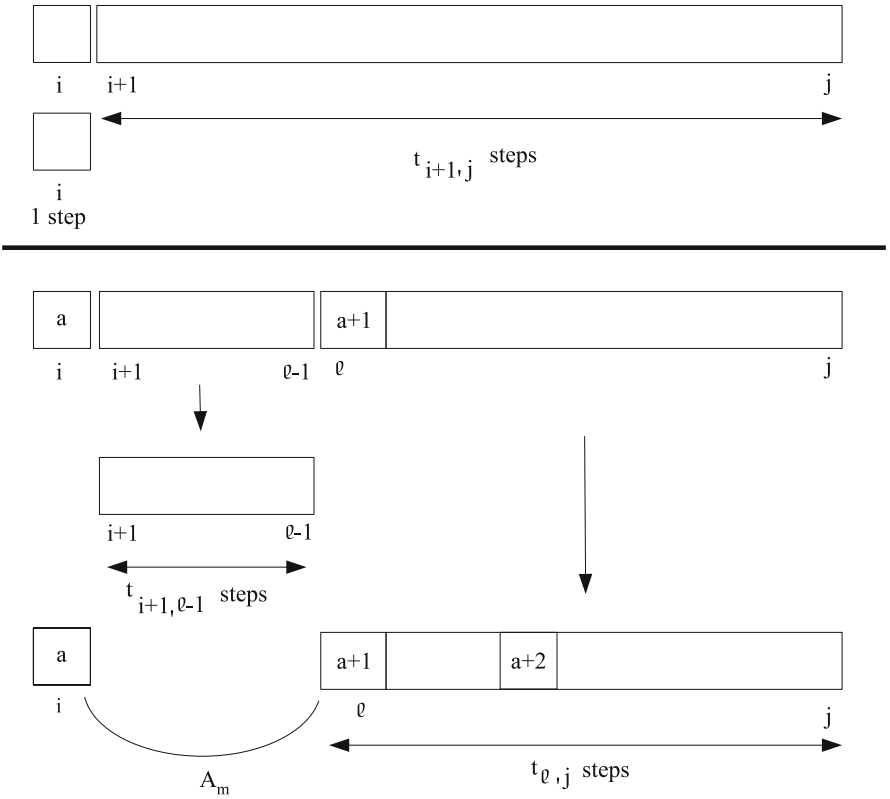


Fig. 9 The recurrence for the $t_{i,j}$ values

Consequently, $A_1, \ldots, A_u$ is a block deletion sequence for $x_{i+1\ldots\ell-1}$ and $A_{u+1}, \ldots, A_m - \{a\}$ is a block deletion sequence for $x_{\ell+1\ldots j}$. Both of these block deletion sequences must be optimum for their respective intervals. That is, for example, if $B_1, \ldots, B_r$ were a shorter complete block deletion sequence for $x_{i+1\ldots\ell-1}$, then $B_1, \ldots B_r, A_{u+1}, \ldots A_m$ would be a complete block deletion sequence for $x_{i\ldots j}$, contradicting the minimality of m. By the induction hypothesis, as $\ell - 1 - (i + 1) < j - i$ and $j - \ell < j - i$, it follows that $t_{i+1,\ell-1} = u$ and $t_{\ell,j} = m - u$, so $t_{i,j} = t_{i+1,\ell-1} + t_{\ell,j} = u + (m - u) = m$. □

The resulting dynamic programming algorithm BLOCKDELETION, which is derived from the recurrence of Theorem 6, is shown below.

Next the analysis of the run time of algorithm BLOCKDELETION is considered. Let $z = (z_1, \ldots, z_n)$ be the inverse permutation of x, i.e., $x_i = k$ if and only if $z_k = i$. Note that z can be computed in $O(n)$ preprocessing time.

Theorem 7 *Algorithm* BLOCKDELETION *has run time* $O(n^2)$.

Algorithm 1 BLOCKDELETION(X)

Let n be the number of elements in x
for $i \leftarrow 1$ **to** n **do**
  $t[i,i] \leftarrow 1$
for $k \leftarrow 2$ **to** n **do**
  for $i \leftarrow 1$ **to** $n-k+1$ **do**
    Let $x_\ell \leftarrow x_i + 1$; $j \leftarrow i+k-1$
    if $i < \ell \leq j$
    then $t[i,j] \leftarrow \min\{(1+t[i+1,j]),\ (t[i+1,\ell-1]+t[\ell,j])\}$
    else $t[i,j] \leftarrow 1+t[i+1,j]$
return

Proof To prove the theorem one shows that if $t_{u,v}$ are already known for all $i < u \leq v \leq j$, then $t_{i,j}$ can be computed in $O(1)$ time. Let $m = t_{i,j}$ for $i < j$ and let A_t be the subsequence of $x_{i\ldots j}$ that is deleted at step t of the complete block deletion sequence of $x_{i\ldots j}$ of length m. By Lemma 3, assume that $x_i \in A_m$. If $|A_m| > 1$, the index $\ell = z_{x_i+1}$, i.e., $x_\ell = 1 + x_i$, can be found in $O(1)$ time, since one has already spent $O(n)$ preprocessing time to compute the array z. The recurrence thus takes $O(1)$ time to execute for each i, j. □

Note that to obtain the actual sequence of steps in the optimum complete block deletion sequence one can store appropriate pointers as the $t_{i,j}$ are computed.

4.3 Computing Block Deletion

The reader is reminded that for the block deletion problem one needs to find the minimum length sequence of block deletions to transform a permutation into a monotone increasing list. Next it is shown how one obtains a solution to the block deletion problem for x in $O(n^2)$-time given that all $t_{i,j}$ are known for $1 \leq i \leq j \leq n$. Define a weighted acyclic directed graph G with one node for each $i \in \{0,\ldots,n+1\}$. There is an edge from i to j if and only if $i < j$ and $x_i < x_j$, and the weight of that edge is $t_{i+1,j-1}$. Simply observe that there is a path $\langle 0, i_1, i_2, \ldots, i_k, n+1\rangle$ in G exactly when $x_{i_1}, \ldots x_{i_k}$ is a monotone increasing list. Furthermore the weight of the edge $\langle i_\ell, i_{\ell+1}\rangle$ gives the minimum number of block deletions necessary to delete elements between position i_ℓ and $i_{\ell+1}$ in x. Thus if there is a block deletion sequence of x of length m, there must be a path from 0 to $n+1$ in G of weight m, and vice-versa.

Using standard dynamic programming, a minimum weight path from 0 to $n+1$ can be found in $O(n^2)$ time. Let $0 = i_0 < i_1 < \cdots < i_\ell = n+1$ be such a minimum weight path, and let $w = \sum_{u=1}^{\ell} t_{i_{u-1}+1,i_u-1}$ be the weight of that minimum path. Since every deletion is a block deletion, the entire list can be deleted to a monotone list in w block deletions. Thus it follows:

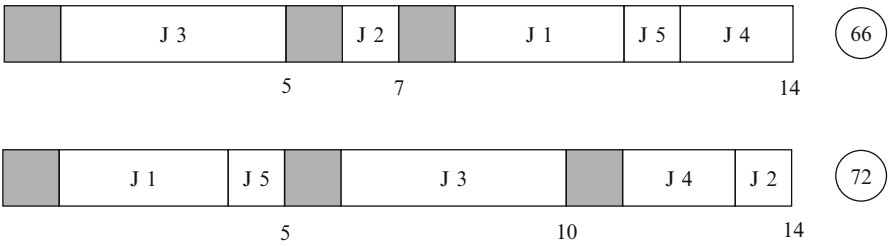


Fig. 10 A batching example. Shown are two feasible schedules for a 5-job problem where processing times are $p_1 = 3, p_2 = 1, p_3 = 4, p_4 = 2, p_5 = 1$ and the weights are $w_1 = w_4 = w_5 = 1$ and $w_2 = w_3 = 2$. The encircled values give the sum of weighted completion times of the depicted schedules

Theorem 8 *The block deletion problem can be solved in time* $O(n^2)$.

5 Total Monotonicity and Batch Scheduling

Although dynamic programming has been around for decades there have been more recent techniques for making dynamic programming more efficient. This section describes an example from scheduling, where such a technique is applied. Before the technique is shown, the simple traditional dynamic programming for the scheduling problem is given and then the dynamic programming speedup technique is described.

5.1 The Problem $1|\text{s} - \text{batch}|\sum w_i C_i$

Consider the *batching problem* where a set of jobs $\mathcal{J} = \{J_i\}$ with processing times $p_i > 0$ and weights $w_i \geq 0$, $i = 1, \ldots, n$, must be scheduled on a single machine, and where $\mathcal{J}$ must be partitioned into *batches* $\mathcal{B}_1, \ldots, \mathcal{B}_r$. All jobs in the same batch are run jointly and each job's completion time is defined to be the completion time of its batch. One assumes that when a batch is scheduled it requires a setup time $s = 1$. The goal is to find a schedule that minimizes the *sum of completion times* $\sum w_i C_i$, where C_i denotes the completion time of J_i in a given schedule. Given a sequence of jobs, a batching algorithm must assign every job J_i to a batch. More formally, a feasible solution is an assignment of each job J_i to the m_i^{th} batch, $i \in \{1, \ldots, n\}$ (Fig. 10).

The problem considered has the jobs executed sequentially, thus the problem is more precisely referred to as the *s-batch* problem. There is a different version of the problem not studied here, where the jobs of a batch are executed in parallel, known as the *p-batch* problem. In that case, the length of a batch is the maximum of the processing times of its jobs. The s-batch is also denoted in $\alpha|\beta|\gamma$ notation as the $1|\text{s-batch}|\sum w_i C_i$ problem. Brucker and Albers [6] showed that the

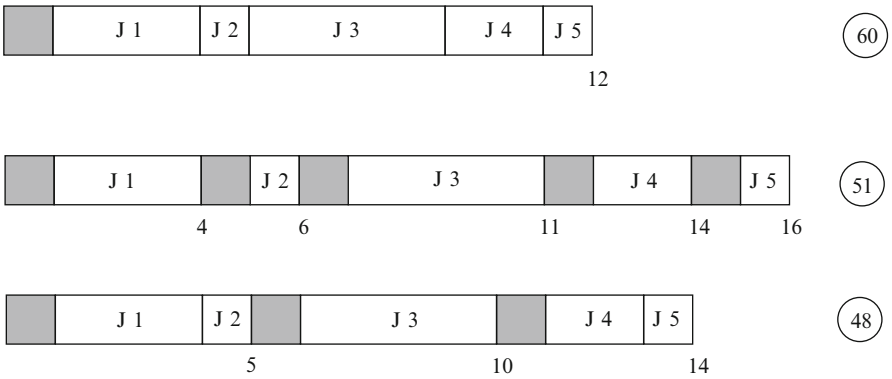


Fig. 11 List batching. Shown are three schedules for a 5-job problem where all weights are 1 and the processing requirements are $p_1 = 3$, $p_2 = 1$, $p_3 = 4$, $p_4 = 2$, $p_1 = 1$. The encircled values give the sum of weighted completion times of the depicted schedules

$1|\text{s-batch}|\sum w_i C_i$ problem is $\mathcal{NP}$-hard in the strong sense by giving a reduction from 3-PARTITION.

There is a large body of work on dynamic programming and batching; see the work of Baptiste [12], Baptiste and Jouglet [13], Brucker, Gladky, Hoogeveen, Kovalyov, Potts Tautenhahn, and van de Velde [36], Brucker, Kovalyov, Shafransky, and Werner [37], and Hoogeveen and Vestjens [64], as well as the scheduling text book by Peter Brucker [33]. Batching has wide application in manufacturing (see e.g., [34, 85, 98]), decision management (see, e.g., [74]), and scheduling in information technology (see e.g., [46]). More recent work on online batching is related to the TCP (Transmission Control Protocol) acknowledgment problem (see [20, 49, 67]).

5.2 List Batching

A much easier version of the problem is the *list* version of the problem where the order of the jobs is given, i.e., $m_i \leq m_j$ if $i < j$. An example is given in Fig. 11.

Assume that the jobs are $1, \ldots, n$ and are given in this order. One can then reduce the list batching problem to a shortest path problem in the following manner: Construct a weighted directed acyclic graph G with nodes $i = 1, \ldots, n$ (i.e., one node for each job) and add a dummy node 0. There is an edge (i, j) if and only if $i < j$. (See Fig. 12 for a schematic.) Let edge costs $c_{i,j}$ for $i < j$ be defined as

$$c_{i,j} = \left(\sum_{\ell=i+1}^{n} w_\ell \right) \left(s + \sum_{\ell=1}^{j} p_\ell \right). \tag{16}$$

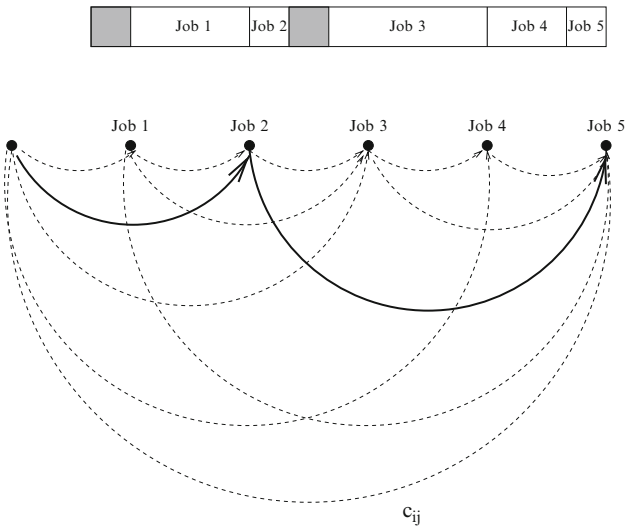


Fig. 12 Reduction of the list batching problem to a path problem

It is easily seen (see [6] for details) that the cost of path $< 0, i_1, i_2, \ldots, i_k, n >$ gives the $\sum C_i w_i$ value of the schedule which batches at each job $i_1, i_2, \ldots, i_k$. Conversely, any batching with cost A corresponds to a path in G with with path length A.

A shortest path can be computed in time $O(n^2)$ using the following dynamic program:
Let

$$E[\ell] = \text{ cost of the shortest path from 0 to } \ell,$$

then

$$E[\ell] = \min_{0 \le k < \ell} \{E[k] + c_{k,l}\} \text{ with } E[0] = 0, \tag{17}$$

which results in a table, in which elements can be computed row by row (see Fig. 13).

In other words, the dynamic program computes the row minima of the $n \times n$ matrix E, where

$$E[\ell, k] = \begin{cases} E[k] + c[k, \ell] \text{ if } \ell < k \\ \infty \text{ else} \end{cases} \tag{18}$$

with $\ell = 1, \ldots n$ and $k = 0, \ldots, n-1$.

As it turns out it is not necessary to calculate all entries of E to calculate the optimal solution. Surprisingly, only $O(n)$ have to be looked up throughout the entire calculation. The reason that this is possible is that E is a matrix with special properties discussed next.

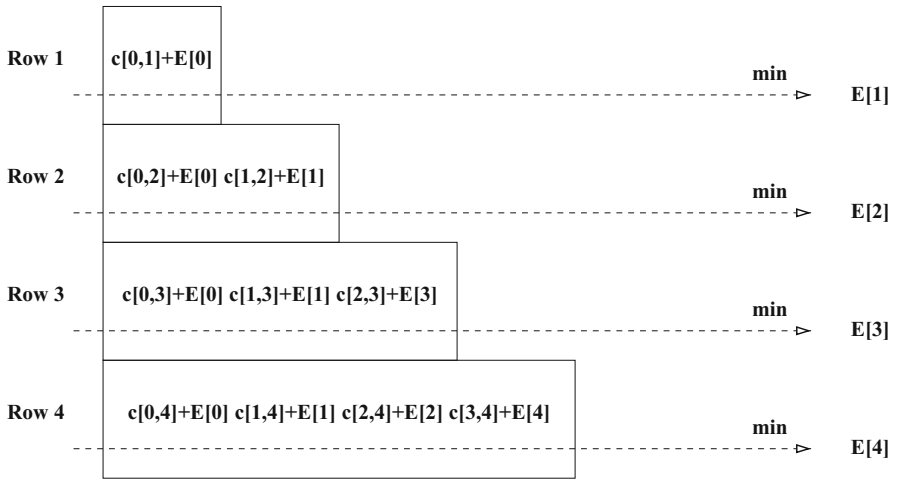


Fig. 13 Dynamic programming tableau

5.3 The Monge Property and Total Monotonicity

Definition 1 A matrix A is Monge if for all $i < i'$ and $j < j'$,

$$A[i, j] + A[i', j'] \leq A[i', j] + A[i, j']. \tag{19}$$

Definition 2 A 2×2 matrix is monotone if the rightmost minimum of the upper row is not to the right of the rightmost minimum of the lower row. More formally, $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is monotone if $b \leq a$ implies that $d \leq c$.

Definition 3 A matrix A is called totally monotone if all 2×2 dimensional submatrices are monotone.

Observation 1 *Every Monge matrix is totally monotone.*

The reader is referred to Fig. 14. Monge matrices occur routinely. For example in the batching to shortest path reduction, the cost matrix C is a Monge matrix:

Lemma 4 *The matrix $C = (c_{i,j})$ defined in (16) is Monge for all choices of $p_i, w_i \geq 0$. Furthermore values can be queried in $O(1)$ time after linear preprocessing.*

Proof Let $W_i = \sum_{v=1}^{i} w_v$ and $P_i = \sum_{v=1}^{i} p_v$ be the partial sum of the p_i and w_i values. Then

$$c[i, j] = c_{i,j} = (W_n - W_i)(s + P_j - P_i)$$

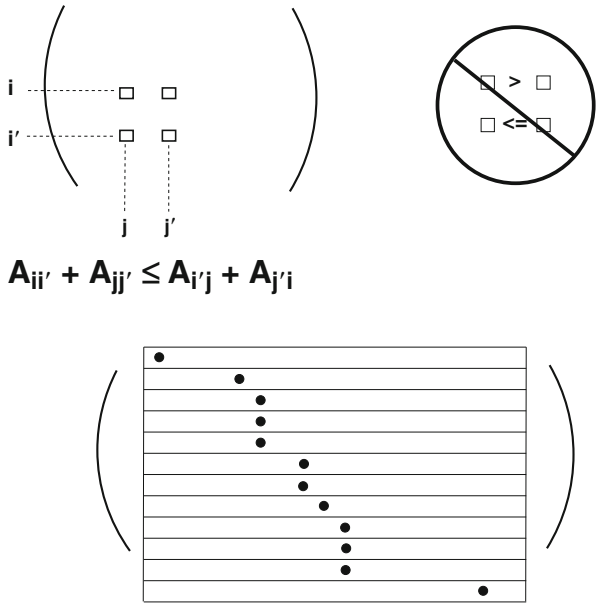


Fig. 14 Monge and monotonicity. *Top left* the Monge property is shown, which prohibits the situation *top right*. Thus row minima veer to the right (*bottom* figure)

For $i < i'$ and $j < j'$

$$c[i, j] + c[i', j'] - c[i', j] - c[i, j'] \tag{20}$$
$$= (P_{j'} - P_j)(W_{i'} - W_i) \tag{21}$$
$$\geq 0. \tag{22}$$

Also, notice that these values can be queried in $O(1)$ time after linear preprocessing by setting up arrays of partial sums for W_i and P_i in linear time. □

Furthermore, the matrix E is also a Monge matrix:

Lemma 5 *The matrix $E = (E_{\ell,k})$ defined in (18) is Monge.*

Proof Monge is preserved under addition and taking the minimum. □

Computing the row minima of a Monge (or totally monotone) matrix can be done trivially in $O(n \log n)$, cf. Fig. 15. A complex recursive procedure by Agarwal, Klawe, Moran, Shor, and Wilber [3] known as the SMAWK algorithm (the name was derived using the initials of authors of the original paper in which the algorithm was introduced), which has linear run time.

Note that the dynamic program of Fig. 13 is essentially used to calculate the row minima of E – a Monge matrix. However, the trivial $O(n \log n)$ cannot be used here since that algorithm requires all elements of E to be available offline,

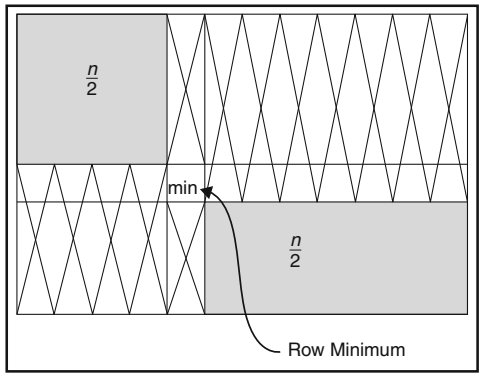


Fig. 15 Calculating all row minima of a Monge matrix. The algorithm finds the minimum of the middle row in linear time and then recurses on the upper left and lower right parts of the matrix

i.e., before the computation begins. Instead, there is a protocol by which each element can be queried: Once the minimum of row 1 is known, then any element in column 1 can be generated in constant time; once the minimum of row 2 is known then any element of row 1 and 2 are "knowable"; and so forth. (See also Fig. 16.)

More formally the protocol is as follows:

1. For each row index ℓ of E, there is a column index γ_ℓ such that for $k > \gamma_\ell$, $E_{\ell,k} = \infty$. Furthermore, $\gamma_\ell \leq \gamma_{\ell+1}$.
2. If $k \leq \gamma_\ell$, then $E[\ell, k]$ can be evaluated in $O(1)$ time *provided that the row minima of the first* ℓ *rows are already known*.
3. E is a totally monotone matrix.

Larmore and Schieber [77] have developed an algorithm that generalizes the SMAWK to run in linear time in this case as well. Their algorithm is also known as the LARSCH algorithm – again, as with the SMAWK algorithm, the name was derived using initials of authors of the original paper in which the algorithm was introduced in Larmore and Schieber [77]. Both SMAWK and LARSCH are important in dynamic programming speedup and are explained in the next section.

6 The SMAWK and LARSCH Algorithm

This section gives "ready to implement" descriptions of the SMAWK and LARSCH algorithms mentioned in the previous section.

6.1 The Matrix Searching Problem

The *matrix searching problem* is the problem of finding all row minima of a given matrix M. If n, m are the number of rows columns, respectively, of M, the problem clearly takes $\Theta(nm)$ time in the worst case. However, if M is *totally monotone* the problem can be solved in $O(n + m)$ time using the SMAWK algorithm [3].

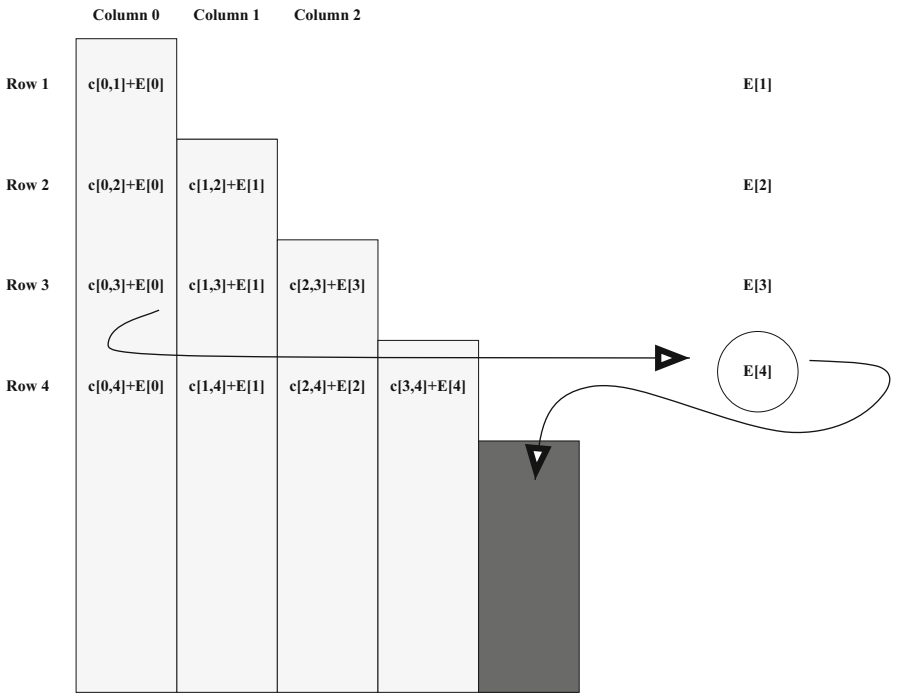


Fig. 16 The "online" protocol of the tableau. Note that once the minimum of row 4 is known, column 4 is "knowable"

SMAWK is recursive, but uses two kinds of recursion, called INTERPOLATE and REDUCE, which are described in the following. Let $1 \leq J(i) \leq m$ be the index of the minimum element of the i^{th} row.[1] Since M is totally monotone, $J(i) \leq J(k)$ for any $i < k$.

1. For small cases, SMAWK uses a trivial algorithm. If $m = 1$, the problem is trivial. If $n = 1$, simply use linear search.
2. If $n \geq 2$, then INTERPOLATE can be used. Simply let M' be the $\lfloor n/2 \rfloor \times m$ submatrix of M consisting of all even indexed rows of M. Recursively, find all row minima of M', and then use linear search to find the remaining minima of M in $O(n + m)$ time.
3. If $m > n$, then REDUCE can be used. The set $\{J(i)\}$ clearly has cardinality at most n. REDUCE selects a subset of the columns of M which has cardinality at most n, and which includes Column $J(i)$ for all $1 \leq i \leq n$. Let M' be the submatrix of M consisting of all the columns selected by REDUCE. Then, recursively, find all row minima of M'. These will exactly be the row minima of M. The time required to select the columns of M' is $O(n + m)$.

[1] Use the leftmost rule to break ties.

SMAWK then operates by alternating the two kinds of recursion. If the initial matrix is $n \times n$, then INTERPOLATE reduces to the problem on a matrix of size roughly $n/2 \times n$, and then REDUCE reduces to the problem on a matrix of size roughly $n/2 \times n/2$, and so forth.

Time Complexity

Let $T(n)$ be the time required by SMAWK to find all row minima of an $n \times n$ totally monotone matrix. Applying both INTERPOLATE and REDUCE, obtain the recurrence

$$T(n) = O(n) + T(n/2)$$

and thus $T(n) = O(n)$. By a slight generalization of the recurrence, one can show that SMAWK solves the problem for an $n \times m$ matrix in $O(n + m)$ time.

INTERPOLATE is explained using the code below.

Code for INTERPOLATE

```
{M is an n × m totally monotone matrix, where m ≤ n.}
if n = 1 then
    J(1) = 1
else
    Let M′ be the matrix consisting of the even indexed rows of M.
    Obtain J(i) for all even i by a recursive call to REDUCE on M′.
    for all odd i in the range 1 to n do
        if i = n then
            Find J(n) ∈ [J(n − 1), n] by linear search.
        else
            Find J(i) ∈ [J(i − 1), J(i + 1] by linear search.
        end if
    end for
end if
```

The total number of comparisons needed to find the minima of all odd rows is at most $n - 1$, as illustrated in Fig. 17 below.

Now for REDUCE. The procedure REDUCE operates by maintaining a stack, where each item on the stack is a column (actually, a column index). Initially the stack is empty, and columns are pushed onto the stack one at a time, starting with Column 1. The capacity of the stack is n, the number of rows.

But before any given column is pushed onto the stack, any number (zero or more) of earlier columns are popped off the stack. A column is popped if it is determined that it is *dominated*, which implies that none of its entries can be the minimum of any row. In some cases, if the stack is full, a new column will not be pushed.

At the end, those columns which remain on the stack form the matrix M'.

Dominance

There is a loop invariant, namely that if the stack size is at least i, and if $S[i] = j$, then all entries of Column j above Row i are known to be useless, i.e., will

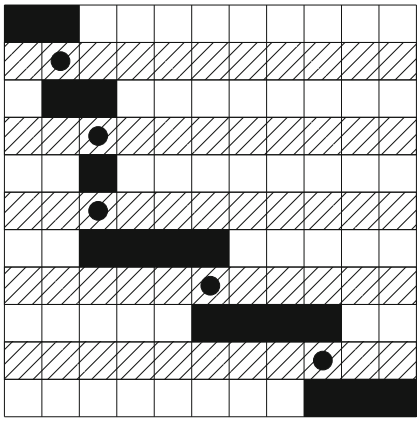


Fig. 17 INTERPOLATE. M' consists of the hatched rows, and the minima of those rows are indicated by *black dots*. By the monotonicity property of J, only search the interval indicated in *black* to find the minimum of any odd numbered row

not be the minima of any row. Formally, $M[i', S[i-1]] \leq M[i', j]$ for all $1 \leq i' < i$.

Suppose that Column k is the next column to be possibly pushed onto the stack. If the stack is empty, then $S[1] \leftarrow k$, and one is done. Otherwise, it must be checked to see whether the top column in the stack is dominated. Let i be the current size of the stack, and let $S[i] = k$. Then, necessarily, $i \leq j < k$. Column j is dominated if $M[i,k] < M[i,j]$. The reason is that $M[i', j] \geq M[i', S[i-1]]$ for any $i' < i$ by the loop invariant, and that $M[i', j] < M[i', k]$ for all $i' \geq i$ by monotonicity.

If the stack is popped, then the new top is tested for being dominated. This continues until the test fails. If the stack size is currently less than n, k is pushed onto the stack; otherwise, Column k is useless and is discarded.

Code for REDUCE

1: {M is an $n \times m$ totally monotone matrix.}
2: $top \leftarrow 0$ {Initialize the stack to empty}
3: **for** j from 1 to m **do**
4: **while** $top > 0$ and $M[top, j] < M[top, S[top]]$ **do**
5: $top \leftarrow top - 1$ {Pop the stack.}
6: **end while**
7: **if** $top < n$ **then**
8: $top \leftarrow top + 1$ and then $S[top] \leftarrow j$ {Push j onto the stack.}
9: **end if**
10: **end for**
11: Call INTERPOLATE on M', consisting of columns remaining on the stack.

The total number of comparisons needed to execute REDUCE (other than the recursive call) is at most $2m$. One see this using an amortization argument. Each column is given two credits initially. When a column is pushed onto the stack, it spends one credit, and when it is popped off the stack it spends one credit. Each of those two events requires at most one comparison. Thus, the total number of comparisons is at most $2m$.

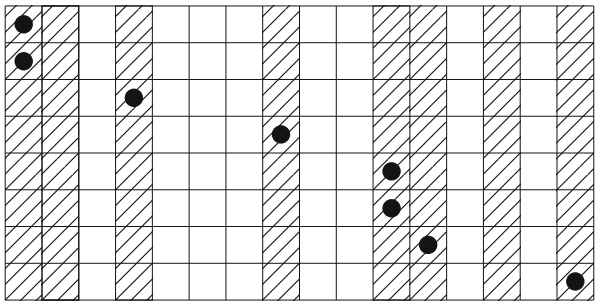


Fig. 18 REDUCE. M' consists of the hatched columns. All row minima, indicated by *black dots*, lie within those columns, although possibly not all columns contain minima

Figure 18 shows an example of an 8×16 matrix M, where the shaded columns survive to form M'.

6.2 The Online Matrix Searching Problem

Define an *almost lower triangular matrix* to be matrix M of n rows, numbered $1 \ldots n$, and m columns such that the length of the rows increases as n increases. More formally, let the columns be numbered $0 \ldots m-1$. Then there must exist n row lengths, $1 \leq \ell[1] \leq \ell[2] \leq \cdots \leq \ell[n] = m$, such that $M[i, j]$ is defined if and only if $1 \leq i \leq n$ and $0 \leq j < \ell[i]$. If $\ell[i] = i$ for all i, then M is a standard lower triangular matrix.

Given an almost lower triangular matrix, let $J(i)$ be the column index of the minimum entry in the i^{th} row of M. (Ties are broken arbitrarily). The *online matrix search problem* is the problem of finding all row minima of M, with the condition that a given entry $M[i, j]$ is available if and only if $J(k)$ has been computed for every k such that $\ell[k] < j$. For example, if M is a standard lower triangular matrix, $M[i, j]$ is available if and only if $J[k]$ has been computed for all $k \leq j$. Note that $M[i, 0]$ is available initially.

Assume that the computation is done by a process B which is in communication with a supervisor A. The online computation then proceeds as follows.

- A grants permission for B to see Column 0.
- B reports $J[1]$ (which is certainly 0) to A.
- A grants permission for B to see Column 1.
- B reports $J[2]$ to A.
- A grants permission for B to see Column 1.
- B reports $J[3]$ to A.
- etc.

Note that B is unable to compute $J(i)$ until it has seen Columns 0 through $i-1$, since the minimum of Row i could be in any of those columns. Conversely, one must have computed $J(1)$ through $J(i-1)$ in order to be allowed to see those columns. Thus, the order of the events in the above computation is strict.

6.3 Algorithm LARSCH

In general, it takes $O(nm)$ time to solve the online matrix search problem. However, if M is totally monotone, the values of J are monotone increasing, and the online matrix search problem can be solved in $O(n+m)$ time by using an online algorithm LARSCH, which is the online version of SMAWK, with some adaptations.

Let M be the input matrix. LARSCH works using a chain of processes, each of which is in communication with its *supervisor*, the process above it in the chain. The supervisor of the top process is presumably an application program. Refer to the process below a process as its *child*.

Each process in the chain works an instance of the online matrix searching problem. The top process works the instance given by the application. Each process reports results interleaved with messages received from its supervisor.

Each process P works the problem on a strictly monotone almost lower triangular matrix M_P, which is a submatrix of M. If Q is the child of P, then P creates a submatrix M_Q of M_P and passes that submatrix to Q. Of course, P never passes the entries of the matrix to Q; rather, it tells Q which entries of M it is allowed to see. For clarity assume that each process uses its own local row and column indices, but is aware of the global indices of each entry. For example, in example calculation in Table 5, $M_5[3,2] = M[15,11]$.

Alternating Types.
In the following detailed description of LARSCH, assume that the initial matrix M is standard lower triangular. (The algorithm can easily be generalized to cover other cases.)

Let $P_0, P_1, \ldots P_h$, $h = 2(\lfloor \log_2(n+1) \rfloor - 1)$, be the processes, where P_{t+1} is the child of P_t. The processes are of alternating *types*. If t is even, say that P_t has *standard* type, while if t is odd, say that P_t has *stretched* type.

Let M_t be the submatrix of M visible to P_t. $M_0 = M$. If t is odd, then M_t has $n_t = \left\lfloor 2^{-t/2}\left(n - 2^{t/2} - 1\right)\right\rfloor$ rows and $m_t = 2n_t$ columns, and the length of its i^{th} row is $2i$. If $t \geq 2$ is even, the M_t has $n_t = n_{t-1}$ rows and $m_t = n_t$ columns, and the length of its i^{th} row is i.

The choice of M_{t+1} as a submatrix of M_t is illustrated below. If t is even, then the rows of M_{t+1} are rows $3, 5, 7, \ldots \left\lfloor \frac{n-1}{2} \right\rfloor$ of M_t, ie. all rows of odd index other than 1. Row i of M_{t+1} consists of all but the last entry of Row $2i+1$ of M_t.

If t is odd, then M_{t+1} has the same number of rows as M_t, but only half the columns. Since there are n_t rows in M_t, the number of possible columns of M_t which contain values of J cannot exceed n_t. The REDUCE procedure of LARSCH chooses exactly n_t columns of M_t in a way that makes certain that all columns which contain row minima of M_t are chosen. The figure below illustrates one possible choice (Fig. 19). The shaded entries are passed to M_{t+1}. Note that not every entry of a selected column is part of M_{t+1}.

Table 5 LARSCH algorithm example: INTERPOLATE and REDUCE

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	32																		
2	31	48																	
3	10	15	21																
4	5	9	14	16															
5	19	23	27	27	40														
6	35	39	42	42	52	60													
7	27	31	33	33	41	47	51												
8	22	26	27	26	34	39	41	39											
9	33	37	37	36	44	48	47	44	54										
10	18	22	21	20	28	31	30	27	36	41									
11	41	45	43	42	50	52	51	47	55	57	46								
12	30	33	30	28	35	37	36	32	40	42	34	34							
13	48	50	46	43	49	51	50	46	53	55	44	42	47						
14	60	62	58	55	61	62	61	54	60	62	53	49	53	48					
15	54	56	52	48	53	53	52	46	52	53	42	38	42	37	41				
16	40	41	36	32	37	36	34	28	33	34	23	19	22	14	18	19			
17	69	69	64	60	64	62	59	53	57	58	47	43	45	39	43	38	35		
18	90	89	84	79	81	77	83	76	73	72	61	57	59	53	56	51	45	48	
19	78	77	71	66	68	64	60	52	54	53	42	38	39	33	35	30	22	24	18
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
3	10	15																	
5	19	23	27	27															
7	27	31	33	33	41	47													
9	33	37	37	36	44	48	47	44											
11	41	45	43	42	50	52	51	47	55	57									
13	48	50	46	43	49	51	50	46	53	55	44	42							
15	54	56	52	48	53	53	52	46	52	53	42	38	42	37					
17	69	69	64	60	64	62	59	53	57	58	47	43	45	39	43	38			
19	78	77	71	66	68	64	60	52	54	53	42	38	39	33	35	30	22	24	
	0	1	2	3	7	11	13	15	16										
3	10																		
5	19	23																	
7	27	31	33																
9	33	37	37	36															
11	41	45	43	42	47														
13	48	50	46	43	46	42													
15	54	56	52	48	46	38	37												
17	69	69	64	60	53	43	39	38											
19	78	77	71	66	52	38	33	30	22										

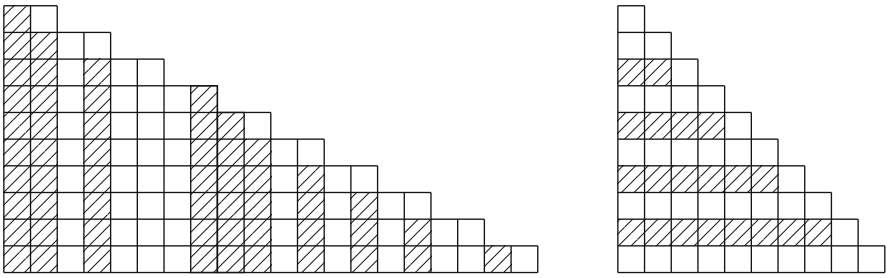


Fig. 19 Shaded entries of M_t indicate the submatrix M_{t+1} passed to the child process. *Left*: odd t. *Right*: even t

6.4 Standard Type Process: P_t for t Even (INTERPOLATE)

Code for a Process of Standard Type: P_t for t Even.

```
1:  {Matrix M_t with n_t rows and n_t columns}
2:  for i from 1 to n_t do
3:      Receive permission to view Column i − 1 from supervisor.
4:      if i = 1 then
5:          J(1) ← 0
6:      else if i is even and i < n_t then
7:          Grant permission to child to view Columns i − 2 and i − 1.
8:          Receive J_Partial from child.
9:              {Min of all entries but last of Row i + 1 is in Column J_Partial.}
10:         Find J(i) ∈ [J(i − 1), J_Partial] by linear search.
11:     else if i is even and i = n_t then
12:         Find J(n_t) ∈ [J(i − 1), n_t − 1] by linear search.
13:     else[i ≥ 3 is odd]
14:         if M_t[i, i − 1] < M_t[i, J_Partial] then
15:             J(i) ← i − 1
16:         else
17:             J(i) ← J_Partial
18:         end if
19:     end if
20:     Send J(i) to supervisor.   {Min of Row i is in Column J(i)}
21: end for
```

6.5 Standard Type Process: P_t for t Odd (REDUCE)

A stack called the *column stack* is maintained. Let *top* be the number of items (which are column indices) stored in the stack, and let $S[i]$ be the ith entry (counting from the bottom) of the stack. That is, $S[i]$ is defined for all $1 \leq i \leq top$.

Each time process P_t is able to view a new column, if the column stack is not empty, it pops all columns which are *dominated* by the new column off the stack, and then pushes the new column. The domination rule is that Column $S[i]$ is dominated by the new column, say Column j, if $M_t[i, j] < M_t[i, S[i]]$. Of course, it is possible that $j \geq 2i$, in which case $M_t[i, j]$ is taken to be infinity.

Code for a Process of Stretched Type: P_t for t Odd.

```
1:  {Matrix M_t with n_t rows and 2n_t columns}
2:  top ← 0        {Initialize the column stack to empty}
3:  for i from 1 to n_t do
4:      Receive permission to view Columns 2i − 2 and 2i − 1 from supervisor.
5:      Pop all columns dominated by Column 2i − 2 from column stack.
6:      Push Column 2i − 2 onto column stack.
7:      Pop all columns dominated by Column 2i − 1 from column stack.
8:      Push Column 2i − 1 onto column stack.
9:      Grant permission to child to view Column S[i].
10:         {The i^th column on the stack, which will become Column i − 1 of M_{t+1}}
11:     Receive J(i) from child.        {Min of Row i is in Column J(i)}
12:     Send J(i) to supervisor.
13: end for
```

How may a new column possibly dominate existing columns on the column stack? Lines 2, 3, and 4 of the code below are the detail of Line 5 and Line 7 of the code above, while Lines 5 and 6 are the detail of Line 6 and Line 8 of the code above.

Code for Popping and Pushing the Column Stack.

```
1:  {Column j is the new column}
2:  while top > 0 and 2 · top > j and M_t[top, j] < M_t[top, S[top]] do
3:      top ← top − 1        {Pop off the dominated column}
4:  end while
5:  top ← top + 1
6:  S[top] ← j        {Push the new column}
```

7 The Quadrangle Inequality and Binary Search Trees

Another type of speedup is based in the Knuth-Yao quadrangle inequality. This section discusses this kind of speedup and the relation with SMAWK/LARSCH speedup.

7.1 Background

Recall construction of optimal binary search trees discussed in Sect. 2.5. Gilbert and Moore [59] gave a $O(n^3)$ time algorithm. More than a decade later, in 1971,

it was noticed by Knuth [71] that, using a complicated amortization argument, the $B_{i,j}$ can all be computed using only $O(n^2)$ time. Around another decade later, in the early 1980s, Yao (see [96, 97]) simplified Knuth's proof and, in the process, showed that this *dynamic programming speedup* worked for a large class of problems satisfying a *quadrangle inequality* property. Many other authors then used the Knuth-Yao technique, either implicitly or explicitly, to speed up different dynamic programming problems. For this see for example the work of Wessner [93], the work of Atallah, Kosaraju, Larmore, Miller, and Teng [9] and Bar-No and Ladner [14].

Even though both the SMAWK algorithm and the Knuth-Yao (KY) speedup (best described in [71, 96, 97]) use an implicit quadrangle inequality in their associated matrices, on second glance, they seem quite different from each other. In the SMAWK technique, the quadrangle inequality is on the entries of a given $m \times n$ *input* matrix, which can be any totally monotone matrix. The KY technique, by contrast, uses a quadrangle inequality in the upper-triangular $n \times n$ matrix B. That is, it uses the QI property of its *result* matrix to speed up the evaluation, via dynamic programming, of the entries in the same result matrix. Aggarwal and Park [2] demonstrated a relationship between the KY problem and totally-monotone matrices by building a *3-D monotone matrix* based on the KY problem and then using an algorithm due to Wilber [94] to find *tube* minima in that 3-D matrix. They left as an open question the possibility of using SMAWK directly to solve the KY problem.

Definition 4 A two dimensional upper triangular matrix $A, 0 \leq i \leq j \leq n$ satisfies the *quadrangle inequality (QI)* if for all $i \leq i' \leq j \leq j'$,

$$A(i, j) + A(i', j') \leq A(i', j) + A(i, j'). \tag{23}$$

Observation 2 *A Monge matrix satisfies a quadrangle inequality, but a totally monotone matrix may not.*

Yao's result (of [96]) was formulated as follows: For $0 \leq i \leq j \leq n$ let $w(i, j)$ be a given function and

$$B_{i,j} = \begin{cases} 0, & \text{if } i = j; \\ w(i, j) + \min\limits_{i<t\leq j} \left(B_{i,t-1} + B_{t,j}\right), & \text{if } i < j. \end{cases} \tag{24}$$

Definition 5 $w(i, j)$ is *monotone in the lattice of intervals* if $[i, j] \subseteq [i', j']$ implies $w(i, j) \leq w(i', j')$.

As an example, it is not difficult to see that the $w(i, j) = \sum_{l=i+1}^{j} p_l + \sum_{l=i}^{j} q_l$ of the BST recurrence (12) satisfies the quadrangle inequality and is monotone in the lattice of intervals.

Definition 6 Let

$$K_B(i,j) = \max\{t \,:\, w(i,j) + B_{i,t-1} + B_{t,j} = B_{i,j}\},$$

i.e., the largest index which achieves the minimum in (24).

Yao then proves two Lemmas (see Fig. 21 for an example):

Lemma 6 (Lemma 2.1 in [96]) *If $w(i,j)$ satisfies the quadrangle inequality as defined in Definition 4, and is also monotone on the lattice of intervals, then the $B_{i,j}$ defined in (24) also satisfy the quadrangle inequality.*

Lemma 7 (Lemma 2.2 in [96]) *If the function $B_{i,j}$ defined in (24) satisfies the quadrangle inequality then*

$$K_B(i,j) \le K_B(i,j+1) \le K_B(i+1,j+1) \qquad \forall\, i < j.$$

Lemma 6 proves that a QI in the $w(i,j)$ implies a QI in the $B_{i,j}$. Suppose then that one evaluates the values of the $B_{i,j}$ in the order $d = 1,2,\ldots,n$, where, for each fixed d, one evaluates all of $B_{i,i+d}$, $i = 0,1,\ldots,n-d$. Then Lemma 7 says that $B_{i,i+d}$ can be evaluated in time $O(K_B(i+1,i+d) - K_B(i,i+d-1))$. Note that

$$\sum_{i=0}^{n-d} (K_B(i+1,i+d) - K_B(i,i+d-1)) \le n,$$

and thus all entries for fixed d can be calculated in $O(n)$ time. Summing over all d, it follows that all $B_{i,j}$ can be obtained in $O(n^2)$ time.

As mentioned, Lemma 7 and the resultant $O(n^2)$ running time have long been viewed as unrelated to the SMAWK algorithm. While they seem somewhat similar (a QI leading to an order of magnitude speedup) they appeared not to be directly connected until very recently. In the next section it is shown how to solve the Knuth-Yao problem directly using decompositions into total monotone matrices.

7.2 Decomposition Techniques

Definition 7 For $1 \le d \le n$ define the $(n-d+1) \times (n+1)$ matrix D^d by

$$D^d_{i,t} = \begin{cases} w(i,i+d) + B_{i,t-1} + B_{t,i+d}, & \text{if } 0 \le i < t \le i+d \le n; \\ \infty & \text{otherwise.} \end{cases} \tag{25}$$

Figure 20 illustrates a first decomposition. Note that (24) immediately implies

$$B_{i,i+d} = \min_{0 \le t \le n} D^d_{i,t} \tag{26}$$

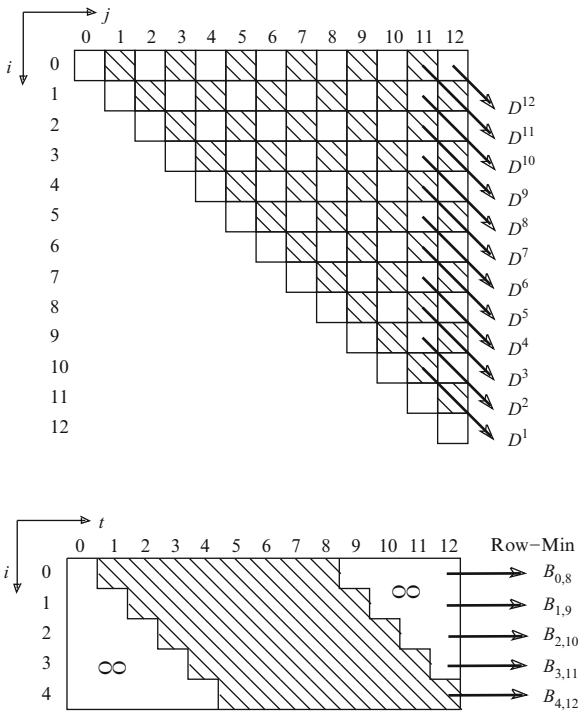


Fig. 20 The top figure on shows the $B_{i,j}$ matrix for $n = 12$. Each diagonal, $d = j - i$, in the matrix will correspond to a totally monotone matrix D^d. The minimal item of row i in D^d will be the value $B_{i,i+d}$. The other figure shows D^8

so finding the row-minima of D^d yields $B_{i,i+d}$, $i = 0, \ldots, n-d$. Put another way, the $B_{i,j}$ entries on diagonal $d = j - i$ are exactly the row-minima of matrix D^d.

Lemma 8 *If $w(i, j)$ and the function $B_{i,j}$ defined in (24) satisfies the QI then, for each d $(1 \leq d \leq n)$, D^d is a totally monotone matrix.*

Proof It suffices to prove that

$$D^d_{i,t} + D^d_{i+1,t+1} \leq D^d_{i+1,t} + D^d_{i,t+1} \tag{27}$$

Note that if $i + 1 < t < i + d$, then from Lemma 6,

$$B_{i,t-1} + B_{i+1,t} \leq B_{i+1,t-1} + B_{i,t} \tag{28}$$

and

$$B_{t,i+d} + B_{t+1,i+1+d} \leq B_{t+1,i+d} + B_{t,i+1+d}. \tag{29}$$

Thus,

$$\begin{aligned}
& D^d_{i,t} + D^d_{i+1,t+1} \\
= \; & [w(i, i+d) + B_{i,t-1} + B_{t,i+d}] + [w(i+1, i+1+d) + B_{i+1,t} + B_{t+1,i+1+d}] \\
= \; & w(i, i+d) + w(i+1, i+1+d) + [B_{i,t-1} + B_{i+1,t}] + [B_{t,i+d} + B_{t+1,i+1+d}] \\
\leq \; & w(i, i+d) + w(i+1, i+1+d) + [B_{i+1,t-1} + B_{i,t}] + [B_{t+1,i+d} + B_{t,i+1+d}] \\
= \; & [w(i+1, i+1+d) + B_{i+1,t-1} + B_{t,i+1+d}] + [w(i, i+d) + B_{i,t} + B_{t+1,i+d}] \\
= \; & D^d_{i+1,t} + D^d_{i,t+1}
\end{aligned}$$

and (27) is correct (where it is noted that the right hand side is ∞ if $i + 1 \not< t$ or $t \not< i + d$). □

Lemma 9 *Assuming that all of the row-minima of $D^1, D^2, \ldots, D^{d-1}$ have already been calculated, all of the row-minima of D^d can be calculated using the SMAWK algorithm in $O(n)$ time.*

Proof From the previous lemma, D^d is a totally monotone matrix. Also, by definition, its entries can be calculated in $O(1)$ time, using the previously calculated row-minima of $D^{d'}$ where $d' < d$. Thus SMAWK can be applied. □

Combined with (26) this immediately gives a new $O(n^2)$ algorithm for solving the KY problem; just run SMAWK on the D^d in the order $d = 1, 2, \ldots, n$ and report all of the row-minima.

7.3 Online Decomposition

The online problem restricted to the optimal binary search tree would be to construct the OBST for items $\text{Key}_{L+1}, \ldots, \text{Key}_R$, and, at each step, add either Key_{R+1}, a new key to the right, or Key_L, a new key to the left. Every time a new element is added, it is desired to update the $B_{i,j}$ (dynamic programming) table and thereby construct the optimal binary search tree of the new full set of elements. (See Fig. 21.)

KY speedup *cannot* be used to do this. The reason that the speedup fails is that the KY speedup is actually an amortization over the evaluation of all entries when done in a particular order. In the online case, adding a new item n to previously existing items $1, 2, \ldots, n-1$ requires using (24) to compute the n new entries $B_{i,n}$, in the fixed order $i = n, n-1, \ldots, 1, 0$ and it is not difficult to construct an example in which calculating these new entries in this order using (24) requires $O(n^2)$ work.

Neither can the decomposition technique from the previous section solve the online problem. To see why, suppose that items $1, \ldots, n$ have previously been given, new item $n+1$ has just been added, and one needs to calculate the values $B_{i,n+1}$ for $i = 0, \ldots, n+1$. In this formulation it would correspond to adding a new bottom row to *every* matrix D^d and creating a new matrix D^{n+1} and one would need to

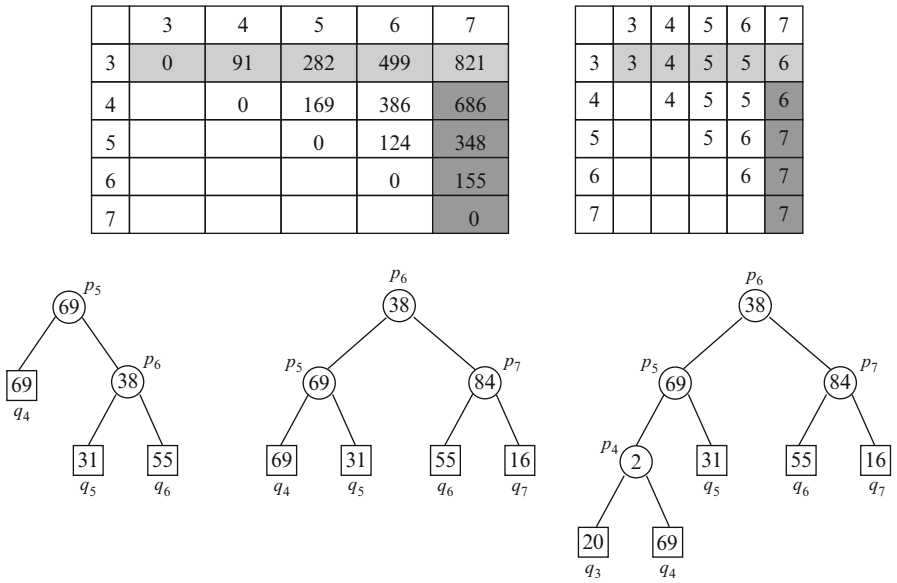


Fig. 21 An example of the online case for optimal binary search trees where $(p_4, p_5, p_6, p_7) = (2, 69, 38, 84)$ and $(q_3, q_4, q_5, q_6, q_7) = (20, 69, 31, 55, 16)$. The left table contains the $B_{i,j}$ values; the right one, the $K_B(i, j)$ values. The unshaded entries in the table are for the problem restricted to only keys 5, 6. The dark gray cells are the entries added to the table when key 7 is added to the right. The light gray cells are the entries added when key 4 is added to the left. The corresponding optimal binary search trees are also given, where circles correspond to successful searches and squares to unsuccessful ones. The values in the nodes are the weights of the nodes (not their keys)

find the row-minima of all of the n new bottom rows. Unfortunately, the SMAWK algorithm only works on the rows of matrices all at once and cannot help to find the row-minima of a single new row.

Note that for the online problem it is certainly possible to *recompute* the entire table; however this comes at the price of $O(n^2)$ time, where $n = R - L$ is the number of keys currently in the table, leading to a total running time of $O(n^3)$ to insert all of the keys. Of interest here is the question of whether one can maintain the speedup while inserting the keys in an online fashion. The goal is an algorithm in which a sequence of n online key insertions will result in a worst case $O(n)$ per step to maintain an optimal tree, yielding an overall run time of $O(n^2)$ (Fig. 22).

To this end now a second decomposition. It is indexed by the *leftmost* element seen so far. See Fig. 23.

Definition 8 For $0 \leq i < n$ define the $(n - i) \times (n - i)$ matrix L^i by

$$L^i_{j,t} = \begin{cases} w(i, j) + B_{i,t-1} + B_{t,j}, & \text{if } i < t \leq j \leq n; \\ \infty, & \text{otherwise.} \end{cases} \tag{30}$$

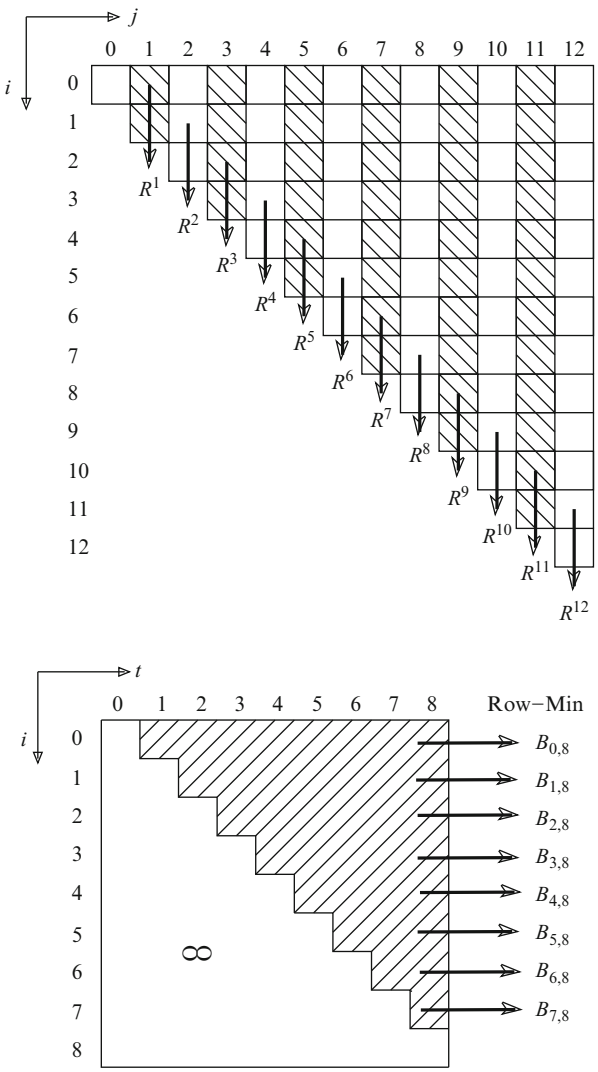


Fig. 22 The top figure shows the $B_{i,j}$ matrix for $n = 12$. Each column in the $B_{i,j}$ matrix will correspond to a totally monotone matrix R^j. The minimal element of row i in R^j will be the value $B_{i,j}$. The other figure shows R^8

(For convenience, set the row and column indices to run from $(i + 1) \ldots n$ and not $0 \ldots (n - i - 1)$.) Note that (24) immediately implies

$$B_{i,j} = \min_{i<t\leq n} L^i_{j,t} \tag{31}$$

so finding the row-minima of L^i yields $B_{i,j}$ for $j = i + 1, \ldots, n$. Put another way, the $B_{i,j}$ entries in row i are exactly the row minima of matrix L^i.

Lemma 10 *If the function defined in (24) satisfies the QI then R^j (resp. L^i) are totally monotone matrices for each fixed j (resp. i).*

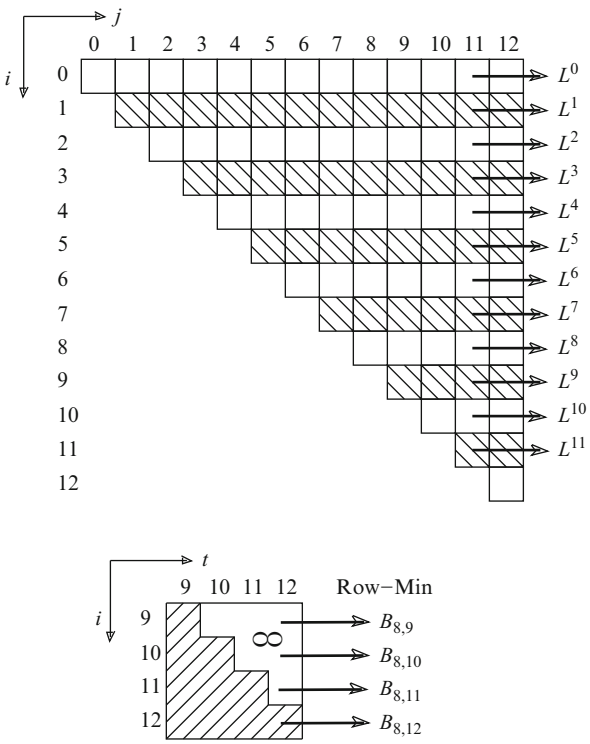


Fig. 23 The top figure on the left shows the $B_{i,j}$ matrix for $n = 12$. Each row in the $B_{i,j}$ matrix will correspond to a totally monotone matrix L^i. The minimal element of row j in L^i will be the value $B_{i,j}$. The other figure shows L^8

Proof The proofs are very similar to that of Lemma 8. To prove R^j is totally monotone, note that if $i + 1 < t < j$, one can again use (28); writing the entries from (28) in boldface gives

$$\begin{aligned} & R^j_{i,t} + R^j_{i+1,t+1} \\ = \ & \left[w(i,j) + \mathbf{B_{i,t-1}} + B_{t,j}\right] + \left[w(i+1,j) + \mathbf{B_{i+1,t}} + B_{t+1,j}\right] \\ \leq \ & \left[w(i+1,j) + \mathbf{B_{i+1,t-1}} + B_{t,j}\right] + \left[w(i,j) + \mathbf{B_{i,t}} + B_{t+1,j}\right] \\ = \ & R^j_{i+1,t} + R^j_{i,t+1} \end{aligned}$$

and thus R^j is Monge (where the right hand side is ∞ if $i + 1 \not< t$) and thus totally monotone. To prove L^i is totally monotone, if $i < t < j$ then again use (28) (with j replaced by $j + 1$) to get

$$\begin{aligned} & L^i_{j,t} + L^i_{j+1,t+1} \\ = \ & \left[w(i,j) + B_{i,t-1} + \mathbf{B_{t,j}}\right] + \left[w(i,j+1) + B_{i,t} + \mathbf{B_{t+1,j+1}}\right] \\ \leq \ & \left[w(i,j+1) + B_{i,t-1} + \mathbf{B_{t,j+1}}\right] + \left[w(i,j) + B_{i,t} + \mathbf{B_{t+1,j}}\right] \\ = \ & L^i_{j+1,t} + L^i_{j,t+1} \end{aligned}$$

and thus L^i is Monge (where the right hand side is ∞ if $t \not< j$) and thus totally monotone. □

Note that this decomposition immediately imply a new proof of Lemma 7 (Lemma 2.2 in [96]) which states that

$$K_B(i, j) \leq K_B(i, j+1) \leq K_B(i+1, j+1). \tag{32}$$

To see this note that $K_B(i, j+1)$ is the location of the rightmost row-minimum of row i in matrix R^{j+1}, while $K_B(i+1, j+1)$ is the location of the rightmost row-minimum of row $i+1$ in matrix R^{j+1}. Thus, the definition of total monotonicity immediately gives

$$K_B(i, j+1) \leq K_B(i+1, j+1). \tag{33}$$

Similarly, $K_B(i, j)$ is the rightmost row-minimum of row j in L^i while $K_B(i, j+1)$ is the location of the rightmost row-minimum of row $j+1$ in L^i. Thus

$$K_B(i, j) \leq K_B(i, j+1). \tag{34}$$

Combining (33) and (34) yields (32). Since the actual speedup in the KY technique comes from an amortization argument based on (32), it follows that the original KY-speedup itself is also a consequence of total monotonicity.

Up to this point it is not clear how to actually calculate the $B_{i,j}$ using the R^j and L^i. Note first that even though the R^j are totally monotone, their row minima *cannot* be calculated using the SMAWK algorithm. This is because, for $0 \leq i < t \leq j$, the value of entry $R^j_{i,t} = w(i, j) + B_{i,t-1} + B_{t,j}$, which is dependent upon $B_{t,j}$ which is itself the row-minimum of row t in the *same* matrix R^j. Thus, the values of the entries of R^j depend upon other entries in R^j which is something that SMAWK does not allow. The same problem occurs with the L^i.

But despite this dependence, the LARSCH algorithm can still be used to find the row-minima of the R^j. Recall that to execute the LARSCH algorithm one needs only that the matrix X satisfy the following conditions:

1. X is an $n \times m$ totally monotone matrix.
2. For each row index i of X, there is a column index C_i such that for $j > C_i$, $X_{i,j} = \infty$. Furthermore, $C_i \leq C_{i+1}$.
3. If $j \leq C_i$, then $X_{i,j}$ can be evaluated in $O(1)$ time *provided that the row minima of the first* $i-1$ *rows are already known.*

If these conditions are satisfied, the LARSCH algorithm then calculates all of the row minima of X in $O(n+m)$ time. This algorithm can now be used to derive

Lemma 11

- *Given that all values* $B_{i',j}$, $i < i' \leq j \leq n$ *have already been calculated, all of the row-minima of* L^i *can be calculated in* $O(n-i)$ *time.*
- *Given that all values* $B_{i,j'}$, $0 \leq i \leq j' < j$ *have already been calculated, all of the row-minima of* R^j *can be calculated in* $O(j)$ *time.*

Proof For the first part, it is easy to see that L^i satisfies the first two conditions required by the LARSCH algorithm with $C_j = j$. For the third condition, note that, for $i < t \leq j$, $L^i_{j,t} = w(i, j) + B_{i,t-1} + B_{t,j}$. The values $w(i, j)$ and $B_{t,j}$ are already known and can be retrieved in $O(1)$ time. $B_{i,t-1}$ is the minimum of row $t-1$ of L^i but, since we are assuming $t \leq j$, this means that $B_{i,t-1}$ is the minimum of an earlier row in L^i, and the third LARSCH condition is satisfied. Thus, all of the row-minima of the $(n-i) \times (n-i)$ matrix L^i can be calculated in $O(n-i)$ time.

For the second part set X to be the $(j+1) \times (j+1)$ matrix defined by $X_{i,t} = R^j_{j-i,j-t}$. Then X satisfies the first two LARSCH conditions with $C_i = i - 1$. For the third condition note that $X_{i,t} = R^j_{j-i,j-t} = w(j-i, j) + B_{j-i,j-t-1} + B_{j-t,j}$. The values $w(j-i, j)$ and $B_{j-i,j-t-1}$ are already known and can be calculated in $O(1)$ time. $B_{j-t,j}$ is the row minima of row t of X; but, since we are assuming $t \leq C_i = i - 1$ this means that $B_{j-t,j}$ is the row minima of an earlier row in X so the third LARSCH condition is satisfied. Thus, all of the row-minima of X and equivalently R^j can be calculated in $O(j)$ time. □

Note that Lemma 11 immediately solves the "right-online" and "left-online" problems. Given the new values $w(i, R+1)$ for $L \leq i \leq R+1$, simply find the row minima of R^{R+1} in time $O(R-L)$. Given the new values $w(L-1, j)$ for $L-1 \leq j \leq R$, simply find the row minima of L^{L-1}. Therefore it was just shown that *any* dynamic programming problem for which the KY speedup can statically improve run time from $O(n^3)$ to $O(n^2)$ time can be solved in an online fashion in $O(n)$ time per step. That is, online processing incurs no penalty compared to static processing. In particular, the optimum binary search tree can be maintained in $O(n)$ time per step as nodes are added to both its left and right.

At this point note that decompositions L^i could also be derived by careful cutting of the 3-D monotone matrices of Aggarwal and Park [2] along particular planes. Aggarwal and Park [2] used an algorithm of Wilber [94] (derived for finding the maxima of certain concave-sequences) to find various tube maxima of their matrices, leading to another $O(n^2)$ algorithm for solving the KY-problem. In fact, even though their algorithm was presented as a static algorithm, careful decomposition of what they do permits using it to solve what is called here the left-online KY-problem. A symmetry argument could then yield a right-online algorithm. This never seems to have been noted in the literature, though.

8 Conclusion

Too small an academic community is aware of the many advanced tools available related to dynamic programming. Routinely, applications are solved by simple-minded dynamic programs, where much faster solutions are possible. In fact, for many massively large problems arising in Molecular Biology, for example, a quadratic solution might be completely useless, equivalent to no solution at all.

It is the hope of the author that this book chapter will open up some these advanced techniques to a larger community of scientists.

As well the use of dynamic programming in online optimization and by extension the concept of work functions is not as widely embraced as is desirable. Many online algorithms are ad-hoc and work functions make it possible to "de-adhocify" the construction of competitive and efficient algorithms in this setting. The very recent concept of *knowledge states* (see [23]), which has dynamic programming at its core, makes it possible to derive online algorithms even in the randomized case in a systematic way.

Readers are invited to go beyond the obvious when using dynamic programming and to avail themselves of these powerful techniques.

Further Reading

Introductions to dynamic programming with numerous elementary examples can be found in the textbooks by Brassard and Gilles [32], Cormen, Leiserson, Rivest, and Stein [45], Baase and van Gelder [10] and Dasgupta, Papadimitriou, and Vazarani [47].

The classical treatises by Bellmann [24] and [25] are still relevant today though the language is somewhat antiquated and the applications outdated. Other general reading is Dreyfus and Law [50], Stokey, Lucas, and Prescott [88], Bertsekas [28], Denardo [48], Meyn [83], and more recently Sniedovich [87]. One classical algorithm worth mentioning is the algorithm by Viterbi [90] for Hidden Markov Model inference. Many problems in areas such as digital communications can be cast in the Hidden Markov Model. Another classic is Hu and Tucker [65] on alphabetic trees. Bellmore and Nemhauser [27] as well as Lawler, Lenstra, Rinnoy Kan, and Shmoys [79] and Burkard, Deineko, Dal, van der Veen, and Woeginger [39] contain material regarding the use of dynamic programming for the traveling salesman problem. Gusfield [60] is good general resource on dynamic programming for computational biology. Examples with regards to scheduling can be found in Bellman, Esogbue, and Nabeshima [26] as well as in Brucker [33] and in Brucker and Knust [35]. David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano [52, 53] give a two paper sequence in the Journal of the ACM for sparse dynamic programming.

Regarding online algorithms (discussed in Sect. 3) the book by Borodin and El-Yanif [30] is the standard text. Of note is also the monograph by Karlin [66]. The article Larmore and Chrobak [42], which introduced work functions, extends the material in Sect. 3. For further immersion into the realm of work functions the article by Bein, Chrobak, and Larmore [16] is recommended. Recent work by Bein, Larmore, Noga, and Reischuk [23] on knowledge states has extended the use of work functions to randomized online algorithms: such algorithms are "guided" in their operation by work functions.

Some of the material presented in Sect. 4 can be found in greater detail in Bein, Larmore, Morales, and Sudborough [22]. Further reading regarding this section:

Sorting problems under various operations have been studied extensively, including work on sorting with prefix reversals Gates and Papadimitriou [58] (Gates of Microsoft fame) as well as Heydari and Sudborough [62], transpositions [11] and block moves ([17, 80] as well as Mahajan, Rama, and Vijayakumar [81]).

Some of the material presented in Sect. 5 can found in greater detail in Bein, Noga, and Wiegley [19]. The paper by Brucker and Albers [6] contains an alternate linear time algorithm for list batching. There is a large body of work on dynamic programming and batching; see the work of Baptiste [12], Baptiste and Jouglet [13], Brucker, Gladky, Hoogeveen, Kovalyov, Potts Tautenhahn, and van de Velde [36], Brucker, Kovalyov, Shafransky, and Werner [37], Hoogeveen, and Vestjens [64], as well as the scheduling text book by Peter Brucker [33]. Batching has wide application in manufacturing (see, e.g., [34, 85, 98]), decision management (see, e.g., [74]), and scheduling in information technology (see, e.g., [46]). More recent work on online batching is related to the TCP (Transmission Control Protocol) acknowledgment problem (see [20, 49, 67]).

Regarding Monge properties and total monotonicity the best resource is Park's thesis, [84]. Another excellent survey on Monge properties is Burkard, Klinz and Rudolf [38]. A generalization of the Monge property to an algebraic property is in Bein, Brucker, Larmore, and Park [18]. Interesting applications are in Woeginger [95]. Burkard, Deineko, and Woeginger [40] survey the traveling salesman problem on Monge matrices. Agarwal and Sen [1] give results for selection in monotone matrices for computing kth nearest neighbors. Schieber [86] gives results on k-link paths in graphs with Monge properties.

Section 6 is based on Agarwal, Klawe, Moran, Shor, and Wilber [3] and Larmore and Schieber [78]. Generalizations of the matrix searching techniques are in Klawe [69] and Klawe and Kleitman [70], as well as in Kravets and Park [73], Aggarwal and Park [5] (Parallel Searching). and Wilber [94]. A good survey paper is Galil and Park [57].

An extended version of the material of Sect. 7 can be found in Bein, Larmore, Golin, and Zhang [21]. For Sect. 7 the papers of Knuth [71] followed by Yao [97] are key. Aggarwal and Park used an algorithm of Wilber [94] to find various tube maxima of their matrices, leading to another $O(n^2)$ algorithm for solving the KY-problem. There are two extensions of the Knuth-Yao quadrangle inequality: the first is due to Wachs [91] and the second to Borchers and Gupta (BG) [29]. This is discussed in detail in Bein, Larmore, Golin, and Zhang [21]. Belatedly, Aggarwal, Bar-Noy, Khuller, Kravets, and Schieber [4] describe solutions for matching using the quadrangle inequality.

The main gist of this chapter is dynamic programming speedup: Applications abound. Classic examples include the work by Hirschberg and Larmore [63] on the weight subsequence problem, Larmore and Przytycka [76] on parallel construction of trees with optimal path length, and Larmore and Hirschberg [75] on length limited coding. David Eppstein [51] considers sequence comparisons. Apostolico, Atallah, Larmore, and McFaddin [7] give algorithms for string editing. Highly recommended is the paper by Galil and Giancarlo [56] on speeding up dynamic programming in Molecular Biology. Work by Arslan and Egecioglu [8] is on sequence alignment

Bradford, Golin, Larmore, and Rytter [31] give dynamic programs for optimal prefix-free codes. Recent speedup work is for the online maintenance of k-medians by Fleischer, Golin, and Zhang [54] (see also [61, 89]).

Acknowledgements This chapter is dedicated to Lawrence L. Larmore, a great mentor. A sabbatical (academic year 2006/07) granted by the University of Nevada, Las Vegas, which benefited this book chapter, is acknowledged.

Cross-References

▸Advances in Scheduling Problems
▸Computing Distances between Evolutionary Trees
▸Efficient Algorithms for Geometric Shortest Path Query Problems
▸Geometric Optimization in Wireless Networks
▸Online and Semi-online Scheduling
▸Resource Allocation Problems

Recommended Reading

1. P.K. Agarwal, S. Sen, Selection in monotone matrices and computing kth nearest neighbors. J. Algorithms **20**(3), 581–601 (1996); A preliminary version appeared, in *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory* (1994), pp. 13–24
2. A. Aggarwal, J.K. Park, Notes on searching in multidimensional monotone arrays, in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science* (IEEE Computer Society, Washington, DC, 1988), pp. 497–512
3. A. Aggarwal, M.M. Klawe, S. Moran, P.W. Shor, R.E. Wilber, 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* (1986), pp. 285–292
4. A. Aggarwal, A. Bar-Noy, S. Khuller, D. Kravets, B. Schieber, Efficient minimum cost matching and transportation using the quadrangle inequality. J. Algorithms **19**(1), 116–143 (1995); A preliminary version appeared, in *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science* (1992), pp. 583–592
5. A. Aggarwal, D. Kravets, J.K. Park, S. Sen, Parallel searching in generalized Monge arrays. Algorithmica **19**(3), 291–317 (1997); A preliminary version appeared, in *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures* (1990), pp. 259–268
6. S. Albers, P. Brucker, The complexity of one-machine batching problems. Discret. Appl. Math. **47**, 87–10 (1993)
7. A. Apostolico, M. Atallah, L. Larmore, S. McFaddin, Efficient parallel algorithms for string editing and related problems. SIAM J. Comput. **19**(5), 968–988 (1990)
8. A.N. Arslan, O. Egecioglu, Dynamic programming based approximation algorithms for sequence alignment with constraints. INFORMS J. Comput. **16**(4), 441–458 (2004)
9. M.J. Atallah, S. Rao Kosaraju, L.L. Larmore, G.L. Miller, S-H. Teng, Constructing trees in parallel, in *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures* (ACM, New York, 1989) , pp. 421–431
10. S. Baase, A. van Gelder, *Computer Algorithms* (Addison Wesley, Reading, 2000)
11. V. Bafna, P.A. Pevzner, Sorting by transposition. SIAM J. Discret. Math. **11**, 224–240 (1998)
12. P.Baptiste, Batching identical jobs. Math. Methods Oper. Res. **52**, 355–367 (2000)

13. P. Baptiste, A. Jouglet, On minimizing total tardiness in a serial batching problem. Oper. Res. **35**, 107–115 (2001)
14. A. Bar-Noy, R.E. Ladner, Efficient algorithms for optimal stream merging for media-on-demand. SIAM J. Comput. **33**(5), 1011–1034 (2004)
15. W. Bein, M. Chrobak, L.L. Larmore, The 3-server problem in the plane, in *Proceedings of 7th European Symposium on Algorithms (ESA)*. Volume 1643 of Lecture Notes in Computer Science (Springer, Berlin/New York, 1999), pp. 301–312
16. W. Bein, M. Chrobak, L.L. Larmore, The 3-server problem in the plane. Theoret. Comput. Sci. **287**, 387–391 (2002)
17. W. Bein, L.L. Larmore, S. Latifi, I. Hal Sudborough, Block sorting is hard. Int. J. Found. Comput. Sci. **14**(3), 425–437 (2003)
18. W. Bein, P. Brucker, L.L. Larmore, J.K. Park, The algebraic Monge property and path problems. Discret. Appl. Math. **145**(3), 455–464 (2005)
19. W. Bein, J. Noga, J. Wiegley, Approximation for batching via priorities. Sci. Ann. Comput. Sci. **XVII**, 1–18 (2007)
20. W. Bein, L. Epstein, L.L. Larmore, J. Noga, Optimally competitive list batching. Theoret. Comput. Sci. **410**(38–40), 3631–3639 (2009)
21. W. Bein, M. Golin, L. Larmore, Y. Zhang, The Knuth-Yao quadrangle-inequality speedup is a consequence of total-monotonicity. Trans. Algorithms **6**(1) (2009)
22. W. Bein, L.L. Larmore, L. Morales, I. Hal Sudborough, A quadratic time 2-approximation algorithm for block sorting. Theor. Comput. Sci. **410**, 711–717 (2009)
23. W. Bein, L.L. Larmore, J. Noga, R. Reischuk, Knowledge state algorithms. Algorithmica **60**(3), 653–678 (2011)
24. R. Bellman, The theory of dynamic programming. Bull. Am. Math. Soc. **60**:503–516 (1954)
25. R. Bellman, *Dynamic Programming*, Dover Paperback edition 2003 edn. (Princeton University Press, Princeton, 1957)
26. R. Bellman, A.O. Esogbue, I. Nabeshima, *Mathematical Aspects of Scheduling and Applications* (Pergamon, Oxford/New York, 1982)
27. M. Bellmore, G.L. Nemhauser, The traveling salesman problem: A survey. Oper. Res. **16**(3), 538–558 (1968)
28. D.P. Bertsekas, *Dynamic Programming and Optimal Control*, 2nd edn. (Athena Scientific, Belmont, 2000)
29. A. Borchers, P. Gupta, Extending the quadrangle inequality to speed-up dynamic programming. Inf. Process. Lett. **49**(6), 287–290 (1994)
30. A. Borodin, R. El-Yaniv, *Online Computation and Competitive Analysis* (Cambridge University Press, Cambridge/New York , 1998)
31. P.G. Bradford, M.J. Golin, L.L. Larmore, W. Rytter, Optimal prefix-free codes for unequal letter costs: Dynamic programming with the Monge property. J. Algorithms **42**(2), 277–303 (2002); A preliminary version appeared, in *Proceedings of the 6th Annual European Symposium on Algorithms* (1998), pp. 43–54
32. G. Brassard, P. Bratley, *Fundamentals of Algorithms* (Prentice Hall, Englewood, 1996)
33. P. Brucker, *Scheduling Algorithms* 5th edn. (Springer, Berlin/New York, 2007)
34. P. Brucker, J. Hurink, Solving a chemical batch scheduling problem by local search. Ann. Oper. Res. **96**, 17–38 (2000)
35. P. Brucker, S. Knust, Complex Scheduling (Springer, Berlin, 2006)
36. P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn, S. van de Velde, Scheduling a batch processing machine. J. Sched. **1**(1), 31–54 (1998)
37. P. Brucker, M. Kovalyov, Y. Shafransky, F. Werner, Batch scheduling with deadline on parallel machines. Ann. Oper. Res. **83**, 23–40 (1998)
38. R.E. Burkard, B. Klinz, R. Rudolf, Perspectives of Monge properties in optimization. Discret. Appl. Math. **70**(2), 95–161 (1996)
39. R.E. Burkard, V.G. Deineko, R. van Dal, J.A.A. van der Veen, G.J. Woeginger, Well-solvable special cases of the TSP: A survey. SIAM Rev. **40**(3), 496–546 (1998)

40. R.E. Burkard, V.G. Deineko, G.J. Woeginger, The travelling salesman problem on permuted Monge matrices. J. Comb. Optim. **2**(4), 333–350 (1999)
41. M. Chrobak, L.L. Larmore, An optimal online algorithm for k servers on trees. SIAM J. Comput. **20**, 144–148 (1991)
42. M. Chrobak, L.L. Larmore, The server problem and on-line games, in *On-line Algorithms*, ed. by L.A. McGeoch, D.D. Sleator. Volume 7 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science (AMS/ACM, Providence, RI, 1992), pp. 11–64
43. M. Chrobak, L.L. Larmore, Metrical task systems, the server problem, and the work function algorithm, in *Online Algorithms: The State of the Art*, ed. by A. Fiat, G.J. Woeginger (Springer, Berlin/New York, 1998), pp. 74–94
44. M. Chrobak, H. Karloff, T.H. Payne, S. Vishwanathan, New results on server problems. SIAM J. Discret. Math. **4**, 172–181 (1991)
45. T. Corman, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, 2nd edn. (McGraw Hill, New York, 2001)
46. A. Dan, D. Sitaram, P. Shahabuddin, Scheduling policies for an on-demand video server with batching, in *Proceedings of the second ACM international conference on Multimedia* (ACM, New York, 1994), pp. 15–23
47. S. Dasgupta, C. Papadimitriou, U. Vazirani, *Algorithms* (McGraw Hill, Boston, 2008)
48. E.V. Denardo, *Dynamic Programming: Models and Applications* (Dover, Mineola, 2003)
49. D.R. Dooly, S.A. Goldman, S.D. Scott, On-line analysis of the TCP acknowledgment delay problem. J. ACM **48**(2), 243–273 (2001)
50. S.E. Dreyfus, A.M. Law, *The art and theory of dynamic programming* (Academic, New York, 1977)
51. D. Eppstein, Sequence comparison with mixed convex and concave costs. J. Algorithms **11**(1), 85–101 (1990)
52. D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano, Sparse dynamic programming I: Linear cost functions. J. ACM **39**(3), 519–545 (1992); A preliminary version appeared, in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, (1990), pp. 513–522
53. D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano, Sparse dynamic programming II: Convex and concave cost functions. J. ACM **39**(3), 546–567 (1992); A preliminary version appeared, in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 513–522
54. R. Fleischer, M.J. Golin, Y. Zhang, 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* (2004), pp. 102–113
55. R.F. Floyd, Algorithm 97: Shortest path. Commun. ACM **5**(6), 345 (1962)
56. Z. Galil, R. Giancarlo, Speeding up dynamic programming with applications to molecular biology. Theor. Comput. Sci. **64**(1), 107–118 (1989)
57. Z. Galil, K. Park, Dynamic programming with convexity, concavity and sparsity. Theor. Comput. Sci. **92**(1), 49–76 (1992)
58. W.H. Gates, C.H. Papadimitriou, Bounds for sorting by prefix reversal. Discret. Math. **27**, 47–57 (1979)
59. E.N. Gilbert, E.F. Moore, Variable length encodings. Bell Syst. Tech. J. **38**, 933–967 (1959)
60. D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (Cambridge University Press, Cambridge, 1997)
61. R. Hassin, A. Tamir, Improved complexity bounds for location problems on the real line. Oper. Res. Lett. **10**(7), 395–402 (1991)
62. H. Heydari, I. Hal Sudborough, On the diameter of the pancake network. J. Algorithms **25**(1), 67–94 (1997)
63. D.S. Hirschberg, L.L. Larmore, The least weight subsequence problem. SIAM J. Comput. **16**(4), 628–638 (1987)
64. H. Hoogeveen, A. Vestjens, Optimal on-line algorithms for single-machine scheduling, in *Proceedings of 5th Conference Integer Programming and Combinatorial Optimization (IPCO)* (Springer Verlag, London, 1996), pp. 404–414

65. T.C. Hu, A.C. Tucker, Optimal computer search trees and variable-length alphabetical codes. SIAM J. Appl. Math. **21**(4), 514–532 (1971)
66. A. Karlin, On the performance on competitive algorithms in practice, in *Online Algorithms: The State of the Art*, ed. by A. Fiat, G.J. Woeginger (Springer, 1998), pp. 373–382
67. A. Karlin, C. Kenyon, D. Randall, Dynamic TCP acknowledgment and other stories about e/(e-1). Algorithmica **36**(3), 209–224 (2003)
68. A. Karlin, M. Manasse, L. Rudolph, D. Sleator, Competitive snoopy caching. Algorithmica **3**, 79–119 (1988)
69. M.M. Klawe, Superlinear bounds for matrix searching problems. J. Algorithms **13**(1), 55–78 (1992); A preliminary version appeared; in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 485–493
70. M.M. Klawe, D.J. Kleitman, An almost linear time algorithm for generalized matrix searching. SIAM J. Discret. Math. **3**(1), 81–97 (1990)
71. D.E. Knuth, Optimum binary search trees. Acta Inf. **1**, 14–25 (1971)
72. E. Koutsoupias, C. Papadimitriou, On the k-server conjecture. J. ACM **42**, 971–983 (1995)
73. D. Kravets, J.K. Park, Selection and sorting in totally monotone arrays. Theory Comput. Syst. **24**(3), 201–220, (1991); A preliminary version appeared, in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 494–502
74. R. Kuik, M. Salomon, L.N. van Wassenhove, Batching decisions: structure and models. Eur. J. Oper. Res. **75**, 243–263 (1994)
75. L.L. Larmore, D.S. Hirschberg, A fast algorithm for optimal length-limited Huffman codes. J. ACM **37**(3), 464–473 (1990); A preliminary version appeared, in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 310–318
76. L.L. Larmore, T.M. Przytycka, Parallel construction of trees with optimal weighted path length, in *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures* (ACM Press, New York, 1991), pp. 71–80
77. L.L. Larmore, B. Schieber, On-line dynamic programming with applications to the prediction of rna secondary structure. J. Algorithms **12**, 490–515 (1991)
78. L.L. Larmore, B. Schieber, 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* (1990), pp. 503–512
79. E. Lawler, J. Lensta, A.H.G Rinooy Kan, D. Smoys (eds.), *The Traveling Salesman: A Guided Tour of Combinatorial Optimization* (Wiley, New York, 1985)
80. M. Mahajan, R. Rama, V. Raman, S. Vijayakumar, Approximate block sorting. Int. J. Found. Comput. Sci. **12**(2), 337–356 (2006)
81. M. Mahajan, R. Rama, S. Vijayakumar, Block sorting: a characterization and some heuristics. Nord. J. Comput. **14**(1), 25 (2007)
82. M. Manasse, L.A. McGeoch, D. Sleator, Competitive algorithms for server problems. J. Algorithms **11**, 208–230 (1990)
83. S. Meyn, *Control Techniques for Complex Networks* (Cambridge University Press, Cambridge, MA, 2007)
84. J.K. Park, The Monge array: an abstraction and its applications. PhD thesis, Massachusetts Institute of Technology, 1991
85. C.N. Potts, L.N. van Wassenhove, Integrating scheduling with batching and lot-sizing: A review of algorithms and complexity. J. Oper. Res. Soc. **43**, 395–406 (1992)
86. B. Schieber, Computing a minimum weight k-link path in graphs with the concave Monge property. J. Algorithms **29**(2), 204–222 (1998); A preliminary version appeared, in *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 405–411
87. M. Sniedovich, *Dynamic Programming: Foundations and Principles*. (Taylor and Francis, Boca Raton, FL, 2010)
88. N. Stokey, R.E. Lucas, E. Prescott, *Recursive Methods in Economic Dynamics* (Harvard University Press, Cambridge, MA, 1989)

89. A. Tamir, An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs. Oper. Res. Lett. **19**(2), 59–64 (1996)
90. A. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **13**, 260–269 (1967)
91. M.L. Wachs, On an efficient dynamic programming technique of F. F. Yao. J. Algorithms **10**(4), 518–530 (1989)
92. S. Warshall, A theorem on boolean matrices. J. ACM **9**(1) 11–12 (1962)
93. R.L. Wessner, Optimal alphabetic search trees with restricted maximal height. Inf. Process. Lett. **4**(4), 90–94 (1976)
94. R. Wilber, The concave least-weight subsequence problem revisited. J. Algorithms **9**(3), 418–425 (1988)
95. G.J. Woeginger, Monge strikes again: Optimal placement of web proxies in the Internet. Oper. Res. Lett. **27**(3), 93–96 (2000)
96. F.F. Yao, Efficient dynamic programming using quadrangle inequalities, in *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (ACM Press, New York, 1980), pp. 429–435
97. F.F. Yao, Speed-up in dynamic programming. SIAM J. Matrix Anal. Appl. **3**(4), 532–540 (1982)
98. G. Zhang, X. Cai, C.Y. Lee, C.K. Wong, Minimizing makespan on a single batch processing machine with nonidentical job sizes. Naval Res. Logist. **48**, 226–240 (2001)