The Monge Array: An Abstraction and Its Applications

by

James Kimbrough Park

B.S.E., Electrical Engineering and Computer Science
Princeton University
(1985)
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

© Massachusetts Institute of Technology 1991

Signature of Author
Department of Electrical Engineering and Computer Science
May 20, 1991

Certified by
Charles E. Leiserson
~~Associate~~ Professor of Computer Science and Engineering
~~Thesis Supervisor~~

Accepted by
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 24 1991
LIBRARIES

The Monge Array: An Abstraction and Its Applications

by

James Kimbrough Park

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1991,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

This thesis develops a body of versatile algorithmic techniques. We demonstrate the power and generality of these techniques by applying them to a wide variety of problems. These problems are drawn from such diverse areas of study as computational geometry, VLSI theory, operations research, and molecular biology.

The algorithmic techniques described in this thesis are centered around a family of highly-structured arrays known as Monge arrays. An $m \times n$ array $A = \{a[i,j]\}$ is called Monge if

$$a[i,j] + a[k,\ell] \leq a[i,\ell] + a[k,j]$$

for all i, j, k, and ℓ such that $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$. We will show that Monge arrays capture the essential structure of many practical problems, in the sense that algorithms for searching in the abstract world of Monge arrays can be used to obtain efficient algorithms for these practical problems.

The first part of this thesis describes the basic Monge-array abstraction. We begin by defining several different types of Monge and Monge-like arrays. These definitions include a generalization of the notion of two-dimensional Monge arrays to higher-dimensional arrays. We also present several important properties of Monge and Monge-like arrays and introduce a computational framework for manipulating such arrays. We then develop a variety of algorithms for searching in Monge arrays. In particular, we give efficient sequential and parallel (PRAM) algorithms for computing minimal entries in Monge arrays and efficient sequential algorithms for selection and sorting in Monge arrays. Highlights include an $O(dn \lg^{d-2} n)$-time sequential algorithm for computing the minimum entry in an $n \times n \times \cdots \times n$ d-dimensional Monge array, an $O(n^{3/2} \lg^2 n)$-time sequential algorithm for computing the median entry in each row of an $n \times n$ two-dimensional Monge array, and an optimal $O(\lg n)$-time, $(n^2/\lg n)$-processor CREW-PRAM algorithm for computing the minimum entry in each $1 \times n \times 1$ subarray of an $n \times n \times n$ three-dimensional Monge array.

The second part of this thesis investigates the diverse applications of the Monge-array abstraction. We first consider a number of geometric problem relating to convex polygons in the plane. Specifically, we use Monge-array techniques to develop efficient algorithms for several proximity problems involving the vertices of a convex polygon, as well as the maximum-perimeter-inscribed-k-gon problem and the minimum-area-circumscribing-k-gon problem. We

then present several applications of Monge-array techniques to problems involving dynamic programming. These applications include a special case of the traveling salesman problem, the optimal-binary-search-tree problem, and several variants of the economic lot-size problem from operations research, We conclude with several parallel algorithms for a shortest-paths problem involving certain grid-like directed acyclic graphs. These algorithms are used to obtain fast parallel algorithms for string editing and surface reconstruction from planar contours. Highlights of this part of the thesis include an $O(kn + n \lg n)$-time sequential algorithm for the minimum-area-circumscribing-k-gon problem, an $O(n)$-time sequential algorithm for a special case of the n-vertex traveling-salesman problem, an $O(n^2)$-time sequential algorithm for the backlogging economic lot-size problem with arbitrary concave production, inventory, and backlogging cost functions, and an $O(\lg^2 n)$-time, $(n^2/\lg n)$-processor CREW-PRAM algorithm for the string-editing problem.

Following the body of this thesis is an appendix that provides a comprehensive overview of the Monge-array abstraction and its many applications. This appendix is organized as a list of problems and includes many results not discussed elsewhere in the thesis.

Keywords: algorithmic techniques, Monge arrays, array searching, convex polygons, dynamic programming, the traveling-salesman problem, optimal binary search trees, economic lot-sizing, string editing, surface reconstruction from planar contours.

Thesis Supervisor: Charles E. Leiserson

Title: Associate Professor of Computer Science and Engineering

Acknowledgements

First and foremost, I must acknowledge Alok Aggarwal. As my mentor and frequent coauthor, he deserves an enormous amount of credit for the contents of this thesis. I am deeply indebted to him for introducing me to the wonderful world of Monge arrays and for showing confidence in my ability as a researcher when my own self-confidence was low. Without him, this thesis would never have been written. I must also thank my thesis advisor Charles Leiserson for his advice, his patience, and the financial support he provided me. I am especially grateful for his suggestion that I compile the Monge-array compendium included at the end of the thesis; this compendium allowed me to indulge my desire to mention every single Monge-array result that I know about and still finish the thesis. I would also like to thank Leo Guibas, the third member of my thesis committee, for his helpful comments on a early draft of the thesis.

Portions of this thesis represents collaborative work with Alok Aggarwal, Dina Kravets, Yishay Mansour, Baruch Schieber, and Sandeep Sen. I have also benefited greatly from technical discussions with Tom Cormen, Mic Grigni, Alex Ishii, Nabil Kahale, Mike Klugerman, Mark Newman, Carolyn Haibt Norton, Rob Schapire, Eric Schwabe, Cliff Stein, Joel Wein, and Julia Yang.

In my six years at M.I.T., I have found the Theory of Computation group a very stimulating environment, and I have learned a great deal from its faculty and students. I would also like to acknowledge the group's amazing support staff, especially William Ang, Be Hubbard, David Jones, and Denise Sergent.

The work presented in this thesis was supported in part by the Defense Advanced Research Projects Agency under Contracts N00014-87-K-0825 and N00014-89-J-1988, the Office of Naval Research under Contract N00014-86-K-0593, and an NSF Graduate Fellowship.

In memory of Bessie Byrne Gorrell Park

Contents

Introduction 1

I The Abstraction 7

1 Preliminaries 9
- 1.1 Two-Dimensional Monge Arrays . . . 9
- 1.2 Higher-Dimensional Monge Arrays . . . 17
- 1.3 Related Concepts . . . 23
- 1.4 The Computational Model . . . 25

2 Minimization Algorithms 27
- 2.1 Two-Dimensional Monge Arrays . . . 29
- 2.2 On-Line Algorithms . . . 38
- 2.3 Higher-Dimensional Monge Arrays . . . 40
- 2.4 Partial Monge Arrays . . . 50

3 Selection and Sorting Algorithms 51
- 3.1 Row Selection . . . 53
  - 3.1.1 Row Selection When k is Small . . . 54
  - 3.1.2 Row Selection When k is Large . . . 56
- 3.2 Array Selection . . . 66
- 3.3 Row Sorting . . . 68
- 3.4 Array Sorting . . . 70
- 3.5 Open Problems . . . 71

4 Parallel Algorithms 73
- 4.1 Preliminaries . . . 74
- 4.2 Two-Dimensional Monge Arrays . . . 76
- 4.3 Plane Minima in Three-Dimensional Monge Arrays . . . 85
- 4.4 Tube Minima in Three-Dimensional Monge Arrays . . . 86

II The Applications **95**

5 Convex-Polygon Problems **97**
5.1 Intervertex Distances 97
5.2 Maximum-Perimeter Inscribed d-Gons 103
5.3 Minimum-Area Circumscribing d-Gons 105
5.3.1 Finding the Best Flush d-gon 107
5.3.2 Using the Best Flush d-gon to Obtain the Best Arbitrary d-gon 112

6 Two Dynamic-Programming Applications **119**
6.1 A Special Case of the Traveling-Salesman Problem 120
6.2 Yao's Dynamic-Programming Problem 128
6.2.1 Optimal Binary Search Trees 128
6.2.2 Yao's Algorithm 131
6.2.3 An Alternate Quadratic-Time Algorithm 133

7 Dynamic Programming and Economic Lot Sizing **137**
7.1 Background and Definitions 139
7.1.1 The Basic Model 139
7.1.2 The Backlogging Model 144
7.1.3 Two Periodic Models 148
7.2 Arborescent Flows and Dynamic Programming 148
7.3 The Basic Problem 151
7.3.1 Nearly Linear Costs 152
7.3.2 Other Cost Structures 160
7.4 The Backlogging Problem 162
7.4.1 Nearly Linear Costs 162
7.4.2 Concave Costs 168
7.4.3 Other Cost Structures 171
7.5 Two Periodic Problems 173
7.5.1 Erickson, Monma, and Veinott's Problem 173
7.5.2 Graves and Orlin's Problem 174
7.6 Some Final Remarks 177

8 Shortest Paths in Grid DAGs **181**
8.1 A Shortest-Paths Algorithm 182
8.2 String Editing and Related Problems 189
8.3 Surface Reconstruction from Planar Contours 190

Conclusion **193**

A A Monge-Array Compendium **195**
A.1 Array-Searching Problems 195
A.2 Geometric Problems 198
A.3 VLSI Problems 202
A.4 Dynamic-Programming Problems 203

A.5 Problems from Operations Research 206
A.6 Graph-Theoretic Problems 208

Bibliography **211**

Introduction

This subject of this thesis is Monge arrays and their applications to algorithm design. An $m \times n$ two-dimensional array (or matrix) of real numbers, denoted $A = \{a[i,j]\}$, is called a *Monge array* if it satisfies the following property: for all rows i_1 and i_2 and columns j_1 and j_2 satisfying $1 \leq i_1 < i_2 \leq m$ and $1 \leq j_1 < j_2 \leq n$,

$$a[i_1, j_1] + a[i_2, j_2] \leq a[i_1, j_2] + a[i_2, j_1] .$$

In other words, if we consider any 2×2 subarray of A, corresponding to any two rows and any two columns, then the sum of the upper left and lower right entries is always at most the sum of the upper right and lower left entries. Figure I.1 depicts an array with this property.

Monge arrays take their name from the French mathematician Gaspard Monge (1746–1818). He is associated with such arrays because of work done by Hoffman [Hof63] on easily-solved special cases of the transportation problem. Hoffman showed that if the cost array associated with a transportation problem is an $m \times n$ Monge array, then a simple greedy algorithm solves the transportation problem in $O(m + n)$ time. Hoffman applied Monge's name to such arrays because, as Hoffman remarked, "the essential idea behind [the crucial observation exploited in Hoffman's paper] was first noticed by Monge in 1781!" (See Appendix A for more information on Hoffman's work.)

We study Monge arrays in the thesis for two mutually dependent reasons.

First, the structure of Monge arrays allows us to locate certain entries in a Monge array without having to examine all the array's entries. For example, we need only examine $O(m+n)$ of the mn entries in an $m \times n$ Monge array to find the array's smallest entry. (This result, due to Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87], is described in Section 2.1.)

			j_1			j_2	
	10	17	13	28	23	38	49
	17	22	16	29	23	35	45
	24	28	22	34	24	33	40
i_1	11	13	6	17	7	15	21
	45	44	32	37	23	28	32
	36	33	19	21	6	7	10
i_2	73	66	51	53	34	30	31
	62	52	32	32	13	9	6
	77	66	45	43	21	15	8

Figure I.1: Consider the 9×7 array $A = \{a[i,j]\}$ shown above, which is drawn so that the entry $a[1,1] = 10$ appears in the upper left corner. This array is Monge, since for any two rows $i_1 < i_2$ and any two columns $j_1 < j_2$, the sum of $a[i_1, j_1]$ and $a[i_2, j_2]$ is at most the sum of $a[i_1, j_2]$ and $a[i_2, j_1]$ For example, if we set $i_1 = 4$, $i_2 = 7$, $j_1 = 3$, and $j_2 = 6$, we find $a[4,3]$ and $a[7,6]$ (the entries in white boxes) sum to 36, while $a[4,6]$ and $a[7,3]$ (the entries in black boxes) sum to 66.

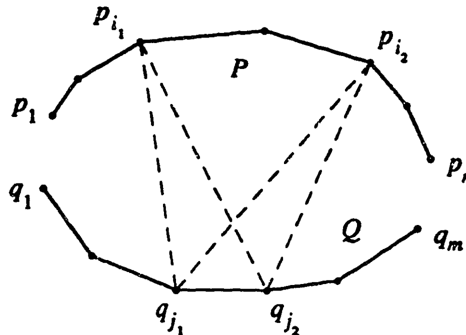

Figure I.2: For $i_1 < i_2$ and $j_1 < j_2$, $\mathrm{d}(p_{i_1}, q_{j_1}) + \mathrm{d}(p_{i_2}, q_{j_2}) \geq \mathrm{d}(p_{i_1}, q_{j_2}) + \mathrm{d}(p_{i_2}, q_{j_1})$.

Second, many algorithmic problems from theoretical computer science and related areas can be reduced to finding certain entries in Monge arrays. Moreover, combining these reductions with efficient algorithms for searching in Monge arrays often yields new and improved algorithms for the original problems.

As an example of such a reduction, consider the following closest-pair problem from computational geometry. Suppose we are given a convex polygon that has been broken into two convex chains P and Q (containing m and n vertices, respectively) by the removal of two edges, as is shown in Figure I.2. Furthermore, let $p_1, \ldots, p_m$ denote the vertices of P in clockwise order and let $q_1, \ldots, q_n$ denote the vertices of Q in counterclockwise order. The problem we

The trading of options and the scientific study of options both have long histories, yet both underwent revolutionary changes at virtually the same time in the early 1970s. These changes, and the subsequent events to which they led, have greatly increased the practical value of a thorough understanding of options.

(a)

The trading of options and the scientific study of options both have long histories, yet both underwent revolutionary changes at virtually the same time in the early 1970s. These changes, and the subsequent events to which they led, have greatly increased the practical value of a thorough understanding of options.

(b)

Figure I.3: Two different ways of forming a left- and right-justified paragraph from the same sequence of words.

want to solve is that of finding a vertex p_i of P and a vertex q_j of Q minimizing the Euclidean distance $\mathrm{d}(p_i, q_j)$ separating p_i and q_j.

This closest-pair problem can be reduced to a Monge-array problem as follows. Let $A = \{a[i,j]\}$ denote the $m \times n$ array where $a[i,j] = \mathrm{d}(p_i, q_j)$. This array is Monge. To see why, consider any two rows i_1 and i_2 and columns j_1 and j_2 such that $1 \leq i_1 < i_2 \leq m$ and $1 \leq j_1 < j_2 \leq n$. As indicated in Figure I.2, the entries $a[i_1, j_1]$ and $a[i_2, j_2]$ correspond to opposite sides of the quadrilateral formed by p_{i_1}, p_{i_2}, q_{j_1}, and q_{j_2}, and the entries $a[i_1, j_2]$ and $a[i_1, j_2]$ correspond to diagonals. By the *quadrangle inequality* (which states that the sum of the lengths of the diagonals of any quadrilateral is greater than the sum of the lengths of any pair of opposite sides), we have

$$\mathrm{d}(p_{i_1}, q_{j_1}) + \mathrm{d}(p_{i_2}, q_{j_2}) \leq \mathrm{d}(p_{i_1}, q_{j_2}) + \mathrm{d}(p_{i_2}, q_{j_1}) .$$

Thus, A is Monge, and we have reduced our closest-pair problem to the problem of finding the smallest entry in a Monge array.

As a second (more natural) motivating example (borrowed from Hirschberg and Larmore [HL87]), consider the following simple paragraph-formation problem. We are given a sequence of n words $w_1, w_2, \ldots, w_n$, where the ith word w_i has length ℓ_i, and we want to form a left- and right-justified paragraph from these words, so that each line of the paragraph (except the last) has a length that is as close to an ideal line length L as possible. Figure I.3 shows two different ways of forming a paragraph from the same sequence of words.

More precisely, let B denote the length of the ideal spacing between two words, and for

$1 \le i < j \le n$, let $\ell_{i,j}$ denote the natural length of a line containing words w_i through w_{j-1}, i.e., let

$$\ell_{i,j} = \left(\sum_{m=i}^{j-1} \ell_m\right) + (j-i-1)B .$$

Furthermore, for $1 \le i < j \le n+1$, let $w(i,j)$ denote the penalty assigned to a line containing words w_i through w_{j-1}. Presumably, this penalty function is chosen so that $w(i,j)$ is small when $\ell_{i,j}$ is close to L and large when $\ell_{i,j}$ is significantly smaller or larger than L. For example, we might have

$$w(i,j) = \begin{cases} (\ell_{i,j} - L)^2 & \text{if } j \le n, \\ 0 & \text{if } j = n+1 \text{ and } \ell_{i,j} \le L, \\ +\infty & \text{if } j = n+1 \text{ and } \ell_{i,j} > L. \end{cases}$$

Now forming the sequence of words $w_1, \ldots, w_n$ into a paragraph is equivalent to choosing a number of lines p and a sequence of line breaks $b[1], b[2], \ldots, b[p], b[p+1]$, where $1 = b[1] < b[2] < \cdots < b[p] < b[p+1] = n+1$ and the paragraph's ith line consists of words $w_{b[i]}$ through $w_{b[i+1]-1}$. Thus, the optimal paragraph-formation problem is that of choosing p and $b[1], b[2], \ldots, b[p]$ so that

$$\sum_{k=1}^{p} w(b[k], b[k+1]) .$$

is minimized.

The optimal paragraph-formation problem has a natural dynamic-programming formulation. Specifically, for $1 \le j \le n+1$, let $E(j)$ denote the penalty of the minimum-penalty breaking of words $w_1, \ldots, w_{j-1}$ into lines. We can then write

$$E(j) = \begin{cases} 0 & \text{if } j = 1, \\ \min_{1 \le i < j} \{E(i) + w(i,j)\} & \text{if } 2 \le j \le n+1. \end{cases}$$

Note that computing $E(2), \ldots, E(n+1)$ in the naive fashion gives an $O(n^2)$-time algorithm for the optimal paragraph-formation problem.

So where is the Monge array lurking in this problem? Consider the $n \times (n+1)$ array

$A = \{a[i,j]\}$ where

$$a[i,j] = \begin{cases} E(i) + w(i,j) & \text{if } i < j, \\ +\infty & \text{if } i \geq j. \end{cases}$$

As we shall see in Section 1.1, this array is Monge for many natural penalty functions $w(\cdot,\cdot)$. Moreover, for $2 \leq j \leq n+1$ $E(j)$ is the minimum entry in column j of A.

In this thesis, we will undertake a detailed study of Monge arrays and their applications. The goal of the thesis is to demonstrate both the power and generality of the Monge-array techniques developed herein. We will concentrate, of course, on this author's own research, but several fundamental algorithms due to other researchers will also be covered in detail. Furthermore, in several places, we will recast others' work in terms of the Monge-array framework developed in this thesis.

We conclude this introduction with an outline of the thesis. The body of this thesis is divided into two parts. Part I describes the basic Monge-array abstraction, while Part II investigates its diverse applications.

Part I consists of four chapters. In Chapter 1, we define several different types of Monge and Monge-like arrays and present a number of properties of such arrays. We also introduce a computational framework for manipulating Monge arrays. Then, in Chapters 2 through 4, we develop algorithms for searching in Monge arrays. In particular, Chapter 2 contains sequential algorithms for computing minimal entries in Monge arrays. Joint work with Aggarwal [AP89b] is included, along with results due to Aggarwal, Klawe, Moran, Shor and Wilber [AKM+87] and Larmore and Schieber [LS91]. (Additional algorithms due to Klawe and Kleitman [KK90] are also mentioned for use in Part II.) Chapter 3 gives more sequential algorithms, this time for selection and sorting in Monge arrays. It contains joint work with Kravets [KP91] and with Mansour, Schieber, and Sen [MPSS91]. Finally, Chapter 4 presents parallel algorithms for computing minimal entries in Monge arrays. The algorithms given in this chapter represent collaborative work with Aggarwal [AP89a]. (We also mention algorithms due to Apostolico, Atallah, Larmore, and McFaddin [AALM90], Atallah [Ata90], and Atallah and Kosaraju [AK91].)

Part II also consists of four chapters. Chapter 5 centers around convex polygons in the plane; it considers several problems involving the distances separating a convex polygon's vertices, as well as the maximum-perimeter inscribed d-gon problem and minimum-area circum-

scribing **d-gon problem.** The results presented in the chapter represent work by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] and joint work with Aggarwal [AP89b], Kravets [KP91], and Mansour, Sen, and Schieber [MPSS91]. The next two chapters focus on applications of the Monge-array abstraction to problems involving dynamic programming. Chapter 6 uses Monge-array techniques to obtain efficient algorithms for a special case of the traveling salesman problem and a family of dynamic-programming problems satisfying the quadrangle inequality studied by Yao in [Yao80]. The former application was first described in [Par91], while the latter application represents joint work with Aggarwal [AP89b]. Chapters 7, which again covers joint work with Aggarwal [AP91], provides efficient algorithms for several variants of the economic lot-size problem from operations research. The last chapter of Part II, Chapter 8, begins by presenting a parallel shortest-paths algorithm based on the parallel algorithms of Chapter 4. This algorithm is then used to provide parallel algorithms for string editing and surface reconstruction from planar contours. This chapter represents joint work with Aggarwal [AP89a].

Following the body of this thesis is Appendix A, which provides a comprehensive overview of the Monge-array abstraction and its many applications. This appendix is organized as a list of problems and includes many results not discussed elsewhere in this thesis.

Part I

The Abstraction

Chapter 1

Preliminaries

The Monge-array abstraction may be decomposed into two conceptual parts: the mathematical notion of a Monge array and the algorithmic machinery for searching in such arrays. The former part will be the focus of this chapter. (We will postpone the discussion of algorithms for searching in Monge arrays to Chapters 2 through 4.)

This chapter is organized as follows. In Section 1.1, we define a two-dimensional Monge array and present several basic properties of such arrays. Then, in Section 1.2, we generalize the notion of Mongeness[1] to d-dimensional arrays, where $d \geq 2$. We also present several properties of d-dimensional Monge arrays and describe several important subclasses of such arrays. Section 1.3 briefly describes the following related concepts: totally monotonicity, the quadrangle inequality, submodular functions, and partial Monge arrays. Finally, Section 1.4 introduces our computational model.

1.1 Two-Dimensional Monge Arrays

In this section, we discuss two-dimensional Monge arrays and their basic properties. We begin with our primary definition of a two-dimensional Monge array.

[1] Leo Guibas has proposed instead the term *Mongité*.

Definition 1.1 An $m \times n$ two-dimensional array $A = \{a[i,j]\}$ is *Monge* if for all i, j, k, and ℓ such that $1 \le i < k \le m$ and $1 \le j < \ell \le n$, we have

$$a[i,j] + a[k,\ell] \;\le\; a[i,\ell] + a[k,j]\,.$$

The requirements of this definition are actually stronger than they need to be. Specifically, we have the following lemma.

Lemma 1.1 Let $A = \{a[i,j]\}$ denote an $m \times n$ array. If

$$a[i,j] + a[i+1,j+1] \;\le\; a[i,j+1] + a[i+1,j]$$

for all i and j such that $1 \le i < n$ and $1 \le j < m$, then A is Monge.

Proof Suppose

$$a[s,t] + a[s+1,t+1] \;\le\; a[s,t+1] + a[s+1,t]$$

for all s and t such that $1 \le s < n$ and $1 \le t < m$, and consider any i, j, k, and l such that $1 \le i < k \le n$ and $1 \le j < l \le m$. For $1 \le t < m$,

$$\sum_{s=i}^{k-1}(a[s,t] + a[s+1,t+1]) \;\le\; \sum_{s=i}^{k-1}(a[s,t+1] + a[s+1,t])\,.$$

Canceling identical terms from both sides of this inequality, we obtain

$$a[i,t] + a[k,t+1] \;\le\; a[i,t+1] + a[k,t]\,.$$

Consequently,

$$\sum_{t=j}^{\ell-1}(a[i,t] + a[k,t+1]) \;\le\; \sum_{t=j}^{\ell-1}(a[i,t+1] + a[k,t])\,.$$

Again canceling identical terms, we obtain

$$a[i,j] + a[k,\ell] \;\le\; a[i,\ell] + a[k,j]\,.$$

This implies A is Monge. ■

Since

$$a[i,j]+a[k,\ell] \le a[i,\ell]+a[k,j] .$$

for $1 \le i < k \le m$ and $1 \le j < \ell \le n$ implies

$$a[i,j]+a[i+1,j+1] \le a[i,j+1]+a[i+1,j]$$

for $1 \le i < m$ and $1 \le j < n$, the following alternate definition of a two-dimensional Monge array is equivalent to Definition 1.1.

Definition 1.2 An $m \times n$ two-dimensional array $A = \{a[i,j]\}$ is *Monge* if for all i and j such that $1 \le i < m$ and $1 \le j < n$, we have

$$a[i,j]+a[i+1,j+1] \le a[i,j+1]+a[i+1,j] .$$

Definition 1.3 An $m \times n$ two-dimensional array $A = \{a[i,j]\}$ is *inverse-Monge* if for all i, j, k, and ℓ such that $1 \le i < k \le m$ and $1 \le j < \ell \le n$, we have

$$a[i,j]+a[i+1,j+1] \ge a[i,j+1]+a[i+1,j] .$$

Definition 1.4 An $m \times n$ two-dimensional array $A = \{a[i,j]\}$ is *inverse-Monge* if for all i and j such that $1 \le i < m$ and $1 \le j < n$, we have

$$a[i,j]+a[i+1,j+1] \ge a[i,j+1]+a[i+1,j] .$$

We will now give ten useful properties of two-dimensional Monge arrays. (Analogous properties hold for inverse-Monge arrays.) We begin with two very simple but fundamental properties.

Property 1.1 Let $A = \{a[i,j]\}$ denote an $m \times n$ array. If A is Monge, then for all indices i, j, and ℓ satisfying $1 \le i \le m$ and $1 \le j < \ell \le n$,

1. $a[i,j] < a[i,\ell]$ implies $a[k,j] < a[k,\ell]$ for all k satisfying $1 \le k < i$,
2. $a[i,j] \le a[i,\ell]$ implies $a[k,j] \le a[k,\ell]$ for all k satisfying $1 \le k < i$,

3. $a[i,j] \geq a[i,\ell]$ implies $a[k,j] \geq a[k,\ell]$ for all k satisfying $i < k \leq m$, and
4. $a[i,j] > a[i,\ell]$ implies $a[k,j] > a[k,\ell]$ for all k satisfying $i < k \leq m$.

Equivalently, if A is Monge, then for all indices j and ℓ satisfying $1 \leq j < \ell \leq n$, there exist indices I_1 and I_2 satisfying $0 \leq I_1 \leq I_2 \leq m$ such that

1. $a[i,j] < a[i,\ell]$ for all i satisfying $1 \leq i \leq I_1$,
2. $a[i,j] = a[i,\ell]$ for all i satisfying $I_1 < i \leq I_2$, and
3. $a[i,j] > a[i,\ell]$ for all i satisfying $I_2 < i \leq m$.

■

Property 1.2 Let $A = \{a[i,j]\}$ denote an $m \times n$ array, and let B denote a subarray of A corresponding to a subset of A's rows and columns. If A is Monge, then B is Monge. ■

An important consequence of Property 1.1 is the following property.

Property 1.3 Let $A = \{a[i,j]\}$ denote an $m \times n$ array, and for $1 \leq i \leq m$, let $j(i)$ denote the column of A containing the leftmost minimum entry in row i, so that

$$a[i,j(i)] = \min_{1 \leq j \leq n} a[i,j] .$$

If A is Monge, then

$$j(1) \leq j(2) \leq \cdots \leq j(n) .$$

■

The next seven properties relate to the construction of Monge arrays.

Property 1.4 Let $A = \{a[i,j]\}$ denote an $m \times n$ array, and let $B = \{b[i,j]\}$ denote the $n \times m$ transpose of A, so that $b[i,j] = a[j,i]$. If A is Monge, then B is Monge. ■

Property 1.5 Let $A = \{a[i,j]\}$ and $B = \{b[i,j]\}$ denote $m \times n$ arrays, and let $C = \{c[i,j]\}$ denote the entry-wise sum of A and B, so that $c[i,j] = a[i,j] + b[i,j]$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. If both A and B are Monge, then C is Monge. ■

Property 1.6 Let $A = \{a[i,j]\}$ denote an $m \times n$ array, let $B = \{b[i]\}$ denote an m-vector, and let $C = \{c[j]\}$ denote an n-vector. If $a[i,j] = b[i]$ for $1 \le i \le m$, then A is Monge. Similarly, if $a[i,j] = c[j]$ for $1 \le j \le n$, then A is again Monge. ∎

The following property relates Monge arrays and concave functions. A function $f(\cdot)$ mapping real numbers to real numbers is called *concave* if

$$f(x+z) - f(x) \le f(y+z) - f(y)$$

for all $z \ge 0$ and all $x \ge y$.

Property 1.7 Let $f(\cdot)$ denote a function mapping real numbers to real numbers, let $B = \{b[i]\}$ denote an m-vector, and let $C = \{c[j]\}$ denote an n-vector. Furthermore, let $A = \{a[i,j]\}$ denote the $m \times n$ array where $a[i,j] = f(b[i] + c[j])$. If

1. $f(\cdot)$ is concave,
2. $b[1] \le b[2] \le \cdots \le b[m]$, and
3. $c[1] \le c[2] \le \cdots \le c[n]$,

then A is Monge.

Proof Consider any i and j such that $1 \le i < m$ and $1 \le j < n$, and let $x = b[i] + c[j+1]$, $y = b[i]+c[j]$, and $z = b[i+1]-b[i]$. Clearly, $a[i,j] = f(y)$, $a[i,j+1] = f(x)$, $a[i+1,j] = f(y+z)$, and $a[i+1,j+1] = f(x+z)$. Moreover, $x \ge y$ and $z \ge 0$. Thus, by the definition of a concave function,

$$a[i,j] + a[i+1,j+1] \;=\; f(y) + f(x+z) \;\le\; f(x) + f(y+z) \;=\; a[i,j+1] + a[i+1,j]\,,$$

which implies A is Monge. ∎

As an example of why these rather simple observations are useful, consider the following lemma, which follows from Properties 1.6, 1.5, 1.7.

Lemma 1.2 Let $B = \{b[i]\}$ and $D = \{d[i]\}$ denote arbitrary m-vectors, and let $C = \{c[j]\}$ and $E = \{e[j]\}$ denote arbitrary n-vectors. Furthermore, let $A = \{a[i,j]\}$ denote the $m \times n$

array where $a[i,j] = b[i]c[j] + d[i] + e[j]$. If $b[1] \leq b[2] \leq \cdots \leq b[m]$ and $c[1] \geq c[2] \geq \cdots \geq c[n]$, then A is Monge.

Proof Let $A' = \{a'[i,j]\}$ denote the $m \times n$ array where $a'[i,j] = b[i]c[j]$, and consider the concave function $f(x) = -2^x$, the m-vector $B' = \{b'[i]\}$ where $b'[i] = \log_2 b[i]$, and the n-vector $C' = \{c'[j]\}$ where $c'[j] = -\log_2(c[j])$. Clearly, $a'[i,j] = -f(b'[i] - c'[j])$, and the entries of B' and C' are both in increasing order; thus, by Property 1.7, A' is Monge. Furthermore, since $a[i,j] = a'[i,j] + d[i] + e[j]$, A is also Monge, by Properties 1.6 and 1.5. ■

Note that even if the entries of B and C in the above lemma are not sorted, we can still make the array A Monge by permuting its rows and columns. Specifically, if we find permutations β and γ such that $b[\beta(1)] \leq b[\beta(2)] \leq \cdots \leq b[\beta(m)]$ and $c[\gamma(1)] \geq c[\gamma(2)] \geq \cdots \geq c[\gamma(n)]$, then the array $A'' = \{a''[i,j]\}$ where

$$a''[i,j] \;=\; b[\beta(i)]c[\gamma(j)] + d[\beta(i)] + e[\gamma(j)] \;=\; a[\beta(i),\gamma(j)]$$

is Monge.

Property 1.8 Let $A = \{a[i,j]\}$ denote an $m \times n$ array. Furthermore, for $1 \leq i \leq m$ and $1 \leq j \leq n$, let

$$L(i,j) \;=\; |\{j' : 1 \leq j' < j \text{ and } a[i,j'] < a[i,j]\}|$$

and

$$R(i,j) \;=\; |\{j' : j \leq j' \leq m \text{ and } a[i,j'] \leq a[i,j]\}| \,.$$

If A is Monge, then for $1 \leq j \leq n$,

$$L(1,j) \;\geq\; L(2,j) \;\geq\; \cdots \;\geq\; L(m,j)$$

and

$$R(1,j) \;\leq\; R(2,j) \;\leq\; \cdots \;\leq\; R(m,j)\,.$$

Proof For any two rows i and k such that $1 \leq i < k \leq m$ and any column j such that $1 \leq j \leq n$, suppose $a[k,j'] < a[k,j]$ for some j' such that $1 \leq j' \leq j$. By Property 1.1, this implies

$a[i,j'] < a[i,j]$. As this is true for any j' such that $1 \le j' \le j$, we must have $L(i,j) \ge L(k,j)$. Similarly, if $a[i,j'] \le a[i,j]$ for some j' such that $j < j' \le m$, then $a[k,j'] \le a[k,j]$. Thus, $R(i,j) \le R(k,j)$. ■

Property 1.9 Let $A = \{a[i,j]\}$ denote an $m \times n$ array. Furthermore, for $1 \le \ell < n$, let $B_\ell = \{b_\ell[i,j]\}$ denote the $m \times (n-1)$ array where

$$b_\ell[i,j] = \begin{cases} a[i,j] & \text{if } 1 \le j < \ell, \\ \min\{a[i,\ell], a[i,\ell+1]\} & \text{if } j = \ell, \\ a[i,j+1] & \text{if } \ell < j < n, \end{cases}$$

for $1 \le i \le m$ and $1 \le j < n$. (Intuitively, B_ℓ is A with columns ℓ and $\ell+1$ replaced by a new column formed from the row minima of the $m \times 2$ subarray of A corresponding to columns ℓ and $\ell+1$.) If A is Monge, then for all ℓ between 1 and $n-1$, B_ℓ is Monge.

Proof We will prove B_ℓ is Monge using Definition 1.2. Consider any i and j such that $1 \le i < m$ and $1 \le j < n-1$. Let

$$j' = \begin{cases} j & \text{if } j < \ell, \\ j & \text{if } j = \ell \text{ and } a[i+1,j] \le a[i+1,j+1], \\ j+1 & \text{if } j = \ell \text{ and } a[i+1,j] > a[i+1,j+1], \\ j+1 & \text{if } j > \ell, \end{cases}$$

and let

$$j'' = \begin{cases} j+1 & \text{if } j+1 < \ell, \\ j+1 & \text{if } j+1 = \ell \text{ and } a[i,j+1] \le a[i,j+2], \\ j+2 & \text{if } j+1 = \ell \text{ and } a[i,j+1] > a[i,j+2], \\ j+2 & \text{if } j+1 > \ell. \end{cases}$$

Clearly, $j' < j''$, $b_\ell[i,j+1] = a[i,j'']$ and $b_\ell[i+1,j] = a[i+1,j']$. Moreover, $b_\ell[i,j] \le a[i,j']$ and $b_\ell[i+1,j+1] \le a[i+1,j'']$. (We may have $b_\ell[i,j] < a[i,j']$ if $j = \ell$, $a[i+1,j] > a[i+1,j+1]$, and $a[i,j] < a[i,j+1]$, and, similarly, we may have $b_\ell[i+1,j+1] < a[i+1,j'']$ if $j+1 = \ell$,

$a[i, j+1] < a[i, j+2]$, and $a[i+1, j+1] > a[i+1, j+2]$.) Thus, since A is Monge, we have

$$b_\ell[i,j]+b_\ell[i+1,j+1] \;\le\; a[i,j']+a[i+1,j''] \;\le\; a[i,j'']+a[i+1,j'] \;=\; b_\ell[i,j+1]+b_\ell[i+1,j]\,,$$

which implies B_ℓ is Monge. ■

The above property, though not used in this thesis, is used by Aposolico, Atallah, Larmore, and McFaddin [AALM90] and Atallah and Kosaraju [AK91] to obtain efficient parallel algorithms for computing the row minima of a Monge array.

Property 1.10 Let $B = \{b[i,j]\}$ denote an $m \times n$ Monge array, where $m \ge n$. If each column of B contains at least one row maximum, then each row of B is bitonic, i.e., for $1 \le i \le m$,

$$b[i,1] \;\le\; \cdots \;\le\; b[i,c(i)-1] \;\le\; b[i,c(i)]$$

and

$$b[i,c(i)] \;>\; b[i,c(i)+1] \;>\; \cdots \;>\; b[i,n]\,,$$

where $c(i)$ denotes the column containing the maximum entry in row i.

Proof Suppose each column of B contains at least one row maximum, but some row of B is not bitonic. This means there exist indices i, j_1, and j_2 such that $1 \le i \le m$, $1 \le j_1 < j_2 \le n$, and either

(1) $j_1 < j_2 < c(i)$ and $b[i,j_1] > b[i,j_2]$, or

(2) $c(i) < j_1 < j_2$ and $b[i,j_1] < b[i,j_2]$.

We consider only the first possibility; the proof for the second possibility is analogous. Since each column of B contains at least one row maximum, there exists an i' such that $c(i') = j_2$. We must have $i' < i$, since Mongeness implies monotonicity. Now consider the 2×2 subarray of B corresponding to rows i' and i and columns j_1 and j_2. (This subarray is depicted in Figure 1.1.) Since $b[i',c(i')]$ is the maximum entry in row i', we have $b[i',j_1] < b[i',c(i')]$. By assumption (1), $b[i,j_1] > b[i,c(i')]$. This contradicts the Mongeness of B. ■

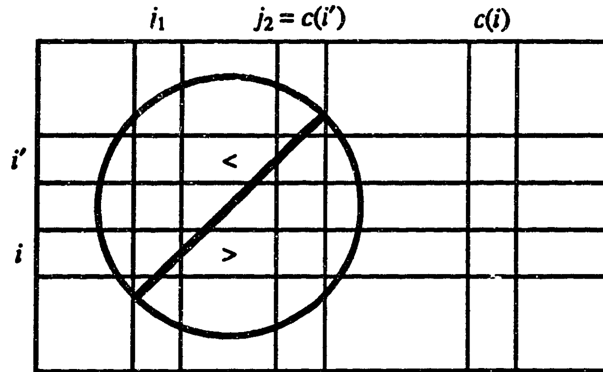

Figure 1.1: If the maximum entry in row i' lies in column j_2, then by the Mongeness of B, we cannot have $b[i, j_1] > b[i, j_2]$.

1.2 Higher-Dimensional Monge Arrays

In this section, we generalize the definition of a two-dimensional Monge array given in Section 1.1 to d-dimensional arrays, $d \geq 2$, and present a number of fundamental properties of such arrays. We also describe several important subclasses of d-dimensional Monge arrays.

We begin with our primary definition of a d-dimensional Monge array.

Definition 1.5 For $d \geq 2$, an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ is *Monge* if for all $i_1, i_2, \ldots, i_d$ and $j_1, j_2, \ldots, j_d$ such that $1 \leq i_k \leq n_k$ and $1 \leq j_k \leq n_k$ for $1 \leq k \leq d$, we have

$$a[s_1, s_2, \ldots, s_d] + a[t_1, t_2, \ldots, t_d] \leq a[i_1, i_2, \ldots, i_d] + a[j_1, j_2, \ldots, j_d] ,$$

where for $1 \leq k \leq d$, $s_k = \min\{i_k, j_k\}$ and $t_k = \max\{i_k, j_k\}$.

Note how this definition reduces to Definition 1.1 when $d = 2$.

As was the case with our first definition of a two-dimensional Monge array, the requirements of this definition are again stronger than they need to be. Specifically, we have the following lemma.

Lemma 1.3 Let $A = \{a[i_1, i_2, \ldots, i_d]\}$ denote an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array. If every two-dimensional subarray of A corresponding to fixed values of $d-2$ of A's d indices is Monge, then A is Monge.

Proof We use an induction on d to show that this lemma holds. For the base case of $d = 2$, the lemma follows immediately from the definition of a two-dimensional Monge array.

Now suppose the lemma holds for all $(d-1)$-dimensional arrays, and consider a d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ and any two entries $a[i_1, i_2, \ldots, i_d]$ and $a[j_1, j_2, \ldots, j_d]$ from A. We must show that

$$a[s_1, s_2, \ldots, s_d] + a[t_1, t_2, \ldots, t_d] \leq a[i_1, i_2, \ldots, i_d] + a[j_1, j_2, \ldots, j_d] ,$$

where for $1 \leq k \leq d$, $s_k = \min\{i_k, j_k\}$ and $t_k = \max\{i_k, j_k\}$.

Without loss of generality, we assume $i_1 < j_1$, i.e., $s_1 = i_1$ and $t_1 = j_1$. The proof then breaks down into two cases.

Case 1 For all k between 2 and d, $i_k > j_k$ (i.e., $s_k = j_k$ and $t_k = i_k$).

Consider the $(d-1)$-dimensional subarray of A containing those entries whose second coordinate is i_2 and the $(d-1)$-dimensional subarray of A containing those entries whose second coordinate is j_2. If every two-dimensional subarray of A corresponding to fixed values of $d-2$ of A's d indices is Monge, then every two-dimensional subarray of a subarray of A is Monge. Thus, we can invoke the inductive hypothesis on the $(d-1)$-dimensional subarrays corresponding to i_2 and j_2 and obtain

$$a[i_1, i_2, j_3, \ldots, j_d] + a[j_1, i_2, i_3, \ldots, i_d] \leq a[i_1, i_2, i_3, \ldots, i_d] + a[j_1, i_2, j_3, \ldots, j_d]$$

and

$$a[i_1, j_2, j_3, \ldots, j_d] + a[j_1, j_2, i_3, \ldots, i_d] \leq a[i_1, j_2, i_3, \ldots, i_d] + a[j_1, j_2, j_3, \ldots, j_d] .$$

Similarly, we can invoke the inductive hypothesis on the two-dimensional subarray containing those entries whose third through d-th indices are $i_3, \ldots, i_d$, respectively, and on the two-dimensional subarray containing those entries whose third through d-th indices are $j_3, \ldots, j_d$,

respectively. This gives us

$$a[i_1, j_2, i_3, \ldots, i_d] + a[j_1, i_2, i_3, \ldots, i_d] \leq a[i_1, i_2, i_3, \ldots, i_d] + a[j_1, j_2, i_3, \ldots, i_d]$$

and

$$a[i_1, j_2, j_3, \ldots, j_d] + a[j_1, i_2, j_3, \ldots, j_d] \leq a[i_1, i_2, j_3, \ldots, j_d] + a[j_1, j_2, j_3, \ldots, j_d] .$$

Summing these four inequalities and canceling, we find

$$2a[i_1, j_2, j_3, \ldots, j_d] + 2a[j_1, i_2, i_3, \ldots, i_d] \leq 2a[i_1, i_2, i_3, \ldots, i_d] + 2a[j_1, j_2, j_3, \ldots, j_d] .$$

Since $s_1 = i_1$, $t_1 = j_1$, and for $2 \leq k \leq d$, $s_k = j_k$ and $t_k = i_k$, this gives the desired result.

Case 2 There exists an l, $2 \leq l \leq n$, such that $i_l < j_l$ (i.e., $s_l = i_l$ and $t_l = j_l$).

Consider the $(d-1)$-dimensional subarray containing those entries whose first coordinate is i_1 and the $(d-1)$-dimensional subarray containing those entries whose first coordinate is j_1. By applying the inductive hypothesis to these subarrays, we obtain

$$a[i_1, s_2, s_3, \ldots, s_d] + a[i_1, t_2, t_3, \ldots, t_d] \leq a[i_1, i_2, i_3, \ldots, i_d] + a[i_1, j_2, j_3, \ldots, j_d]$$

and

$$a[j_1, s_2, s_3, \ldots, s_d] + a[j_1, t_2, t_3, \ldots, t_d] \leq a[j_1, i_2, i_3, \ldots, i_d] + a[j_1, j_2, j_3, \ldots, j_d] .$$

Similarly, by applying the inductive hypothesis to the $(d-1)$-dimensional subarray containing those entries whose l-th coordinate is i_l and to the $(d-1)$-dimensional subarray containing those entries whose l-th coordinate is j_l, we obtain

$$a[i_1, s_2, s_3, \ldots, s_d] + a[j_1, i_2, i_3, \ldots, i_d] \leq a[i_1, i_2, i_3, \ldots, i_d] + a[j_1, s_2, s_3, \ldots, s_d]$$

and

$$a[i_1, j_2, j_3, \ldots, j_d] + a[j_1, t_2, t_3, \ldots, t_d] \leq a[i_1, t_2, t_3, \ldots, t_d] + a[j_1, j_2, j_3, \ldots, j_d] .$$

Summing these four inequalities and canceling, we find

$$2a[i_1, s_2, s_3, \ldots, s_d] + 2a[j_1, t_2, t_3, \ldots, t_d] \geq 2a[i_1, i_2, i_3, \ldots, i_d] + 2a[j_1, j_2, j_3, \ldots, j_d] .$$

Since $s_1 = i_1$ and $t_1 = j_1$, this gives the desired result. ■

Now suppose an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ is Monge in the sense of Definition 1.5, i.e., for all $i_1, i_2, \ldots, i_d$ and $j_1, j_2, \ldots, j_d$ such that $1 \leq i_k \leq n_k$ and $1 \leq j_k \leq n_k$ for $1 \leq k \leq d$, we have

$$a[s_1, s_2, \ldots, s_d] + a[t_1, t_2, \ldots, t_d] \leq a[i_1, i_2, \ldots, i_d] + a[j_1, j_2, \ldots, j_d] ,$$

where for $1 \leq k \leq d$, $s_k = \min\{i_k, j_k\}$ and $t_k = \max\{i_k, j_k\}$. This clearly implies every two-dimensional plane of A is Monge. Thus, the following alternate definition of a d-dimensional Monge array is equivalent to Definition 1.5.

Definition 1.6 For $d \geq 2$, an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ is *Monge* if every two-dimensional subarray of A corresponding to fixed values of $d - 2$ of A's d indices is Monge.

Higher-dimensional inverse-Monge arrays are defined in an analogous fashion.

We now give five important properties of higher-dimensional Monge arrays.

Property 1.11 Let $A = \{a[i_1, i_2, \ldots, i_d]\}$ denote an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array. Furthermore, for $1 \leq i_1 \leq n_1$ and $2 \leq k \leq d$, let $i_k(i_1)$ denote the k-th index of the minimum entry in the $(d-1)$-dimensional subarray of A corresponding to those entries whose first index is i_1, so that

$$a[i_1, i_2(i_1), \ldots, i_d(i_1)] = \min_{\substack{i_2, \ldots, i_d \\ \text{s.t. } 1 \leq i_k \leq n_k \\ \text{for } 2 \leq k \leq d}} a[i_1, i_2, \ldots, i_d] .$$

If A is Monge, then for $2 \le k \le d$,

$$i_k(1) \le i_k(2) \le \cdots \le i_k(n_1) .$$

■

Property 1.12 Let $A = \{a[i_1, i_2, \ldots, i_d]\}$ denote an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array. Furthermore, for $1 \le i_1 \le n_1$, $1 \le i_2 \le n_2$, and $3 \le k \le d$, let $i_k(i_1, i_2)$ denote the k-th index of the minimum entry in the $(d-2)$-dimensional subarray of A corresponding to those entries whose first index is i_1 and whose second index is i_2, so that

$$a[i_1, i_2, i_3(i_1, i_2), \ldots, i_d(i_1, i_2)] = \min_{\substack{i_3, \ldots, i_d \\ \text{s.t. } 1 \le i_k \le n_k \\ \text{for } 3 \le k \le d}} a[i_1, i_2, i_3, \ldots, i_d] .$$

If A is Monge, then for $1 \le i_2 \le n_2$ and $2 \le k \le d$,

$$i_k(1, i_2) \le i_k(2, i_2) \le \cdots \le i_k(n_1, i_2)$$

and for $1 \le i_1 \le n_1$ and $2 \le k \le d$,

$$i_k(i_1, 1) \le i_k(i_1, 2) \le \cdots \le i_k(i_1, n_2) .$$

■

Let $A = \{a[i_1, \ldots, i_d]\}$ and $B = \{b[i_1, \ldots, i_d]\}$ be $n_1 \times \cdots \times n_d$ d-dimensional arrays. The *sum of A and B* (written $A + B$) is the $n_1 \times \cdots \times n_d$ d-dimensional array $C = \{c[i_1, \ldots, i_d]\}$ where

$$c[i_1, \ldots, i_d] = a[i_1, \ldots, i_d] + b[i_1, \ldots, i_d] ,$$

for all $i_1, \ldots, i_d$.

Now let $E = \{e[i_1, \ldots, i_d]\}$ be an $n_1 \times \cdots \times n_d$ d-dimensional array. For any dimension k between 1 and $d + 1$ and any size $\hat{n}$, the

$$n_1 \times \cdots \times n_{k-1} \times \hat{n} \times n_k \times \cdots \times n_d$$

$(d+1)$-dimensional array $F = \{f[i_1, \ldots, i_{d+1}]\}$ is an *extension of* E if

$$f[i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_{d+1}] = e[i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{d+1}] ,$$

for all $i_1, \ldots, i_{d+1}$. (F is just $\hat{n}$ copies of E, each one a plane of F corresponding to some fixed value of F's k-th coordinate.) Furthermore, any extension of an extension of E is also an extension of E.

Property 1.13 The sum of two d-dimensional Monge arrays is also Monge. ■

Property 1.14 For all $d' > d$, every d'-dimensional extension of a d-dimensional Monge array is Monge. ■

An important subclass of d-dimensional Monge arrays consists of what we call *Monge-composite* arrays. As one might expect, an array is Monge-composite if it is composed of two-dimensional Monge arrays. More precisely, we have the following definition.

Definition 1.7 A d-dimensional array is *Monge-composite* if it is the sum of d-dimensional extensions of two-dimensional Monge arrays and *inverse-Monge-composite* if it is the sum of d-dimensional extensions of two-dimensional inverse-Monge arrays.

From these definitions, it is clear that each entry of a d-dimensional Monge-composite array $A = \{a[i_1, \ldots, i_d]\}$ may be written

$$a[i_1, \ldots, i_d] = \sum_{k<l} w_{k,l}[i_k, i_l] ,$$

where for all $k < l$, the $n_k \times n_l$ array $W_{k,l} = \{w_{k,l}[i_k, i_l]\}$ is a Monge array.

Property 1.15 Every Monge-composite array is Monge, and, similarly, every inverse-Monge-composite array is inverse-Monge. ■

Proof Let A denote a Monge-composite array. We must show that all two-dimensional planes of A, corresponding to fixed values of $d-2$ of A's d coordinates, are Monge. To see why this is true, consider any such plane. This plane is the sum of a two-dimensional Monge array, some vectors, and some scalars; thus, the plane is Monge.

A similar argument shows that every inverse-Monge-composite array is inverse-Monge. ■

We conclude with two special cases of Monge-composite arrays.

Definition 1.8 An $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ is *path-decomposable* if for all d-tuples $i_1, i_2, \ldots, i_d$ such that $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$, we have

$$a[i_1, \ldots, i_d] = w_{1,2}[i_1, i_2] + w_{2,3}[i_2, i_3] + \cdots + w_{d-1,d}[i_{d-1}, i_d] ,$$

where for $1 \leq k < d$, $W_{k,k+1} = w_{k,k+1}[i_k, i_{k+1}]$ is an $n_k \times n_{k+1}$ two-dimensional Monge array.

Definition 1.9 An $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$ is *cycle-decomposable* if for all d-tuples $i_1, i_2, \ldots, i_d$ such that $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$, we have

$$a[i_1, \ldots, i_d] = w_{1,2}[i_1, i_2] + w_{2,3}[i_2, i_3] + \cdots + w_{d-1,d}[i_{d-1}, i_d] + w_{d,1}[i_d, i_1] ,$$

where $W_{d,1} = \{w_{d,1}[i_d, i_1]\}$ is an $n_d \times n_1$ two-dimensional Monge array and for $1 \leq k < d$, $W_{k,k+1} = \{w_{k,k+1}[i_k, i_{k+1}]\}$ is an $n_k \times n_{k+1}$ two-dimensional Monge array.

1.3 Related Concepts

In this section, we introduce several concepts related to the notion of Monge arrays.

A two-dimensional array $A = \{a[i,j]\}$ is called *totally monotone* if for all $i_1 < i_2$ and $j_1 < j_2$, $a[i_1, j_1] < a[i_1, j_2]$ implies $a[i_2, j_1] < a[i_2, j_2]$. Every inverse-Monge array is totally monotone, but not vice-versa.

A interval function $f(\cdot,\cdot)$ is said to satisfy the *quadrangle inequality* if for all i, i', j, and j' satisfying $1 \leq i \leq i' \leq j \leq j' \leq n$, we have

$$f(i,j) + f(i',j') \leq f(i,j') + f(i',j) .$$

Similarly, $f(\cdot,\cdot)$ is said to satisfy the *inverse quadrangle inequality* if for all i, i', j, and j' satisfying $1 \leq i \leq i' \leq j \leq j' \leq n$, we have

$$f(i,j) + f(i',j') \geq f(i,j') + f(i',j) .$$

A function $f(\cdot,\cdot)$ satisfies the quadrangle inequality if and only if the $n \times n$ array $A = \{a[i,j]\}$ where

$$a[i,j] = \begin{cases} f(i,j) & \text{if } i \leq j, \\ +\infty & \text{if } i > j, \end{cases}$$

is Monge.

Higher-dimensional Monge arrays are closely related to sub- and supermodular functions. A function $f(\cdot)$ mapping subsets of some set S to real numbers is called *submodular* if for all $A, B \subseteq S$,

$$f(A) + f(B) \leq f(A \cap B) + f(A \cup B) .$$

Similarly, $f(\cdot)$ is called *supermodular* if for all $A, B \subseteq S$,

$$f(A) + f(B) \geq f(A \cap B) + f(A \cup B) .$$

We can view a $2{\times}2{\times}\cdots{\times}2$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ as a submodular function $f(\cdot)$ on subsets of $\{1, 2, \ldots, d\}$ if we let $f(S) = a[i_1, i_2, \ldots, i_d]$, where for $1 \leq k \leq d$, $i_k = 1$ if $k \notin S$ and $i_k = 2$ if $k \in S$. (See [Lov83] for an overview of the theory of sub- and supermodular functions.)

A two-dimensional array $A = \{a[i,j]\}$ is called a *partial Monge array* if

1. only some of its entries are "interesting," and

2. every 2×2 subarray containing four "interesting" entries satisfies the Monge condition.

There are several varieties of partial Monge arrays. A $m \times n$ partial Monge array $A = \{a[i,j]\}$ is called a *v-array* if every column's "interesting" entries form a contiguous subcolumn. Similarly, A is called an *h-array* if every row's "interesting" entries form a contiguous subrow. A *skyline-Monge* or *stalagmite-Monge* array is a v-array such that the "interesting" entries in any particular column end in row m, and a *stalactite-Monge* array is a v-array such that the "interesting" entries in any particular column start in row 1. Finally, a *staircase-Monge* array has the property that if $a[i,j]$ is "interesting," then so are $a[i,\ell]$ and $a[k,j]$ for all $\ell > j$ and all $k > i$.

1.4 The Computational Model

In this section, we discuss our computation model.

We model a Monge array as function that returns any entry in constant time.

We can assume all the entries in a Monge array are distinct.

We often use $+\infty$'s in the Monge arrays we construct, We define $+\infty + x = +\infty$ for all x.

Chapter 2

Minimization Algorithms

This chapter is the first of three presenting algorithms for searching in Monge and Monge-like arrays. In this chapter, we focus on sequential algorithms for computing minimal entries, while Chapter 3 presents sequential selection and sorting algorithms, and Chapter 4 describes parallel minimization algorithms.

In Section 2.1, we consider the problem of computing the minimum entry in each row of a two-dimensional Monge array. We call this the *row-minimization* problem for A. We show that the row-minimization problem for an $m \times n$ Monge array A can be solved in $O(n)$ time if $m \leq n$ and in $O(n(1+\lg(m/n)))$ time if $m > n$, provided any entry of A can be computed in constant time. We also prove that these time bounds are optimal (up to constant factors). This result is due to Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]. Note that computing the row maxima of A is no harder than computing its row minima; we need only negate the entries of A and reverse the ordering of its columns to convert back and forth between the two problems.

In Section 2.2, we consider an on-line variant of the row-minimization problem. Specifically, we focus on $n \times (n+1)$ arrays $A = \{a[i,j]\}$ where for $1 \leq i \leq j \leq n$, $a[i,j] = +\infty$, and for $1 \leq j < i \leq n$, $a[i,j]$ depends on the minimum entry in row j of A, i.e., the j-th row minimum of A must be computed before $a[i,j]$ can be evaluated. We call the row-minimization problem for such an array A an *on-line* row-minimization problem. (By way of contrast, the row-minimization problem for an array $A = a[i,j]$ is called *off-line* if any entry $a[i,j]$ of A can always be evaluated in constant time.) We show that the on-line row-minimization problem

Problem Type	*Array Type*	*Time*	*Theorem*
off-line	Monge	$\Theta(n)$	2.4
	staircase-Monge	$O(n\alpha(n))$	2.14
on-line	Monge	$\Theta(n)$	2.6
	staircase-Monge	$O(n\alpha(n))$	2.15

Table 2.1: Results for off-line and on-line row minimization in an $n \times n$ Monge or partial Monge array.

for an $n \times n$ Monge array can be solved in $\Theta(n)$ time. The algorithm that we describe for this problem is due to Larmore and Schieber [LS91].

In Section 2.3, we consider a generalization of the off-line row-minimization problem to the higher-dimensional Monge arrays of Section 1.2. Given an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$, $d \geq 2$, the *plane-minima* problem for A is that of computing the minimum entry in each $(d-1)$-dimensional *plane* of A, where the i_1-th plane of A consists of those entries of A whose first index is i_1. In other words, the i_1-th plane minimum of A is

$$\min_{\substack{i_2,\ldots,i_d \\ \text{s.t. } 1 \leq i_k \leq n_k \\ \text{for } 2 \leq k \leq d}} a[i_1, i_2, \cdots, i_d] .$$

We show that the plane-minima problem for an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array A can be solved in $O((n_1 + n_2 + \cdots + n_d) \lg n_1 \lg n_2 \cdots \lg n_{d-2})$ time. We also show that, in contrast to the two-dimensional case, the plane-maxima problem for A is significantly harder than the plane-minima problem; specifically, computing the plane maxima of A requires $\Omega((n_1 n_2 \cdots n_d)/(n_1 + n_2 + \cdots + n_d - d))$ time. Finally, we show that if A is a path-decomposable Monge-composite array then the plane minima of A can be computed in $O(n_1 + n_2 + \cdots + n_d)$ time, and if A is cycle-decomposable, then its plane minima can be computed in $O(n_1 + (n_2 + \cdots + n_d) \lg n_1)$ time. All of these results represent joint work with Aggarwal that first appeared in [AP89b].

Finally, in Section 2.4, we turn to row minimization problems involving the partial Monge arrays of Section 1.3. We briefly mention two algorithms due to Klawe and Kleitman [KK90].

Tables 2.1 and 2.2 summarize the algorithms given in the chapter for computing minimal entries in Monge and stalagmite-Monge arrays.

Problem Type	*Array Type*	*Time*	*Theorem*
minimization	general	$O(dn \lg^{d-2} n)$	2.7
	cycle-decomposable	$O(dn \lg n)$	2.10
	path-decomposable	$O(dn)$	2.9
maximization	general	$\Omega(n^{d-1}/d)$	2.11
		$O(n^{d-1})$	2.13

Table 2.2: Results for plane minimization and maximization in an $n \times n \times \cdots \times n$ d-dimensional Monge array.

2.1 Two-Dimensional Monge Arrays

This section presents an optimal sequential algorithm for computing the minimum entry in each row of a two-dimensional Monge array. This algorithm was developed by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]. It is of central importance to the study of Monge arrays, and we will use it repeatedly in this thesis. To save time and space, we will adopt the convention of Wilber [Wil88] and call this algorithm the SMAWK algorithm. (Wilber coined the name SMAWK by permuting the first letters of the algorithm's originators' last names.)

The SMAWK algorithm was originally developed for the related problem of computing the row maxima of a two-dimensional totally-monotone array. However, as every inverse-Monge array is totally monotone and negating the entries of a Monge array gives an inverse-Monge array whose maximal entries are the original array's minimal entries (see Section 1.1), the original algorithm given by Aggarwal et al. is easily transformed into an algorithm for computing the row minima of a Monge array.

Before describing the SMAWK algorithm, we present a simpler divide-and-conquer algorithm for computing the row minima of a Monge algorithm. We include this algorithm for two reasons. First, the SMAWK algorithm uses this simpler algorithm as a subroutine when m is larger than n. Second, the algorithm illustrates a very simple but useful approach to array searching that works by identifying subarrays of a Monge array that cannot contain minimal entries. (We will use variations of this approach in Section 2.3 and Chapter 4.) This simpler row-minimization algorithm is given in the following lemma and its proof.

Lemma 2.1 The row minima of an $m \times n$ Monge array A can be computed in $O(n(1 + \lg m))$

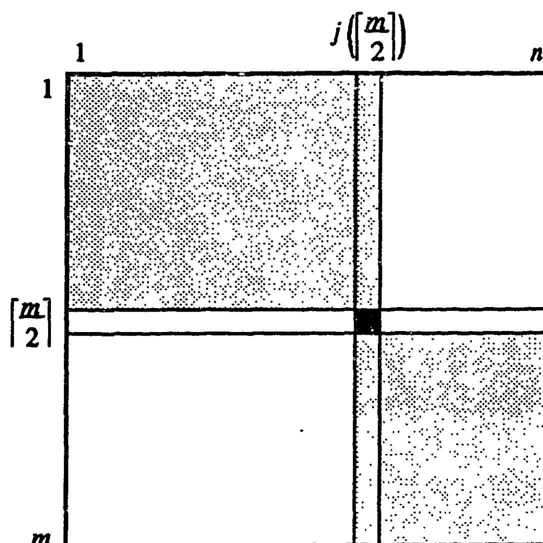

Figure 2.1: If the black square in the $m \times n$ Monge array A shown above denotes the minimum entry in row $\lceil m/2 \rceil$ of A, then the remaining row minima of A lie in the shaded regions.

time.

Proof Let $j(i)$ denote the column of A containing the minimum entry in row i of A. To obtain $j(1), \ldots, j(n)$ in $O(n(1 + \lg m))$ time, we use a simple divide-and-conquer approach. If $n = 1$, no work is necessary, as $j(1) = \cdots = j(m) = 1$. Otherwise, we begin by computing $j(\lceil m/2 \rceil)$, the location of the minimum entry in row $\lceil m/2 \rceil$ of A. This computation takes $O(n)$ time. Now, since the row minima of A are monotone (by Property 1.3), we know that for $i < \lceil m/2 \rceil$, we must have $j(i) \leq j(\lceil m/2 \rceil)$, and similarly for $i > \lceil m/2 \rceil$, we must have $j(i) \geq j(\lceil m/2 \rceil)$. Thus, we need only consider entries in the two subarrays depicted in Figure 2.1 — one $(\lceil m/2 \rceil - 1) \times j(\lceil m/2 \rceil)$ and the other $\lfloor m/2 \rfloor \times (n - j(\lceil m/2 \rceil) + 1)$ — in computing the remaining row minima of A. These minima we compute recursively.

If we let $T(m, n)$ denote this algorithm's running time in computing the row minima of an $m \times n$ Monge array, then $T(m, 1) = 0$, $T(1, n) = O(n)$, and for $n \geq 2$ and $m \geq 2$,

$$T(m, n) = O(n) + \max_{1 \leq n' \leq n} \{T(\lceil m/2 \rceil - 1, n') + T(\lfloor m/2 \rfloor, n - n' + 1)\} .$$

To prove that this recurrence has the desired solution, we will show, by induction on m, that

$$T(m,n) \leq c_1(n-1)(1+\lg m) ,$$

where c_1 is a constant independent of m and n. (This inequality is our inductive hypothesis.) The base case of $m = 1$ is easy. If $n = 1$, then $T(m,n) = 0$; otherwise, $T(m,n) \leq c_2 n$ for some constant c_2. Thus, in both cases, $T(m,n) \leq c_1(n-1)(1+\lg m)$ so long as $c_2 \geq 2c_1$. Now assume that the inductive hypothesis holds for all values of $m < M$. If $n = 1$, then clearly $T(M,n) = 0 \leq c_1(n-1)(1+\lg M)$. Otherwise, by the recurrence for $T(m,n)$, we have

$$\begin{aligned} T(M,n) &\leq c_2 n + c_1(n'-1)(1+\lg(\lceil M/2\rceil - 1)) + c_1(n-n')(1+\lg\lfloor M/2\rfloor) \\ &\leq c_2 n + c_1(n-1)\lg M \end{aligned}$$

This last term is at most $c_1(n-1)(1+\lg M)$ provided $c_2 n \leq c_1(n-1)$ or $c_1 \geq 2c_2$. Thus, we have shown that the inductive hypothesis also holds for $m = M$. ■

At the heart of the SMAWK algorithm are two complementary techniques for reducing the size of a Monge-array row-minima problem. The first allows us to eliminate rows from the Monge array whose row minima we seek, whereas the second permits us to eliminate columns. The following two lemmas summarize these techniques.

Lemma 2.2 Given the minimum entry in each even-numbered row of an $m \times n$ Monge array $A = \{a[i,j]\}$, the remaining row minima (i.e., those in the odd-numbered rows) can be computed in $O(m+n)$ time.

Proof Let $j(i)$ denote the column of A containing the minimum entry in row i of A. Furthermore, let $j(0) = 1$, and let $j(m+1) = n$. (These values may be interpreted as describing "dummy" rows 0 and $m+1$ of A such that $a[0,1] < a[0,2] < \cdots < a[0,n]$ and $a[m+1,1] > a[m+1,2] > \cdots > a[m+1,n]$; such rows can be added without affecting the Mongeness of A.) By Property 1.3, the row minima of A are monotone, i.e., $j(i) \leq j(i+1)$ for $0 \leq i \leq m$. Thus, for all i in the range $0 \leq i < \lceil m/2\rceil$, the minimum entry in row $2i+1$ of A

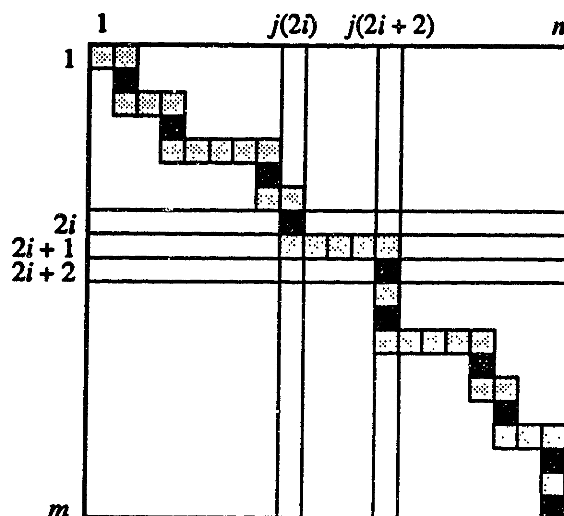

Figure 2.2: If the black squares denote the row minima in the even-numbered rows of a Monge array, then the row minima in the odd-numbered rows must lie in the shaded regions. In particular, if the minimum entry in row $2i$ lies in column $j(2i)$ and the minimum entry in row $2i+2$ lies in column $j(2i+2)$, then the minimum entry in row $2i+1$ must lie in one of columns $j(2i)$ through $j(2i+2)$, inclusive.

is the minimum of the $j(2i+2) - j(2i) + 1$ entries $a[2i+1, j(2i)], \ldots, a[2i+1, j(2i+2)]$. as suggested in Figure 2.2. Consequently, the minimum entry in row $2i+1$ can be computed in $O(j(2i+2) - j(2i) + 1)$ time, and all the row minima in odd-numbered rows can be computed in

$$\begin{aligned}\sum_{i=0}^{\lceil \frac{m}{2} \rceil - 1} O(j(2i+2) - j(2i) + 1) &= O\left(j\left(2\left\lceil \frac{m}{2} \right\rceil\right) - j(0) + \left\lceil \frac{m}{2} \right\rceil\right) \\ &= O(m+n)\end{aligned}$$

time. ■

Lemma 2.3 Given an $m \times n$ Monge array $A = \{a[i,j]\}$ such that $m \leq n$, we can identify $n - m$ columns of A that do not contain row minima in $O(n)$ time.

Proof Our algorithm for identifying columns that do not contain row minima consists of m steps, where in the jth step, we process column j of A. This processing is centered around a stack S holding columns of A that may contain row minima.

During the course of the algorithm, two important invariants are maintained. First, after

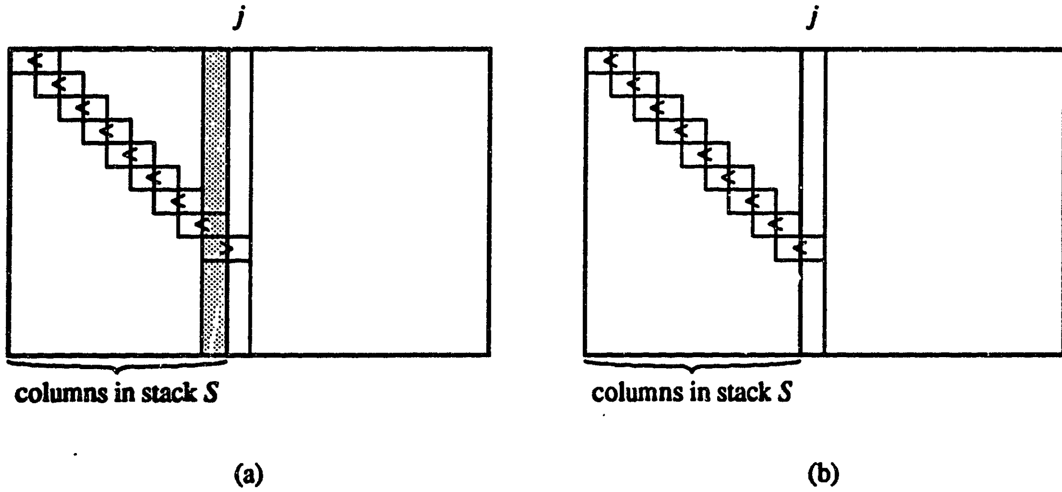

Figure 2.3: (a) If $a[s, S[s]] > a[s, j]$, we can eliminate a column. (b) On the other hand, if $a[s, S[s]] < a[s, j]$, we can push column j on the stack.

step j, S always contains a subsequence of $0, 1, \ldots, j$, i.e., $0 \leq S[1] < S[2] < \cdots < S[s] \leq j$, where $S[r]$ denotes the rth element in the stack, and s denotes the size of the stack, and $S[s]$ denotes the topmost element in the stack. Moreover, for any j' in the range $1 \leq j' \leq j$, $j' \notin S$ implies column j' cannot contain any row minima. Second, S always satisfies the following *staircase* condition: for $1 \leq r \leq s-1$, $a[r, S[r]] < a[r, S[r+1]]$.

To handle boundary conditions, we add two "dummy" rows 0 and $m+1$ to A and one "dummy" column 0, such that $a[0,0] < a[0,1] < \cdots < a[0,n] < a[0,n+1]$, $a[m+1,0] > a[m+1,1] > \cdots > a[m+1,n] > a[m+1,n+1]$, $a[i,0] > a[i,1]$ for $1 \leq i \leq m$. Note that these rows and columns can be added without destroying the Mongeness of A or changing the row minima in rows 1 through m. (We add these "dummy" rows and columns to simplify the presentation of this algorithm.)

Initially, the stack S contains column 0. Since $s = 1$, the staircase condition is trivially satisfied.

For $j = 1, \ldots, m$, we process column j as follows. We begin by comparing $a[s, S[s]]$ and $a[s, j]$. If $a[s, S[s]] > a[s, j]$, as in Figure 2.3(a), then by Property 1.1, $a[i, S[s]] > a[i, j]$ for all i in the range $s < i \leq m$. Furthermore, by the staircase condition, $a[s-1, S[s-1]] < a[s-1, S[s]]$. This last inequality implies $a[i, S[s-1]] < a[i, S[s]]$ for all i in the range $1 \leq i < s-1$, again by

Property 1.1. Thus, column $S[s]$ of A cannot contain any row minima. This observation allows us to pop $S[s]$ from the stack. We then compare column j to the new top of the stack, i.e., we compare $a[s, S[s]]$ and $a[s, j]$.

We continue in the manner until $a[s, S[s]] < a[s, j]$, as in Figure 2.3(b). We then push j on the stack. Note that the staircase condition is maintained.

Column 0 and row 0 of A insure that S never empties, and row $m+1$ insures that S never contains more than $m+2$ columns. Thus, after at most n steps, we eliminate $n-m$ columns from A.

To analyze the running time of the above procedure, we bound the total number of comparisons performed. We begin with a little notation. Let T denote the total of comparisons performed in eliminating $(n-m)$ columns. Furthermore, for $0 \leq t \leq T$ let d_t denote the total number of columns eliminated by the first t comparisons, and let s_t denote the size of stack S after tth comparison. Clearly, $d_0 = 0$, $d_T = n-m$, $s_0 = 1$, and $s_T \leq m+2$. Moreover, since the tth comparison either (1) pops a column from the stack and eliminates it from consideration, or (2) pushes a column on the stack, a simple inductive argument shows that $2d_t + s_t \geq t$ for all t in the range $0 \leq t \leq T$. Thus,

$$\begin{aligned} T &\leq 2(n-m) + m + 2 \\ &= 2n - m + 2 \,. \end{aligned}$$

As this last term is $O(n)$, we are done. ■

We can now describe the SMAWK algorithm of Aggarwal, Klawe, Moran, Shor, and Wilber for computing the row minima of an $m \times n$ Monge array A.

Theorem 2.4 (Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]) The row minima of an $m \times n$ Monge array A can be computed in $O(n)$ time if $m \leq n$ and in $O(n(1 + \lg(m/n)))$ time if $m > n$. Moreover, these time bounds are asymptotically optimal.

Proof This theorem's upper bounds are achieved as follows. We begin by assuming A is *square*, i.e., $m = n$. The SMAWK algorithm then works as follows. First, we apply Lemma 2.3

to the $\lfloor m/2 \rfloor \times m$ subarray of A consisting of A's even-numbered rows. This computation takes $O(m)$ time and produces an $\lfloor m/2 \rfloor \times \lfloor m/2 \rfloor$ array B whose row minima are the row minima of A's even-numbered rows. Then, we recursively compute the row minima of B. Finally, we apply Lemma 2.2 to compute the remaining row minima of A in $O(m)$ additional time. If we let $T(m)$ denote the time used by the SMAWK algorithm in computing the row minima of an $m \times m$ Monge array, then

$$T(m) = \begin{cases} O(1) & \text{if } m = 1, \\ T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + O(m) & \text{if } m \geq 2. \end{cases}$$

The solution to this familiar recurrence is, of course, $T(m) = O(m)$, i.e., the SMAWK algorithm's running time is linear for square Monge arrays.

Now suppose $m < n$. Using Lemma 2.3, we can identify, in $O(n)$ time, an $m \times m$ subarray of A containing all of A's row minima. This subarray's row minima (and hence A's row minima) can then be computed in $O(m)$ additional time, which gives the entire algorithm an $O(n)$ running time.

Finally, suppose $m > n$. For this case, let B denote the $n \times n$ subarray of A consisting of rows $1, r+1, 2r+1, \ldots, (n-1)r+1$ of A, where $r = \lceil m/n \rceil$. Furthermore, let $j(i)$ denote the column of A containing the minimum entry in row i of A, and let $j(nr+1) = n$. Since B is Monge (by Property 1.2), we can locate its row minima (i.e., compute $j(1), j(r+1), j(2r+2), \ldots, j((n-1)r+1))$ in $O(n)$ time. Now, for $1 \leq t \leq n$, let A_t denote the $(r-1) \times (j(tr+1) - j((t-1)r+1)$ subarray of A consisting of rows $(t-1)r+2$ through tr and columns $j(tr+1)$ through $j((t-1)r$ of A. (The last subarray A_n may actually contain fewer than $r-1$ rows, but for simplicity, we will ignore this detail.) Since the row minima of A are monotone, the remaining row minima of A are entries of $A_1, \ldots, A_n$. Moreover, for $1 \leq t \leq n$ the row minima of A_t can be computed in $O((j(tr+1) - j((t-1)r+1) \lg r)$ time using the algorithm of Lemma 2.1. Since

$$\begin{aligned} \sum_{t=1}^{n} (j(tr+1) - j((t-1)r+1) \lg r &\leq (j(nr+1) - j(1)) \lg r + n \lg r \\ &\leq (2n-1) \lg r \\ &= O(n(1 + \lg(m/n))) , \end{aligned}$$

the entire algorithm has an $O(n(1+\lg(m/n)))$ running time.

To prove that the SMAWK algorithm's running time is optimal, we use a slight generalization of the argument given in [AKM+87] for totally monotone arrays. We begin by proving an $\Omega(n)$-time bound that works for all values of m and n. Let $C = \{c[j]\}$ denote any vector of n real numbers, and let $A = \{a[i,j]\}$ denote the $m \times n$ array given by $a[i,j] = c[j]$. By Property 1.6, A is Monge. Moreover, to compute the row minima of A, we must determine the minimum entry in C, which requires examining at least one entry in each column of A. Thus, computing the row minima of an $m \times n$ Monge array requires $\Omega(n)$ time.

The previous bound applies for all m and n, but for $m > n$, the bound is not tight. Specifically, we will now show that computing the row minima of an $m \times n$ Monge array requires $\Omega(n(1+\lg(m/n))$ time when $m > n$.

We begin with the $n = 2$ special case of this lower bound. Let i^* denote any integer in the range $0 \leq i^* \leq m$, and let $A = \{a[i,j]\}$ denote the $m \times 2$ array where $a[i,1] = 0$ and $a[i,2] = 1$ if $1 \leq i \leq i^*$ and $a[i,1] = 1$ and $a[i,2] = 0$ if $i^* < i \leq m$. Such an array is depicted in Figure 2.4(a).

For all i^*, the array A is Monge. To see why, observe that for all i and j, $a[i,j] = f(b[i]+c[j])$ where $f(x) = x^2$,

$$b[i] = \begin{cases} 1 & \text{if } 1 \leq i \leq i^*, \\ 2 & \text{if } i^* < i \leq m, \end{cases}$$

and $c[j] = -j$. Furthermore, $f(\cdot)$ is convex, $b[1] \leq b[2] \leq \cdots \leq b[m]$, and $c[1] \geq c[2]$. Thus, by Property 1.7, A is Monge.

To complete the proof of the lower bound's $n = 2$ special case, we note that computing the row minima of A requires evaluating $\lceil \lg(m+1) \rceil = 1 + \lfloor \lg m \rfloor$ entries of A in the worst case. Thus, computing the row minima of an $m \times 2$ Monge array requires $\Omega(\lg m)$ time.

Turning now to the general lower bound, the basic idea here is to embed several independent copies of A into a larger Monge array A', as suggested in Figure 2.4(b). Specifically, for $1 \leq k \leq \lfloor n/2 \rfloor$, let i_k^* denote any integer in the range $0 \leq i_k^* \leq r$, where $r = \lfloor 2m/n \rfloor$. Furthermore, let $A' = \{a'[i,j]\}$ denote the $m \times n$ array given by $a'[i,j] = f(b[i] + c[j])$, where

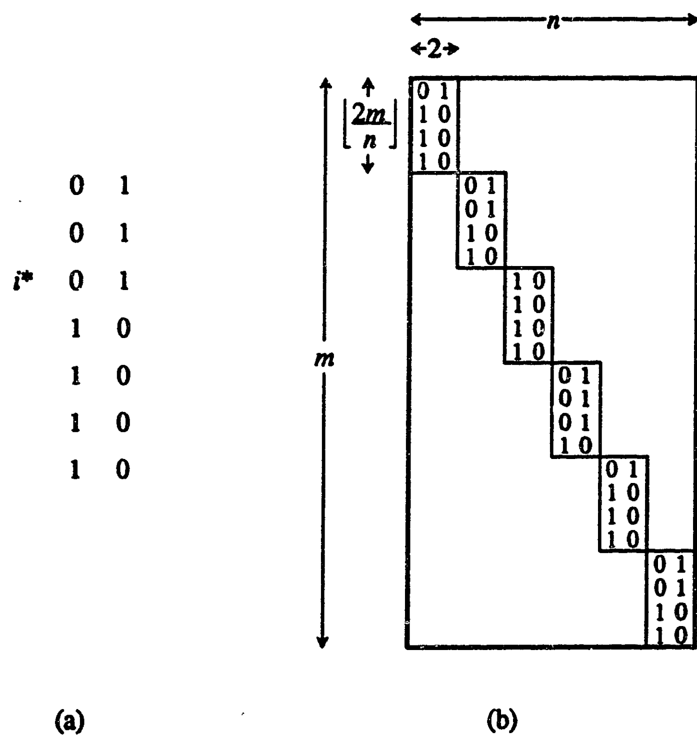

Figure 2.4: (a) The array A. (b) The array A' and the subarrays containing its row minima.

$f(x) = x^2$,

$$b[i] = \begin{cases} 2\lceil i/r \rceil - 1 & \text{if } 1 \leq i \leq i^*_{\lceil i/r \rceil}, \\ 2\lceil i/r \rceil & \text{if } i^*_{\lceil i/r \rceil} < i \leq m, \end{cases}$$

and $c[j] = -j$. By Property 1.7, A is Monge. Moreover, the row minima of A' are the row minima of the $\lfloor n/2 \rfloor$ $r \times 2$ subarrays depicted in Figure 2.4(b). Furthermore, only those entries in rows $(k-1)r+1$ through kr of A contain any information about i^*_k. Thus, roughly speaking, computing the row minima of A' is equivalent to locating the row minima in $\lfloor n/2 \rfloor$ unrelated $\lfloor 2m/n \rfloor \times 2$ Monge arrays. Since each of these smaller problems requires $\Omega(\lg(m/n))$ time to solve, this last observation gives the $\Omega(n \lg(m/n))$ lower bound on the time to compute the row minima of an $m \times n$ Monge array when $m > n$. ■

We also have the following theorem concerning the time complexity of computing the minimum entry overall in a two-dimensional Monge array.

Theorem 2.5 The minimum entry overall in an $m \times n$ Monge array A can be computed in $O(m+n)$ time. Moreover, this time bound is asymptotically optimal.

Proof The upper bound follows immediately from Theorem 2.4, since the minimum of A's m row minima can be computed in $O(m)$ time.

For the lower bound, let $C = \{c[j]\}$ denote any vector of n real numbers, and let $A = \{a[i,j]\}$ denote the $m \times n$ array given by $a[i,j] = c[j]$. By Property 1.6, A is Monge. Moreover, to compute the minimum entry in A, we must determine the minimum entry in C, which requires examining at least one entry in each column of A. Thus, computing the minimum entry of an $m \times n$ Monge array requires $\Omega(n)$ time. Furthermore, since computing the minimum entry in an $m \times n$ Monge array A is computationally equivalent to computing the minimum entry in A's $n \times m$ transpose A^T, the former problem also requires $\Omega(m)$ time. ■

2.2 On-Line Algorithms

In developing a linear-time algorithm for the concave least-weight subsequence problem, Wilber [Wil88] extended the SMAWK algorithm to a dynamic-programming setting. Specifically, he gave

an algorithm for the following *on-line* variant of the Monge-array column-minimization problem. Let $W = \{w[i,j]\}$ denote an $n \times n$ Monge array, where any entry of W can be computed in constant time. Furthermore, let $A = \{a[i,j]\}$ denote the $n \times n$ array defined by

$$a[i,j] = \begin{cases} E(i) + w[i,j] & \text{if } i < j, \\ +\infty & \text{if } i \geq j, \end{cases}$$

where $E(1)$ is given and for $1 < i \leq n$, $E(i)$ is some function that can be computed in constant time from the ith column minimum of A. Using the Mongeness of A, which follows from its definition, Wilber showed that the column minima of A (and hence $E(2), \ldots, E(n)$) can be computed in $O(n)$ time. This problem is called an on-line problem because certain inputs to the problem (i.e., entries of A) are available only after certain outputs of the problem (i.e., column minima of A) have been calculated.

In a subsequent paper dealing with the modified string-editing problem, Eppstein [Epp90] generalized Wilber's result, showing that the column minima of A can be computed in $O(n)$ time even when his algorithm is restricted to computing $E(1), \ldots, E(n)$ *in order*, i.e., $E(i)$ can be computed only after $E(1), \ldots, E(j-1)$ have been computed. This result is significant in that it allows the computation of $E(1), \ldots, E(n)$ to be interleaved with the computation of some other sequence $F(1), \ldots, F(n)$ such that $E(j)$ depends on $F(1), \ldots, F(j-1)$ and $F(j)$ depends on $E(1), \ldots, E(j)$.

Finally, Galil and K. Park [GP90], Klawe [Kla89], and Larmore and Schieber [LS91] independently extended Eppstein's result a step further, showing that as long as $a[i,j]$ can be computed in constant time once the first through ith column minima of A are known, the column minima of A can still be computed in $O(n)$ time.

In this thesis, we will make repeated use of this last result. Specifically, we will use Larmore and Schieber's algorithm, which we call the LIEBERalgorithm.

Theorem 2.6 (Larmore and Schieber [LS91]) The on-line row-minima problem for an $n \times n$ Monge array. can be solved in $\Theta(n)$ time. ■

2.3 Higher-Dimensional Monge Arrays

In this section, we will describe several algorithms for computing minima in higher-dimensional Monge arrays.

For $1 \leq i_1 \leq n_1$ and $2 \leq k \leq d$, let $i_k(i_1)$ denote the k-th index of the minimum entry in the plane of A corresponding to those entries whose first index is i_1.

Theorem 2.7 For $d \geq 2$, the plane minima of an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ can be computed in

$$O\left(\left(\sum_{k=1}^{d} n_k\right)\left(\prod_{k=1}^{d-2} \lg n_k\right)\right)$$

time.

Proof To compute the plane minima of A, we consider two cases.

If $d = 2$, we simply apply the $O(n_1 + n_2)$-time SMAWK algorithm of Theorem 2.4 to obtain the row minima of A.

On the other hand, if $d > 2$, we use a simple divide-and-conquer algorithm. Specifically, we begin by recursively computing the n_2 plane minima of the $(d-1)$-dimensional plane of A corresponding to those entries whose first index is $I_1 = \lceil n_1/2 \rceil$. (In other words, for all i_2 between 1 and n_2, we compute the minimum entry $a[I_1, i_2, i_3, \ldots, i_d]$ over all $(d-2)$-tuples $i_3, \ldots, i_d$ where $1 \leq i_k \leq n_k$ for $3 \leq k \leq d$.) We then compute the minimum of these minima in $O(n_2)$ additional time to obtain $i_k(I_1)$ for $2 \leq k \leq d$.

Now since the plane minima of A are monotone (by Property 1.11), we know that for $i_1 < I_1$, we must have $i_k(i_1) \leq i_k(I_1)$ for $2 \leq k \leq d$, and similarly for $i_1 > I_1$, we must have $i_k(i_1) \geq i_k(I_1)$ for $2 \leq k \leq d$. This means that we need only consider two smaller d-dimensional arrays, one $(I_1 - 1) \times i_2(I_1) \times \cdots \times i_d(I_1)$ and the other $(n_1 - I_1) \times (n_2 - i_2(I_1) + 1) \times \cdots \times (n_d - i_d(I_1) + 1)$, for the remaining plane minima of A. These minima we compute recursively.

If we let $T_d(n_1, n_2, \ldots, n_d)$ denote our algorithm's running time in computing the plane

minima of an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array, we then have

$$T_d(n_1, n_2, \ldots, n_d) = \begin{cases} O(n_1 + n_2) & \text{if } d = 2, \\ T_{d-1}(n_2, \ldots, n_d) & \text{if } d > 2 \text{ and } n_1 = 1, \\ T_{d-1}(n_2, \ldots, n_d) + O(n_2) \\ \quad + \max\limits_{\substack{n'_2, \ldots, n'_d \\ \text{s.t. } 1 \le n'_k \le n_k \\ \text{for } 2 \le k \le d}} \left\{ \begin{array}{l} T_d\left(\left\lceil \frac{n_1}{2} \right\rceil - 1, n'_2, \ldots, n'_d\right) \\ + T_d\left(\left\lfloor \frac{n_1}{2} \right\rfloor, n_2 - n'_2 + 1, \ldots, n_d - n'_d + 1\right) \end{array} \right\} \\ & \text{if } d > 2 \text{ and } n_1 > 1. \end{cases}$$

We will show by a double induction, first on d and then on n_1, that

$$T_d(n_1, n_2, \ldots, n_d) \le c_1 \left(\left(\sum_{k=1}^{d} n_k \right) - (d-2) \right) \left(\prod_{k=1}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) - c_2(n_1 + 1) ,$$

where c_1 and c_2 are constants independent of d, m, and n. Note that this is not the best bound possible — for example, $T_3(n_1, n_2, n_3)$ is actually $O(n_1 + (n_2 + n_3) \lg n_1)$, as the proof of Theorem 2.10 shows — but it is sufficient for our purposes.

We begin with the base case of $d = 2$. Since $T_2(n_1, n_2) \le c_3(n_1 + n_2)$ for some constant c_3, we need only choose $c_1 \ge c_2 + c_3$ to insure $T_2(n_1, n_2) \le c_1(n_1 + n_2) - c_2(n_1 + 1)$.

Now suppose

$$T_D(n_1, n_2, \ldots, n_D) \le c_1 \left(\left(\sum_{k=1}^{D} n_k \right) - (D-2) \right) \left(\prod_{k=1}^{D-2} (1 + \lfloor \lg n_k \rfloor) \right) - c_2(n_1 + 1) ,$$

for all $D < d$. This implies

$$T_d(n_1, n_2, \ldots, n_d) \le \begin{cases} c_1\left(\left(\sum_{k=2}^{d} n_k\right) - (d-3)\right)\left(\prod_{k=2}^{d-2}(1+\lfloor \lg n_k \rfloor)\right) \\ - c_2(n_2+1) & \text{if } n_1 = 1, \\ c_1\left(\left(\sum_{k=2}^{d} n_k\right) - (d-3)\right)\left(\prod_{k=2}^{d-2}(1+\lfloor \lg n_k \rfloor)\right) \\ - c_2(n_2+1) + c_4 n_2 \\ + \max_{\substack{n'_2,\ldots,n'_d \\ \text{s.t. } 1 \le n'_k \le n_k \\ \text{for } 2 \le k \le d}} \left\{ \begin{array}{l} T_d\left(\left\lceil \frac{n_1}{2} \right\rceil - 1, n'_2, \ldots, n'_d\right) \\ + T_d\left(\left\lfloor \frac{n_1}{2} \right\rfloor, n_2 - n'_2 + 1, \ldots, n_d - n'_d + 1\right) \end{array} \right\} \\ & \text{if } n_1 > 1, \end{cases}$$

where c_4 is some constant.

To show that this last recurrence implies

$$T_d(n_1, n_2, \ldots, n_d) \le c_1\left(\left(\sum_{k=1}^{d} n_k\right) - (d-2)\right)\left(\prod_{k=1}^{d-2}(1+\lfloor \lg n_k \rfloor)\right) - c_2(n_1+1)$$

(and thus complete our first inductive argument), we use a second induction, this time on n_1.

The base case of $n_1 = 1$ is easy. Since

$$T_d(1, n_2, \cdots, n_d) \le c_1\left(\left(\sum_{k=2}^{d} n_k\right) - (d-3)\right)\left(\prod_{k=2}^{d-2}(1+\lfloor \lg n_k \rfloor)\right) - c_2(n_2+1)$$

and $n_2 \ge 1$,

$$T_d(1, n_2, \cdots, n_d) \le c_1\left(1 + \left(\sum_{k=2}^{d} n_k\right) - (d-2)\right)\left(\prod_{k=2}^{d-2}(1+\lfloor \lg n_k \rfloor)\right) - 2c_2$$

follows immediately.

For the inductive step of our second inductive argument, suppose

$$T_d(N_1, n_2, \ldots, n_d)$$

$$\leq c_1 \left(N_1 + \left(\sum_{k=2}^{d} n_k \right) - (d-2) \right) (1 + \lfloor \lg N_1 \rfloor) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) - c_2(n_1 + 1)$$

for all $N_1 < n_1$. By our last recurrence for $T_d(n_1, n_2, \ldots, n_d)$, this implies

$$\begin{aligned}
& T_d(n_1, n_2, \ldots, n_d) \\
& \leq c_1 \left(\left(\sum_{k=2}^{d} n_k \right) - (d-3) \right) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) - c_2(n_2 + 1) + c_4 n_2 \\
& \quad + \max_{\substack{n'_2, \ldots, n'_d \\ \text{s.t. } 1 \leq n'_k \leq n_k \\ \text{for } 2 \leq k \leq d}} \left\{ \begin{array}{l}
c_1 \left(\left\lceil \frac{n_1}{2} \right\rceil - 1 + \left(\sum_{k=2}^{d} n'_k \right) - (d-2) \right) \\
\quad \left(1 + \left\lfloor \lg \left(\left\lceil \frac{n_1}{2} \right\rceil - 1 \right) \right\rfloor \right) \\
\quad \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n'_k \rfloor) \right) \\
- c_2 \left\lceil \frac{n_1}{2} \right\rceil \\
+ c_1 \left(\left\lfloor \frac{n_1}{2} \right\rfloor + \left(\sum_{k=2}^{d} (n_k - n'_k + 1) \right) - (d-2) \right) \\
\quad \left(1 + \left\lfloor \lg \left\lfloor \frac{n_1}{2} \right\rfloor \right\rfloor \right) \\
\quad \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg (n_k - n'_k + 1) \rfloor) \right) \\
- c_2 \left(\left\lfloor \frac{n_1}{2} \right\rfloor + 1 \right)
\end{array} \right. \\
& \leq c_1 \left(\left(\sum_{k=2}^{d} n_k \right) - (d-3) \right) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) + (c_4 - c_2) n_2 \\
& \quad + c_1 \left(n_1 - 1 + \left(\sum_{k=2}^{d} (n_k + 1) \right) - 2(d-2) \right) \left(1 + \left\lfloor \lg \frac{n_1}{2} \right\rfloor \right) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) \\
& \quad - c_2(n_1 + 1) \\
& \leq c_1 \left(\left(\sum_{k=2}^{d} n_k \right) - (d-3) \right) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) + (c_4 - c_2) n_2 \\
& \quad + c_1 \left(n_1 - 1 + \left(\sum_{k=2}^{d} n_k \right) - (d-3) \right) (\lfloor \lg n_1 \rfloor) \left(\prod_{k=2}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) \\
& \quad - c_2(n_1 + 1) \\
& \leq c_1 \left(n_1 + \left(\sum_{k=2}^{d} n_k \right) - (d-2) \right) \left(\prod_{k=1}^{d-2} (1 + \lfloor \lg n_k \rfloor) \right) - c_2(n_1 + 1)
\end{aligned}$$

$$+ (c_4 - c_2)n_2 .$$

Since $(c_4 - c_2)n_2 \leq 0$ provided we choose $c_2 \geq c_4$, this gives the desired result. ■

Corollary 2.8 For $d \geq 2$, the minimum entry in an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ can be computed in

$$O\left(\left(\sum_{k=1}^{d} n_k\right)\left(\prod_{k=1}^{d-2} \lg n_k\right)\right)$$

time.

For path- and cycle-decomposable Monge-composite arrays, we can do significantly better.

Theorem 2.9 The plane minima of an $n_1 \times \cdots \times n_d$ d-dimensional path-decomposable Monge-composite array A can be computed in

$$O\left(\sum_{k=1}^{d} n_k\right)$$

time.

Proof We will prove, by induction on d, that the plane minima of an $n_1 \times \cdots \times n_d$ d-dimensional path-decomposable Monge-composite array can be computed in at most

$$c_1 n_1 + \sum_{k=2}^{d-1} (c_1 + c_2) n_k + c_2 n_d + (d-1) c_3$$

time, where c_1, c_2, and c_3 are constants from the SMAWK algorithm's running time. (This is our induction hypothesis.)

The base case of $d = 2$ follows immediately from Theorem 2.4 of Section 2.1, as this theorem guarantees the existence of constants c_1, c_2, and c_3 such that the row minima of an $m \times n$ two-dimensional Monge array can be computed in at most $c_1 m + c_2 n + c_3$ time.

For $d > 2$, we assume that the induction hypothesis holds for all lower-dimensional path-decomposable Monge-composite arrays. Since $A = \{a[i_1, \ldots, i_d]\}$ is path-decomposable, we can

write

$$a[i_1,\ldots,i_d] = \sum_{k=1}^{d-1} w_{k,k+1}[i_k,i_{k+1}] ,$$

where for $1 \le k < d$, $W_{k,k+1} = \{w_{k,k+1}[i_k,i_{k+1}]\}$ is an $n_k \times n_{k+1}$ Monge array. Now consider the $n_2 \times \cdots \times n_d$ $(d-1)$-dimensional array $B = \{b[i_2,\ldots,i_d]\}$ where

$$b[i_2,\ldots,i_d] = \sum_{k=2}^{d-1} w_{k,k+1}[i_k,i_{k+1}] .$$

By the induction hypothesis, we can find the plane minima of B in at most

$$c_1 n_2 + \sum_{k=3}^{d-1}(c_1+c_2)n_k + c_2 n_d + (d-2)c_3$$

time. Since the minimum entry in the plane of A corresponding to a particular value I_1 of the first index is just

$$\min_{i_2}\left\{w_{1,2}[I_1,i_2] + \min_{i_3,\ldots,i_d}\left\{\sum_{k=2}^{d-1} w_{k,k+1}[i_k,i_{k+1}]\right\}\right\} ,$$

we need only find the row minima in the sum of $W_{1,2}$ and the appropriate two-dimensional extension of the vector of B's plane minima. Since this sum is a Monge array, we can find its row minima in at most $c_1 n_1 + c_2 n_2 + c_3$ time. This gives the entire algorithm a running time bounded by

$$\begin{aligned} &\left(c_1 n_2 + \sum_{k=3}^{d-1}(c_1+c_2)n_k + c_2 n_d + (d-2)c_3\right) + (c_1 n_1 + c_2 n_2 + c_3) \\ &= c_1 n_1 + \sum_{k=2}^{d-1}(c_1+c_2)n_k + c_2 n_d + (d-1)c_3 \end{aligned}$$

which proves the inductive hypothesis for d. ■

Theorem 2.10 The plane minima of an $n_1 \times \cdots \times n_d$ d-dimensional cycle-decomposable Monge-composite array A can be computed in

$$O\left(n_1 + \left(\sum_{k=2}^{d} n_k\right)\lg n_1\right)$$

time.

Proof Since $A = \{a[i_1, \ldots, i_d]\}$ is cycle-decomposable, we can write

$$a[i_1, \ldots, i_d] = \sum_{k=1}^{d-1} w_{k,k+1}[i_k, i_{k+1}] + w_{d,1}[i_d, i_1] ,$$

where for $1 \le k < d$, $W_{k,k+1} = \{w_{k,k+1}[i_k, i_{k+1}]\}$ is an $n_k \times n_{k+1}$ Monge array and where $W_{d,1} = \{w_{d,1}[i_d, i_1]\}$ is an $n_d \times n_1$ Monge array. Now consider any $(d-1)$-dimensional plane $A_{I_1} = \{a_{I_1}[i_2, \ldots, i_d]\}$ of A, corresponding to a fixed value I_1 of A's first index. If we let $W'_{2,3} = \{w'_{2,3}[i_2, i_3]\}$ denote the $n_2 \times n_3$ array where $w'_{2,3}[i_2, i_3] = w_{2,3}[i_2, i_3] + w_{1,2}[I_1, i_2]$, then $W'_{2,3}$ is Monge, since it is the sum of a Monge array and a two-dimensional extension of a one-dimensional vector. Similarly, if we let $W'_{d-1,d} = \{w'_{d-1,d}[i_{d-1}, i_d]\}$ denote the $n_{d-1} \times n_d$ array where $w'_{d-1,d}[i_{d-1}, i_d] = w_{d-1,d}[i_{d-1}, i_d] + w_{d,1}[i_d, I_1]$, then $W'_{d-1,d}$ is also Monge. This implies A_{I_1} is a path-decomposable Monge-composite array, since

$$a_{I_1}[i_2, i_3, \ldots, i_d] = w'_{2,3}[i_2, i_3] + \sum_{k=3}^{d-1} w_{k,k+1}[i_k, i_{k-1}] + w'_{d-1,d}[i_{d-1}, i_d] .$$

Thus, by Theorem 2.10, we can compute the plane minima of A_{I_1} in $O(\sum_{k=2}^{d} n_k)$ time. Furthermore, we can compute the minimum of these minima in $O(n_2)$ additional time. This means we can compute any plane minimum of A in at most

$$c_1 \sum_{k=2}^{d} n_k + c_2$$

time, where c_1 and c_2 are constants.

To compute all the plane minima of A, we use the divide-and-conquer approach of Theorem 2.7. Specifically, we begin by computing the minimum entry in the plane corresponding to $I_1 = \lceil n_1/2 \rceil$, which, by the preceding discussion, requires at most

$$c_1 \sum_{k=2}^{d} n_k + c_2$$

time. This gives us $i_k(I_1)$ for $2 \le k \le d$. Now since A is Monge (and thus monotone), we know that for $i_1 < I_1$, we must have $i_k(i_1) \le i_k(I_1)$ for $2 \le k \le d$, and similarly for $i_1 > I_1$, we must have $i_k(i_1) \ge i_k(I_1)$ for $2 \le k \le d$. Thus, we need only consider two smaller arrays, one

$(I_1-1)\times i_2(I_1)\times\cdots\times i_d(I_1)$ and the other $(n_1-I_1)\times(n_2-i_2(I_1)+1)\times\cdots\times(n_d-i_d(I_1)+1)$, in searching for the remaining plane minima. If we compute the plane minima of these two subarrays recursively, then we obtain the following recurrence for the time $T(n_1,n_2,\ldots,n_d)$ to compute the plane minima of an $n_1\times n_2\times\cdots\times n_d$ cycle-decomposable Monge-composite array:

$$
\begin{aligned}
&T(n_1,n_2,\ldots,n_d)\\
&\quad\le c_1\left(\sum_{k=2}^{d} n_k\right)+c_2\\
&\qquad+\max_{\substack{1\le n_k'\le n_k\\ 2\le k\le d}}\left\{T\left(\left\lceil\frac{N_1}{2}\right\rceil-1,n_2',\ldots,n_d'\right)+T\left(\left\lfloor\frac{N_1}{2}\right\rfloor,n_2-n_2'+1,\ldots,n_d-n_d'+1\right)\right\},
\end{aligned}
$$

where

$$T(1,n_2,\ldots,n_d)\le c_1\sum_{k=2}^{d} n_k+c_2 .$$

To prove that $T(n_1,\ldots,n_d)$ has the stated asymptotics, we will show, by induction on n_1, that

$$T(n_1,n_2,\ldots,n_d)\le c_1\left(\sum_{k=2}^{d}(n_k-1)\right)(1+\lfloor\lg n_1\rfloor)+(c_1(d-1)+c_2)n_1 .$$

(This is our inductive hypothesis.) The base case of $n_1=1$ is easy, since

$$T(1,n_2,\ldots,n_d)\le c_1\left(\sum_{k=2}^{d} n_k\right)+c_2=c_1\left(\sum_{k=2}^{d}(n_k-1)\right)+c_1(d-1)+c_2 .$$

Now assume the inductive hypothesis holds for all values of $n_1<N_1$. Then

$$
\begin{aligned}
&T(N_1,n_2,\ldots,n_d)\\
&\quad\le c_1\left(\sum_{k=2}^{d}(n_k-1)\right)+c_1(d-1)+c_2\\
&\qquad+c_1\left(\sum_{k=2}^{d}(n_k'-1)\right)\left(1+\left\lfloor\lg\left(\left\lceil\frac{N_1}{2}\right\rceil-1\right)\right\rfloor\right)+(c_1(d-1)+c_2)\left(\left\lceil\frac{N_1}{2}\right\rceil-1\right)\\
&\qquad+c_1\left(\sum_{k=2}^{d}(n_k-n_k')\right)\left(1+\left\lfloor\lg\left\lfloor\frac{N_1}{2}\right\rfloor\right\rfloor\right)+(c_1(d-1)+c_2)\left\lfloor\frac{N_1}{2}\right\rfloor\\
&\quad\le c_1\left(\sum_{k=2}^{d}(n_k-1)\right)\left(2+\left\lfloor\lg\left\lfloor\frac{N_1}{2}\right\rfloor\right\rfloor\right)+(c_1(d-1)+c_2)N_1
\end{aligned}
$$

$$\leq c_1 \left(\sum_{k=2}^{d} (n_k - 1) \right) (1 + \lfloor \lg N_1 \rfloor) + (c_1(d-1) + c_2) N_1 .$$

Thus, the inductive hypothesis also holds for $n_1 = N_1$. ■

Maximization is harder than minimization when it comes to higher-dimensional Monge arrays, as the following theorem shows.

Theorem 2.11 For $d \geq 2$, computing the maximum entry in an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ requires

$$\Omega \left(\frac{\prod_{k=1}^{d} n_k}{\sum_{k=1}^{d} (n_k - 1)} \right)$$

time.

Proof For $d \leq s \leq n_1 + n_2 + \cdots + n_d$, let $A_s = \{a_s[i_1, i_2, \ldots, i_d]\}$ denote the $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional array where

$$a[i_1, i_2, \ldots, i_d] = -\left(\left(\sum_{k=1}^{d} i_k \right) - s \right)^2 .$$

Using Definitions 1.2 and 1.6, we will show that A_s is Monge. Consider any pairs of dimensions ℓ and ℓ' such that $1 \leq \ell < \ell' \leq d$ and any d-tuple $i_1, i_2, \ldots, i_d$ of indices such that $1 \leq i_\ell < n_\ell$, $1 \leq i_{\ell'} < n_{\ell'}$, and $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$, $k \neq \ell, \ell'$. Clearly,

$$\begin{aligned}
& a[i_1, \ldots, i_\ell, \ldots, i_{\ell'}, \ldots, i_d] + a[i_1, \ldots, i_\ell + 1, \ldots, i_{\ell'} + 1, \ldots, i_d] \\
& - a[i_1, \ldots, i_\ell, \ldots, i_{\ell'} + 1, \ldots, i_d] - a[i_1, \ldots, i_\ell + 1, \ldots, i_{\ell'}, \ldots, i_d] \qquad (2.1) \\
&= (i_1 + \cdots + i_d - s + 1)^2 + (i_1 + \cdots + i_d - s + 1)^2 \\
&\quad - (i_1 + \cdots + i_d - s)^2 - (i_1 + \cdots + i_d - s + 2)^2 \\
&= -2 ,
\end{aligned}$$

which implies A_s is Monge.

Now consider any entry $a[i_1, i_2, \ldots, i_d]$ such that $i_1 + i_2 + \cdots + i_d = s$. This entry's value is 0, and it is clearly a maximum entry of A_s. Moreover, if we replace this entry with a 1, then it becomes A_s's unique maximum entry. Note that this substitution does not affect A_s's Mongeness, since the value of (2.1) does not changes by more than 1 for any pair of dimensions and d-tuple of indices. Since we can change any entry $a[i_1, i_2, \ldots, i_d]$ such that $i_1+i_2+\cdots+i_d = s$ to a 1, finding the maximum entry in A_s once one entry of this form has been changed to a 1 requires looking at all of these entries. In other words, the number of entries of $a[i_1, i_2, \ldots, i_d]$ such that $i_1 + i_2 + \cdots + i_d = s$ is a lower bound on the time to compute this Monge array's maximum entry.

To bound the number of d-tuples $i_1, i_2, \ldots, i_d$ such that $i_1+i_2+\ldots+i_d = s$ (and $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$) over all s such that $d \leq s \leq n_1+n_2+\cdots+n_d$, we use an averaging argument. First note that there are $n_1 n_2 \cdots n_d$ total d-tuples $i_1, i_2, \ldots, i_d$ such that $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$. Furthermore, there are $n_1 + n_2 + \cdots + n_d - d$ possible values for s. Thus, there exists an s such that there are

$$\frac{n_1 n_2 \cdots n_d}{n_1 + n_2 + \cdots + n_d - d}$$

d-tuples $i_1, i_2, \ldots, i_d$ such that $i_1 + i_2 + \cdots + i_d = s$ and $1 \leq i_k \leq n_k$ for $1 \leq k \leq d$. This gives the desired result. ∎

Corollary 2.12 For $d \geq 2$, computing the plane maxima of an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ requires

$$\Omega\left(\frac{\prod_{k=1}^{d} n_k}{\sum_{k=1}^{d}(n_k - 1)}\right)$$

time.

Theorem 2.13 For $d \geq 2$, the plane maxima of an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional Monge array $A = \{a[i_1, i_2, \ldots, i_d]\}$ can be computed in

$$O\left((n_1 + n_2) \prod_{k=3}^{d} n_k\right)$$

time.

Proof We begin by decomposing A into $n_3 \cdots n_d$ $n_1 \times n_2$ two-dimensional arrays. The row maxima in these subarrays can then be computed in $O((n_1 + n_2)n_3 \cdots n_d)$ total time using the SMAWK algorithm. The maximum entry in any plane of A corresponding to a fixed value of A's first index is then the maximum of $n_3 \cdots n_d$ of these row maxima. This implies the plane maxima of A can be computed from the $n_1 n_3 \cdots n_d$ total row maxima in $O(n_1 n_3 \cdots n_d)$ additional time, which gives the desired result. ■

2.4 Partial Monge Arrays

In this section, we briefly mention several algorithms for computing minima in partial Monge arrays.

We begin by defining an inverse for Ackermann's function. For $i \geq -1$,

$$L_i(n) = \begin{cases} n/2 & \text{if } i = -1, \\ \min\{s : L_{i-1}^s(n) \leq 1\} & \text{if } i \geq 0. \end{cases}$$

(For any function $f(\cdot)$ and any integer $s \geq 0$, $f^s(\cdot)$ denotes the function obtained by composing s copies of $f(\cdot)$, so that $f^4(n) = f(f(f(f(n))))$, for example.) Thus, $L_0(n) = \lceil \lg n \rceil$, and $L_1(n)$ is roughly $\lg^* n$.

In terms of $L_{-1}(\cdot), L_0(\cdot), L_1(\cdot), \ldots$, we then have

$$\alpha(n) = \min\{s : L_s(n) \leq s\} .$$

The function $\alpha(\cdot)$ is *very* slowly growing.

Theorem 2.14 (Klawe and Kleitman [KK90]) The off-line row-minimization problem for an $m \times n$ staircase-Monge array can be solved in $O(n\alpha(m) + m)$ time. ■

Theorem 2.15 (Klawe and Kleitman [KK90]) The on-line row-minimization problem for an $n \times n$ staircase-Monge array can be solved in $O(n\alpha(m) + m)$ time. ■

Chapter 3

Selection and Sorting Algorithms

In the previous chapter, we considered several minimization and maximization problems involving Monge arrays. In this chapter, we turn our attention to two more types of comparison problems, selection problems and sorting problems.

The basic selection and sorting problems may be defined as follows. Given a set S of n distinct values $a_1, \ldots, a_n$, the rank $r(b, S)$ in S of some value b is the number of $a_j \in S$ such that $a_j \leq b$. In other words,

$$r(b, S) = |\{j : 1 \leq j \leq n \text{ and } a_j \leq b\}| .$$

Note that b need not be a member of S. Clearly, $1 \leq r(a_i, S) \leq n$ for all $a_i \in S$; moreover, since the elements $a_1, \ldots, a_n$ are distinct, the ranks $r(a_1, S), \ldots, r(a_n, S)$ are distinct as well.

In terms of ranks, the selection problem for S is that of computing the unique i such that $r(a_i, S) = k$, i.e., computing the kth smallest element of S. (The integer k is given as part of the input along with S.) Similarly, the sorting problem for S is that of computing $r(a_i, S)$ for all i between 1 and n, i.e., sorting the n elements in S.

For arbitrary values $a_1, \ldots, a_n$, the selection and sorting problems are well understood: the general selection problem can be solved in $\Theta(n)$ time [BFP+73], and the general sorting problem in $\Theta(n \lg n)$ time (see [Knu73], for example). By using the special structure of Monge arrays, however, it is possible to obtain significantly better results for certain selection and sorting problems involving two-dimensional Monge arrays than are possible with the classical selection

and sorting algorithms. With this goal in mind, we now proceed to an overview of the chapter.

In Section 3.1, we consider the problem of computing the kth smallest entry in each row of a two-dimensional Monge array A. We call this problem the *row-selection* problem for A. In Subsection 3.1.1, we show that the row-selection problem for an $m \times n$ Monge array A can be solved in $O(k(m+n))$ time. For small values of k, this bound represents a significant improvement over the naive $O(mn)$ bound obtained by applying the linear-time selection algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+73] to each row of the array. In Subsection 3.1.2, we give another row-selection algorithm; this algorithm solves the row-selection problem for an $m \times n$ Monge array A in $O((\sqrt{m}\lg m)(n\lg n) + m\lg n)$ time. For large values of k, this bound represents a significant improvement over both the naive $O(mn)$ bound and the $O(k(m+n))$ bound obtained in the previous subsection. Note that computing the kth largest entry in each row of A is no harder than computing the kth smallest entry in each row, just as computing a maximum entry in each row is no harder than computing a minimum entry in each row; we need only negate the entries of A and reverse the ordering of its columns to convert back and forth between the two problems.

In Section 3.2, we turn to the problem of computing the kth smallest entry overall in an $m \times n$ Monge array A, i.e., the kth smallest of the mn entries of A. We call this problem the *array-selection* problem for A. We show that the array-selection problem for an $m \times n$ Monge array A can be solved in $O(m+n+k\lg(mn/k))$ time. For small values of k, this bound represents a significant improvement over the naive $O(mn)$ bound obtained by applying the linear-time selection algorithm of Blum et al. [BFP+73]. Again note that computing the kth largest entry overall in A is no harder than computing the kth smallest entry overall.

The subject of Section 3.3 is the problem of sorting the rows of a Monge array A. We call this problem the *row-sorting* problem for A. We show that the row-sorting problem for an $m \times n$ Monge array A can be solved in $O(mn)$ time if $m \geq n$ and in $O(mn(1+\lg(n/m)))$ time if $m < n$. This bound represents an improvement over the naive $O(mn\lg n)$ bound obtained by applying a general sorting algorithm to each row of A.

Section 3.4 focuses on the problem of sorting all the entries of a Monge array A. We call this problem the *array-sorting* problem for A. We show that the array-sorting problem for an $m \times n$ Monge array A requires $\Omega(mn\lg t)$ comparisons (and thus $\Omega(mn\lg t)$ time), where

Problem	*Time*	*Theorem*
row selection	$O(kn)$	3.2
	$O(n^{3/2}\lg^2 n)$	3.4
array selection	$O(n + k\lg(n^2/k)$	3.5
row sorting	$O(n^2)$	3.6
array sorting	$\Theta(n^2 \lg n)$	3.7

Table 3.1: Results for selection and sorting in an $n \times n$ Monge array. The selection results are for computing the kth smallest (or kth largest) entry in each row or overall.

$t = \min\{m, n\}$. Thus, for $m = \Theta(n)$, the Mongeness of A does not make sorting its entries any easier than sorting mn arbitrary values.

Finally, in Section 3.5, we conclude with some open problems.

Table 3.1 summarizes the algorithms and lower bounds given in this chapter for selection and sorting in two-dimensional Monge arrays. All of these results are from a paper cowritten with Kravets [KP91], except for the second row-selection algorithm, which was developed in collaboration with Mansour, Schieber, and Sen [MPSS91].

3.1 Row Selection

In this section, we give two very different algorithms for computing the kth smallest entry in each row of an $m \times n$ Monge array. The first algorithm represents joint work with Kravets [KP91]. It uses the SMAWK algorithm of Section 2.1 and runs in $O(k(m+n))$ time. The second algorithm, on the other hand, does not involve the SMAWK algorithm, and its $O((\sqrt{m}\lg m)(n\lg n) + m\lg n)$ running time is independent of k. This latter algorithm represents joint work with Mansour, Schieber, and Sen [MPSS91].

Neither of these two algorithms dominates the other for all m, n, and k; rather, the first algorithm is superior for small k, while the second algorithm is better for large k. In particular, if $m = \Theta(n)$, then the first algorithm is more efficient for $k = o(\sqrt{n}\lg^2 n)$, while the second algorithm dominates if $k = \omega(\sqrt{n}\lg^2 n)$.

3.1.1 Row Selection When k is Small

Given an $m \times n$ Monge array $A = \{a[i,j]\}$ and an integer k between 1 and n, this subsection's row-selection algorithm computes the kth smallest entry in each row of A in $O(k(m+n))$ time. The algorithm achieves this time bound by combining two previous results — the SMAWK algorithm of Section 2.1 and a selection algorithm due to Frederickson and Johnson [FJ82] — with yet another property of Monge arrays.

We begin by describing Frederickson and Johnson's result. In [FJ82], they considered the following problem: given an $m \times n$ array $B = \{a[i,j]\}$ whose rows are sorted in ascending order (i.e., $b[i,1] \leq b[i,2] \leq \cdots \leq b[i,n]$ for $1 \leq i \leq m$), compute the kth smallest of the mn entries of B. (This problem is just the array-selection problem for the not-necessarily-Monge array B.) Applying the linear-time selection algorithm of [BFP+73] to either all mn entries of B if $k \geq n$ or the mk entries in columns 1 through k of B if $k < n$ solves this problem in $O(mt)$ time, where $t = \min\{k, n\}$. However, Frederickson and Johnson obtained a significantly better result in [FJ82], which we summarize in the following theorem.

Theorem 3.1 (Frederickson and Johnson [FJ82]) The kth smallest entry in an $m \times n$ array B whose rows are sorted in ascending order can be computed in $O(m + s\lg(k/s))$ time, where $s = \min\{k, m\}$. ∎

This theorem is actually more general than its needs to be for our purposes, as only the $m = O(k)$ special case of Frederickson and Johnson's selection problem is relevant to our row-selection algorithm. Specifically, we require only an $O(k)$-time algorithm for computing the kth smallest element in $O(k)$ sorted lists of length k, which Theorem 3.1 provides.

The property of Monge arrays linking the SMAWK algorithm and Frederickson and Johnson's result is Property 1.10, which states that if $A = \{a[i,j]\}$ is an $m \times n$ Monge array such that $m \geq n$ and each column of A contains at least one row minimum, then each row of A is bitonic. More precisely, for $1 \leq i \leq m$,

$$a[i,1] > \cdots > a[i,c(i)-1] > a[i,c(i)]$$

and

$$a[i,c(i)] < a[i,c(i)+1] < \cdots < a[i,n],$$

where $c(i)$ denotes the column of A containing the minimum entry in row i of A.

With these preliminaries behind us, we can now prove the following theorem.

Theorem 3.2 The kth smallest entry in each row of an $m \times n$ Monge array $A = \{a[i,j]\}$ can be computed in $O(k(m+n))$ time.

Proof Our algorithm for computing the kth smallest entry in each row of A has two parts. First, we extract from A a sequence of k distinct m-row subarrays $B_1, \ldots, B_k$. The first subarray B_1 consists of those columns of A that contain row minima of A. If we let A_1 denote the m-row subarray of A consisting of those columns of A not in B_1, then B_2 consists of those columns of A_1 that contain row minima of A_1. In general, for $i \geq 1$, B_i consists of those columns of A_{i-1} that contain row minima of A_{i-1} (where we define $A_0 = A$), and A_i consists of those columns of A_{i-1} that do not contain row minima of A_{i-1}, as is suggested in Figure 3.1. (Equivalently, A_i consists of those columns of A not in any of $B_1, \ldots, B_i$.)

Using the SMAWK algorithm, we can compute B_i and A_i (or, more precisely, the columns forming these arrays) from A_{i-1}. Thus, k applications of the SMAWK algorithm give $B_1, \ldots, B_k$ in $O(k(m+n))$ total time.

Now for $1 \leq i \leq k$, the definition of B_i implies that each column of B_i contains at least one row minimum; thus, by Property 1.10, the rows of B_i are bitonic. Furthermore, if an entry is among the k smallest entries in some row of A, then the entry must be contained in one of $B_1, \ldots, B_k$. Thus, to compute the the kth smallest entry in row i of A, we merely need to compute the kth smallest element in the $2k$ sorted lists associated with row i. (Each B_i contributes two sorted lists, the first consisting of those entries in the ith row of B_i to the right of the ith row's minimum and the second consisting of those entries to the minimum's left.) By Theorem 3.1, this element can be identified in $O(k)$ time. Since A contains m rows, the total time for this second part of the algorithm is $O(km)$, which gives the entire row-selection algorithm a running time of $O(k(m+n))$. ■

As a final observation, we note that Frederickson and Johnson's selection algorithm computes not only the kth smallest entry in B, the $m \times n$ array whose rows are sorted in ascending order, but also the first through $(k-1)$st smallest entries. (These entries are not given in sorted

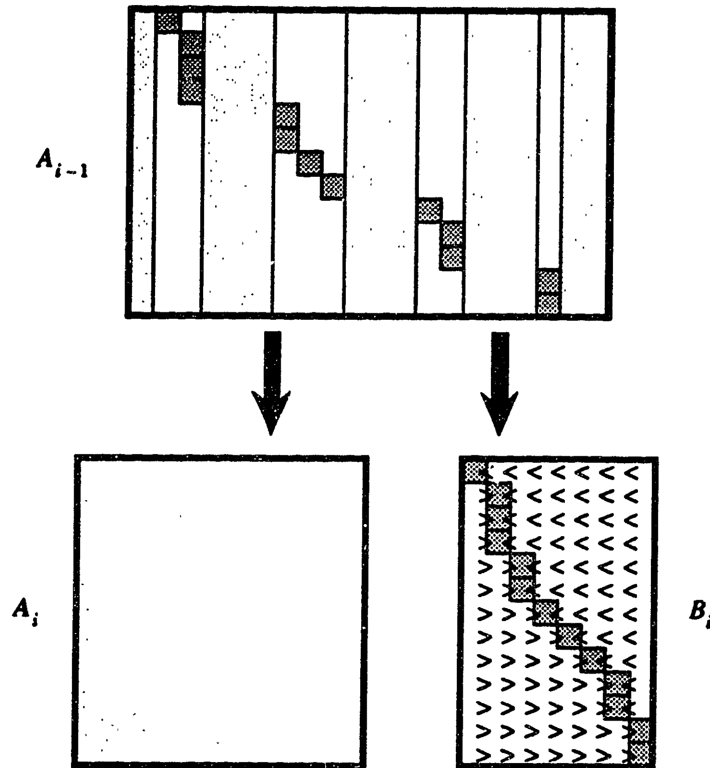

Figure 3.1: For $i \geq 1$, the columns of A_{i-1} are divided between B_i and A_i: B_i gets those columns of A_{i-1} containing row minima, while A_i gets those columns without row minima. (The darkly shaded squares represent the row minima of A_{i-1}; the lightly shaded regions indicate those columns assigned to A_i.) Since each column of B_i contains a row minimum, its rows are bitonic.

order, however.) Thus, this subsection's row-selection algorithm is easily modified to output the first through kth smallest entries in each row of the $m \times n$ Monge array A; moreover, this modification does not affect the asymptotics of our algorithm's running time. We will use this observation in Section 3.2.

3.1.2 Row Selection When k is Large

For larger values of k, better results can be obtained for the row-selection problem using an algorithm first described in [MPSS91]. This algorithm's running time is $O((\sqrt{m}\lg m)(n\lg n) + m\lg n)$, which is independent of k.

At the heart of this subsection's algorithm is the notion of left and right ranks, which we briefly mentioned in Section 1.1. Given a set S of n distinct values $a_1, \ldots, a_n$, the left rank

$L(a_i, S)$ of a_i in S is the number of a_j such that a_j is smaller than a_i *and* $j < i$, i.e.,

$$L(a_i, S) = |\{j : 1 \leq j < i \text{ and } a_j < a_i\}| .$$

Similarly, the right rank $R(a_i, S)$ of a_i in S is the number of a_j such that a_j is at most a_i *and* $j \geq i$, i.e.,

$$R(a_i, S) = |\{j : i \leq j \leq n \text{ and } a_j \leq a_i\}| .$$

Clearly, the rank $r(a_i, S)$ of a_i in S is $L(a_i, S) + R(a_i, S)$ for all i. Moreover, computing the left and right ranks of $a_1, a_2, \ldots, a_n$ is no harder than computing their ranks (i.e., sorting S), as the following lemma demonstrates.

Lemma 3.3 Given a set S of n distinct values $a_1, \ldots, a_n$, we can compute $L(a_i, S)$ and $R(a_i, S)$ for $1 \leq i \leq n$ in $O(n \lg n)$ time.

Proof To compute the $L(a_i, S)$ and $R(a_i, S)$ in $O(n \lg n)$ time, we use a divide-and-conquer approach reminiscent of mergesort. We begin by partitioning S into two subsets S' and S'' so that S' contains the first $\lfloor n/2 \rfloor$ values $a_1, \ldots, a_{\lfloor n/2 \rfloor}$ and S'' contains the remaining values $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$. We then recursively compute the $L(a_i, S')$ and $R(a_i, S')$ for $1 \leq i \leq \lfloor n/2 \rfloor$ and the $L(a_i, S'')$ and $R(a_i, S'')$ for $\lfloor n/2 \rfloor < i \leq n$. Then for $1 \leq i \leq \lfloor n/2 \rfloor$, $L(a_i, S) = L(a_i, S')$, and for $\lfloor n/2 \rfloor < i \leq n$, $R(a_i, S) = R(a_i, S'')$. Furthermore, for $1 \leq i \leq \lfloor n/2 \rfloor$, $R(a_i, S) = R(a_i, S') + r(a_i, S'')$, and for $\lfloor n/2 \rfloor < i \leq n$, $L(a_i, S) = L(a_i, S'') + r(a_i, S')$. Since we know the sorted order of $a_1, \ldots, a_{\lfloor n/2 \rfloor}$ and the sorted order of $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$, we can merge these two lists in $O(n)$ time, thereby obtaining $r(a_i, S'')$ for $1 \leq i \leq \lfloor n/2 \rfloor$ and $r(a_i, S')$ for $\lfloor n/2 \rfloor < i \leq n$. Thus, we obtain the following (familiar) recurrence for the time $T(n)$ to compute $L(a_1, S), L(a_2, S), \ldots, L(a_n, S)$ and $R(a_1, S), R(a_2, S), \ldots, R(a_n, S)$:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2, \\ O(1) & \text{if } n = 1. \end{cases}$$

As the solution for this recurrence is $T(n) = O(n \lg n)$, we are done. ∎

As we observed in Section 1.1, the left and right ranks of Monge-array entries are highly structured. In particular, suppose $A = \{a[i, j]\}$ is an $m \times n$ Monge array. Furthermore,

let $L(i,j)$ denote the left rank of $a[i,j]$ in row i of A. (In terms of our previous notation, $L(i,j) = L(a[i,j], S_i)$, where S_i is the set of entries in A's ith row.) Similarly, let $R(i,j)$ denote the right rank of $a[i,j]$ in row i of A, and let $r(i,j)$ denote the rank of $a[i,j]$ in row i of A. Property 1.8 of two-dimensional Monge arrays tells us that for $1 \le j \le n$,

$$L(1,j) \ge L(2,j) \ge \cdots \ge L(m,j)$$

and

$$R(1,j) \le R(2,j) \le \cdots \le R(m,j)\,.$$

Using this property, the identity $r(i,j) = L(i,j) + R(i,j)$, and Lemma 3.3, we will now prove the following theorem.

Theorem 3.4 The kth smallest entry in each row of an $m \times n$ Monge array $A = \{a[i,j]\}$ can be computed in $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$ time.

Proof If $m \le 4$, we use the linear-time selection algorithm of Blum et al. [BFP+73] to obtain the kth smallest entry in each row in $O(n)$ time. Otherwise, we use the following divide-and-conquer approach.

We begin by partitioning A into x subarrays $A_1, \cdots, A_x$, where x is a parameter of our algorithm in the range $1 < x \le m$. (We will later set $x = \lceil \sqrt{m} \rceil$ to minimize our algorithm's running time, but for now, it is simpler to think of x as a parameter.) In particular, for $1 \le t \le x$, A_t consists of rows $(t-1)\lceil m/x \rceil + 1$ through $t\lceil m/x \rceil$ of A. (The last subarray A_x may actually contain fewer than $\lceil m/x \rceil$ rows, but for simplicity, we will ignore this detail.) In order to simplify the notation, we set

$$\lambda_t(x) = (t-1)\lceil m/x \rceil + 1\,;$$

thus, A_t includes rows $\lambda_t(x)$ through $\lambda_{t+1}(x) - 1$ of A.

We then compute left and right ranks in the first row of each A_t. Specifically, for $1 \le t \le x$ and $1 \le j \le n$, we compute $L(\lambda_t(x), j)$ and $R(\lambda_t(x), j)$. By Lemma 3.3, this computation can be performed in $O(n \lg n)$ time per row (i.e., $O(xn \lg n)$ total time).

Figure 3.2: Suppose $\lambda_t(x) \le i < \lambda_{t+1}(x)$ (i.e., row i of A is in A_t) and $1 \le j \le n$. Since A is Monge, we must have $L(\lambda_t(x), j) \ge L(i,j) \ge L(\lambda_{t+1}(x), j)$ and $R(\lambda_t(x), j) \le R(i,j) \le R(\lambda_{t+1}(x), j)$. Thus, the rank $r(i,j)$ of $a[i,j]$ in row i of A must lie between $L(\lambda_{t+1}(x), j) + R(\lambda_t(x), j)$ and $L(\lambda_t(x), j) + R(\lambda_{t+1}(x), j)$.

Now, by Property 1.8, for $1 \le t \le x$, $\lambda_t(x) \le i < \lambda_{t+1}(x)$, and $1 \le j \le n$, we have

$$L(\lambda_t(x), j) \;\ge\; L(i,j) \;\ge\; L(\lambda_{t+1}(x), j)$$

and

$$R(\lambda_t(x), j) \;\le\; R(i,j) \;\le\; R(\lambda_{t+1}(x), j)\,.$$

(To handle the last row, we define $L(\lambda_{x+1}(x), j) = 0$ and $R(\lambda_{x+1}(x), j) = n - j + 1$ for $1 \le j \le n$.) These bounds are illustrated in Figure 3.2. Since $r(i,j) = L(i,j) + R(i,j)$, this gives us lower and upper bounds on the rank of every entry $a[i,j]$ in the jth column of A_t; specifically, for $\lambda_t(x) \le i < \lambda_{t+1}(x)$, we must have

$$\alpha(t,j) \;\le\; r(i,j) \;\le\; \beta(t,j)$$

where

$$\alpha(t,j) \;=\; L(\lambda_{t+1}(x), j) + R(\lambda_t(x), j)$$

and

$$\beta(t,j) \;=\; L(\lambda_t(x), j) + R(\lambda_{t+1}(x), j)\,.$$

The above bounds allow us to reduce the size of our row-selection problem. In particular, for all t in the range $1 \le t \le x$ and all j in the range $1 \le j \le n$, we can compute $\alpha(t,j)$ and $\beta(t,j)$ and then delete column j from A_t if $\alpha(t,j) > k$ or $\beta(t,j) < k$, as in both these cases, no entry in column j of A_t can have rank k in its row. Deleting columns in this manner takes $O(xn)$ time, since $A_1, \ldots, A_x$ contain xn total columns.

Now let A'_t denote the subarray of A_t consisting of those columns not deleted, i.e., those columns j such that

$$\alpha(t,j) \;\le\; k \;\le\; \beta(t,j)\,.$$

The kth smallest entry in each row of A_t is the k_tth smallest entry in each row of A'_t, where

$$k_t \;=\; k - |\{j \,:\, 1 \le j \le n \text{ and } \beta(t,j) < k\}|\,.$$

Thus, since $A'_1, A'_2, \ldots, A'_x$ are all Monge (by Property 1.2), we need only recursively solve x smaller row-selection problems to obtain the kth smallest entry in each row of A. This completes the description of our row-selection algorithm.

To analyze the above this algorithm's running time, we must bound the number of columns eliminated from $A_1, A_2, \ldots, A_x$. Intuitively, the more columns eliminated, the smaller the problems that remain and the better the running time of our algorithm.

Before proving any bounds, we first introduce a little notation. Let n_t denote the number of columns in A'_t, i.e.,

$$n_t \;=\; |\{j \,:\, \alpha(t,j) \le k \le \beta(t,j)\}|\,.$$

In terms of this new notation, our algorithm spends $O(xn \lg n)$ time reducing one row-selection problem of size $m \times n$ to x row-selection problems such that the tth problem has size $\lceil m/x \rceil \times n_t$. (Since the bound we want to prove on our algorithm's running time does not depend on k, we can ignore the change from looking for the kth smallest entry in each row of A to looking for the k_tth smallest entry in each row of A_t.) In what follows, we derive a bound on $\sum_{1 \le t \le x} n_t$,

which we then use to bound the running time of our algorithm.

To bound $\sum_{1\le t\le x} n_t$, we first select x representative rows $I_1, I_2, \ldots, I_x$, one from each of the A_t. (I_t may take any value from $\lambda_t(x)$ to $\lambda_{t+1}(x)-1$.) Then for $1 \le t \le x$ and $1 \le j \le n$, consider the difference $r(I_t, j) - \alpha(t,j)$. This difference is always nonnegative, since $\alpha(t,j)$ is a lower bound on $r(I_t, j)$. Moreover, since $r(I_t, j)$ is at most n,

$$\sum_{\substack{1\le t\le x\\1\le j\le n}} (r(I_t,j)-\alpha(t,j)) \le xn^2 .$$

However, we can prove a tighter $O(n^2)$ bound on this sum as follows.

From the definition of $\alpha(t,j)$, we have

$$\begin{aligned}
\sum_{\substack{1\le t\le x\\1\le j\le n}} \alpha(t,j) &= \left(\sum_{\substack{1\le t\le x\\1\le j\le n}} L(\lambda_{t+1}(x),j)\right) + \left(\sum_{\substack{1\le t\le x\\1\le j\le n}} R(\lambda_t(x),j)\right) \\
&= \left(\sum_{\substack{2\le t\le x\\1\le j\le n}} (L(\lambda_t(x),j)+R(\lambda_t(x),j))\right) \\
&\quad + \left(\sum_{1\le j\le n} L(\lambda_{x+1}(x),j)\right) + \left(\sum_{1\le j\le n} R(\lambda_1(x),j)\right) \\
&\ge \left(\sum_{\substack{2\le t\le x\\1\le j\le n}} r(\lambda_t(x),j)\right) + n .
\end{aligned}$$

(This last inequality follows from the identity $r(i,j) = L(i,j) + R(i,j)$, our convention that $L(\lambda_{x+1}(x),j) = 0$, and the bound $R(\lambda_1(x),j) \ge 1$.) Furthermore, for all rows i,

$$\begin{aligned}
\sum_{1\le j\le n} r(i,j) &= 1+2+\cdots+n \\
&= \frac{n(n+1)}{2} .
\end{aligned}$$

Thus,

$$\begin{aligned}\sum_{\substack{1\le t\le x\\1\le j\le n}} (r(I_t,j)-\alpha(t,j)) &\le x\frac{n(n+1)}{2}-(x-1)\frac{n(n+1)}{2}-n\\ &= \frac{n(n-1)}{2}\,. \end{aligned}\tag{3.1}$$

In a similar fashion, we can show that

$$\sum_{\substack{1\le t\le x\\1\le j\le n}} (\beta(t,j)-r(I_t,j)) \le \frac{n(n-1)}{2}\,. \tag{3.2}$$

Now let N_1 denote the total number of entries $a[I_t,j]$ in rows $I_1,I_2,\ldots,I_x$ such that $r(I_t,j) < k \le \beta(t,j)$, and let N_2 denote the total number of entries $a[I_t,j]$ in rows $I_1,I_2,\ldots,I_x$ such that $\alpha(t,j)\le k < r(I_t,j)$. Since column j of A_t is a column of A'_t if and only if

$$\alpha(t,j)\le k\le\beta(t,j)\,,$$

we must have

$$\sum_{1\le t\le x} n_t = N_1+N_2+x\,.$$

(The x term in the above expression comes from the unique column in each A_t such that $r(I_t,j)=k$.)

An upper bound on N_1 follows from (3.2). Since the entries in rows $I_1,I_2,\ldots,I_x$ satisfying $r(I_t,j)<k\le\beta(t,j)$ are a subset of all the entries in rows $I_1,I_2,\ldots,I_x$, we must have

$$\sum_{\substack{1\le t\le x\\1\le j\le n\\ \text{s.t. } r(I_t,j)<k\le\beta(t,j)}} (\beta(t,j)-r(I_t,j)) \le \sum_{\substack{1\le t\le x\\1\le j\le n}} (\beta(t,j)-r(I_t,j))\,.$$

Furthermore, for any rank R in the range $1\le R\le n$, there are exactly x entries $a[I_t,j]$ such that $r(I_t,j)=R$. Thus, for any positive d, there are at most xd entries $a[I_t,j]$ in rows $I_1,I_2,\ldots,I_x$ such that $r(I_t,j)<k\le\beta(t,j)$ and $\beta(t,j)-r(I_t,j)\le d$. Picking q_1 and r_1 so that $N_1=q_1x+r_1$

and $0 \leq r_1 < x$, this last observation implies

$$\sum_{\substack{1\leq t\leq x \\ 1\leq j\leq n \\ \text{s.t. } r(I_t,j)<k\leq\beta(t,j)}} (\beta(t,j) - r(I_t,j)) \geq \left(\sum_{s=1}^{q_1} xs\right) + r_1(q_1+1)$$

$$= \left(\frac{q_1 x + 2r_1}{2}\right)(q_1+1)$$

$$\geq \frac{N_1^2}{2x} .$$

Combining the preceding two inequalities with (3.2), we find

$$\frac{N_1^2}{2x} \leq \frac{n(n-1)}{2}$$

$$\leq \frac{n^2}{2}$$

or

$$N_1 \leq \sqrt{x}n .$$

By a similar argument, we can show

$$N_2 \leq \sqrt{x}n .$$

Thus, the total number of columns in $A'_1, A'_2, \ldots, A'_x$ is at most $2\sqrt{x}n + x$.

Given the above upper bound on the number of columns in $A'_1, A'_2, \ldots, A'_x$, we can now write down a recurrence relation describing the running time of our algorithm (as a function of x). Let $T(m,n)$ denote the time to compute the kth smallest entry in each row of an $m \times n$ Monge array. Then, for $m > 4$,

$$T(m,n) = O(xn\lg n) + \max_{\substack{n_1,n_2,\ldots,n_x \\ \text{s.t. } n_1+\cdots+n_x\leq 2\sqrt{x}n+x \\ \text{and } 1\leq n_t\leq n \text{ for } 1\leq t\leq x}} \sum_{t=1}^{x} T(\lceil m/x\rceil, n_t) .$$

To obtain the desired running time for our algorithm, we use $x = \left\lceil\sqrt{m}\right\rceil$. The above recurrence then yields $T(m,n) = O((\sqrt{m}\lg m)(n\lg n) + m\lg n)$.

To prove this last claim, we will argue by induction on m that for $m \geq 4$,

$$T(m,n) \leq c_1\sqrt{m}(\lg m - 1)(n-1)\lg n + c_2(m - \sqrt{m})\lg n ,$$

where c_1 and c_2 are constants independent of m and n. For simplicity, we will assume that $m = 2^{2^z}$ for some positive integer z, so that $m^{2^{-y}}$ is integral for all positive integers $y \leq z$. This assumption allows us to dispense with the ceilings around $\sqrt{m}$.

The base case of $m = 4$ is easy. Since $T(4,n) \leq c_3 n$ for some constant c_3, we need only choose $c_1 \geq c_3/2$ and $c_2 \geq c_1$ to insure that $T(4,n) \leq 2c_1(n-1)\lg n + 2c_2 \lg n$.

For the inductive step of our argument, we assume

$$T(M,n) \leq c_1\sqrt{M}(\lg M - 1)(n-1)\lg n + c_2(M - \sqrt{M})\lg n$$

for $M < m$. By the recurrence for $T(m,n)$, we know that

$$T(m,n) \leq c_4\sqrt{m}n\lg n + \max_{\substack{n_1,n_2,\ldots,n_{\sqrt{m}} \\ \text{s.t. } n_1+\cdots+n_{\sqrt{m}} \leq 2m^{1/4}n+m^{1/2} \\ \text{and } 1\leq n_t \leq n \text{ for } 1\leq t\leq\sqrt{m}}} \sum_{t=1}^{\sqrt{m}} T(\sqrt{m}, n_t)$$

for some constant c_4. Thus, by our inductive hypothesis,

$$\begin{aligned}
&T(m,n) \\
&\quad\leq c_4\sqrt{m}n\lg n \\
&\qquad + \max_{\substack{n_1,n_2,\ldots,n_{\sqrt{m}} \\ \text{s.t. } n_1+\cdots+n_{\sqrt{m}} \leq 2m^{1/4}n+m^{1/2} \\ \text{and } 1\leq n_t \leq n \text{ for } 1\leq t\leq\sqrt{m}}} \sum_{t=1}^{\sqrt{m}} (c_1 m^{1/4}(\lg m^{1/2} - 1)(n_t - 1)\lg n_t + c_2(m^{1/2} - m^{1/4})\lg n_t) \\
&\quad\leq c_4 m^{1/2} n\lg n \\
&\qquad + c_1 m^{1/4}((1/2)\lg m - 1)(2m^{1/4}n + m^{1/2})\lg n - m^{1/2}(c_1 m^{1/4}((1/2)\lg m - 1)\lg n) \\
&\qquad + m^{1/2}(c_2(m^{1/2} - m^{1/4})\lg n) \\
&\quad= c_4 m^{1/2} n\lg n \\
&\qquad + c_1 m^{1/2}(\lg m - 2)n\lg n + c_1 m^{3/4}((1/2)\lg m - 1)\lg n - c_1 m^{3/4}((1/2)\lg m - 1)\lg n \\
&\qquad + c_2 m\lg n - c_2 m^{3/4}\lg n
\end{aligned}$$

$$\begin{aligned}
&= c_4 m^{1/2} n \lg n \\
&\quad + c_1 m^{1/2}(\lg m - 1) n \lg n - c_1 m^{1/2} n \lg n \\
&\quad + c_2 m \lg n - c_2 m^{3/4} \lg n \\
&= c_1 m^{1/2}(\lg m - 1)(n-1)\lg n + c_2(m - m^{1/2})\lg n \\
&\quad + (c_4 - c_1) m^{1/2} n \lg n + (c_1 m^{1/2}(\lg m - 1) + c_2 m^{1/2} - c_2 m^{3/4}) \lg n \ .
\end{aligned}$$

Now

$$(c_4 - c_1) m^{1/2} n \lg n \ \leq \ 0$$

provided $c_1 \geq c_4$, and

$$\begin{aligned}
(c_1 m^{1/2}(\lg m - 1) + c_2 m^{1/2} - c_2 m^{3/4}) \lg n \ &\leq \ (c_1(\lg m - 1) - c_2(m^{1/4} - 1)) m^{1/2} \lg n \\
&\leq \ 0
\end{aligned}$$

provided $c_2 \geq c_1/(\sqrt{2} - 1)$ and $m \geq 4$. Thus,

$$T(m,n) \ \leq \ c_1 m^{1/2}(\lg m - 1)(n-1)\lg n + c_2(m - m^{1/2})\lg n \ ,$$

as we needed to show. ■

At the end of the previous subsection, we observed that our first row-selection algorithm is easily modified to compute not only the kth smallest entry in each row of an Monge array but also the first through $(k-1)$st smallest entries in each row. This subsection's row-selection algorithm can also be modified for this same task (with no change in the asymptotics of its running time.) Roughly speaking, whenever we remove some column j of subarray A_t from consideration because the upper bound $\beta(t, j)$ is strictly less than k, we output this column (or, more precisely, its location in the original array A), as each entry in this column is among the k smallest in its row. In this manner, we obtain the km total entries that we seek, represented as a collection of subcolumns of A.

3.2 Array Selection

In this section, we describe an algorithm for the array-selection problem. This algorithm, which first appeared in [KP91], uses the row-selection algorithm of Subsection 3.1.1 as a subroutine.

Theorem 3.5 The kth smallest entry overall in an $m \times n$ Monge array $A = \{a[i,j]\}$ can be computed in $O(m + n + k\lg(mn/k))$ time.

Proof In this proof, we actually prove a stronger result: we show that $O(m+n+k\lg(mn/k))$ time suffices for computing not only the kth smallest entry overall in A but also the first through $(k-1)$st smallest entries.

We first present an algorithm for those values of k that are greater than or equal to both m and n and then show how to modify this algorithm to handle smaller values of k.

To compute the k smallest entries of A when $\max\{m,n\} \leq k \leq mn$, we begin by checking the relative magnitudes of k and mn. If $k \geq mn/2$ (the "easy" case), we use the linear-time selection algorithm of [BFP+73] to compute the k smallest entries of A in $O(k)$ time. If, on the other hand, $k < mn/2$, we consider two subcases.

If $m \geq n$, we use the row-selection algorithm of Subsection 3.1.1 to compute the $\lceil 2k/m \rceil$ smallest entries in each row of A in $O(\lceil 2k/m \rceil(m+n)) = O(k)$ time. Let b_i denote the $\lceil 2k/m \rceil$th smallest entry in row i of A. Using the linear-time selection algorithm of [BFP+73], we can compute the $\lceil m/2 \rceil$th smallest of $b_1, \ldots, b_m$ in $O(m)$ time. Let b^* denote this $\lceil m/2 \rceil$th smallest b_i, and let B denote the $\lceil m/2 \rceil \times n$ subarray of A consisting of those rows i such that $b_i \geq b^*$. Furthermore, let L denote the list of $\lceil 2k/m \rceil \lfloor m/2 \rfloor = O(k)$ entries formed from the $\lceil 2k/m \rceil$ smallest entries of each row of A not in B. Now if row i of A is not in B, i.e., $b_i < b^*$, then the $n - \lceil 2k/m \rceil$ smallest entries in row i are all smaller than b^*, which means they are all smaller than the $\lceil 2k/m \rceil$ smallest entries in each row of B. Since B has $\lceil m/2 \rceil$ rows, this means that the $n - \lceil 2k/m \rceil$ smallest entries in row i are all smaller than at least $\lceil 2k/m \rceil \lceil m/2 \rceil \geq k$ other entries, i.e., these entries need not be considered as candidates for the kth smallest entry overall of A. Thus, if we recursively compute the k smallest entries in B and then use the linear-time selection algorithm of [BFP+73] to compute in $O(k)$ time the k smallest of these entries and the $O(k)$ entries of L, we obtain the k smallest entries in A.

If $m < n$, we apply the procedure described in the last paragraph to A's transpose A^T rather than A. (By Property 1.4, A^T is also Monge.) This computation requires $O(k)$ time plus the time needed to recursively compute the k smallest entries in an $m \times \lceil n/2 \rceil$ subarray of A.

Letting $T(k, m, n)$ denote the algorithm's running time in computing the k smallest entries in an $m \times n$ Monge array A when $\max\{m, n\} \leq k \leq mn$, we have

$$T(k,m,n) = \begin{cases} O(k) & \text{if } k \geq mn/2, \\ T(k, \lceil m/2 \rceil, n) + O(k) & \text{if } k < mn/2 \text{ and } m \geq n, \\ T(k, m, \lceil n/2 \rceil) + O(k) & \text{if } k < mn/2 \text{ and } m < n. \end{cases}$$

The solution to this recurrence is

$$T(k, m, n) = O(k \lg(mn/k)) .$$

Now suppose $k < m$. We can eliminate all but k of A's rows from consideration as follows. In $O(m + n)$ time, we can compute the row minima of A using the SMAWK algorithm. Then, using the linear-time selection algorithm of [BFP+73], we can select the k smallest of these minima in an additional $O(m)$ time. Now consider the $m - k$ rows of A corresponding to the $m - k$ largest row minima. The entries in these rows are all larger than the k smallest row minima, which means they are not among the k smallest entries of A. Thus, we can eliminate these $m - k$ rows from consideration. Similarly, if $k < n$, we can eliminate all but k of A's columns in $O(m + n)$ time.

Once the number of rows in A has been reduced to k or less and the number of columns in A has been reduced to k or less, we can apply our $O(k \lg(mn/k))$-time selection algorithm for arrays with $m \leq k$ rows and $n \leq k$ columns. This observation gives an algorithm for computing the k smallest entries in A that works for all values of k between 1 and mn and runs in $O(m + n + k \lg(st/k))$ time, where $s = \min\{m, k\}$ and $t = \min\{n, k\}$.

We can simplify the above expression for our algorithm's running time by observing that $m + n + k \lg(st/k) = \Theta(m + n + k \lg(mn/k))$ for all m, n, and k such that $1 \leq k \leq mn$. To see why this claim holds, first note that $m + n + k \lg(st/k) = O(m + n + k \lg(mn/k))$, since $s \leq m$

and $t \leq n$. Now suppose $m+n+k\lg(st/k) = o(m+n+k\lg(mn/k))$. This assumption implies

$$\lg(st/k) = o(\lg(mn/k)) \tag{3.3}$$

and

$$m+n = o(k\lg(mn/k)) . \tag{3.4}$$

Clearly, (3.3) implies k is smaller than at least one of m and n. Thus, if we assume without loss of generality that $m \leq n$, only two possibilities need be considered: $m < k \leq n$ and $k \leq m \leq n$.

If $m < k \leq n$, then $s = m$ and $t = k$. (3.3) then implies $\lg m = o(\lg(mn/k))$, which implies $\lg m = o(\lg(n/k))$. This last relation implies $k\lg(mn/k) = \Theta(k\lg(n/k))$. Since $k\lg(n/k) \leq n$, we then have $k\lg(mn/k) = O(n)$, which contradicts (3.4).

If $k \leq m \leq n$, then $s = k$ and $t = k$. (3.3) then implies $\lg k = o(\lg(mn/k))$, which implies $\lg k = o(\lg(mn))$. This last relation implies k is less than any polynomial in mn. Thus, $k\lg(mn/k)$ is also less than any polynomial in mn. In particular, $k\lg(mn/k) = o(\sqrt{mn})$, which again contradicts (3.4). ■

3.3 Row Sorting

In this section, we describe an algorithm for sorting the rows of an $m \times n$ Monge array in $O(mn)$ time if $m \geq n$ and in $O(mn(1+\lg(n/m)))$ time if $m < n$. This result again represents joint work with Kravets [KP91].

Theorem 3.6 The entries in each row of an $m \times n$ Monge array $A = \{a[i,j]\}$ can be sorted in $O(mn)$ time if $m \geq n$ and in $O(mn(1+\lg(n/m)))$ time if $m < n$.

Proof We begin by describing a more basic $O(mn+n^2)$-time algorithm for the row-sorting problem and then show how this second algorithm's running time can be reduced to $O(mn(1+\lg(n/m)))$ when $m < n$.

For $1 \leq i \leq m$ and $1 \leq R \leq n$, let $c_R(i)$ denote the column of A containing the entry in row i of A with rank R in row i. In other words, $c_R(i)$ is the unique column of A such that

$$r(i, c_R(i)) = R .$$

Furthermore, for $1 \leq R \leq n$, let $c_R(0) = R$. (These values may be interpreted as describing a "dummy" row 0 of A such that $a[0,1] < a[0,2] < \cdots < a[0,n]$; such a row can be added without affecting the Mongeness of A.)

Our basic algorithm consists of m phases, where in the ith phase, we sort row i of A by computing $c_1(i), c_2(i), \ldots, c_n(i)$ using $c_1(i-1), c_2(i-1), \ldots, c_n(i-1)$. Specifically, we use an insertion sort (such as the one described in [Knu73]) to sort row i, inserting first $a[i, c_1(i-1)]$, then $a[i, c_2(i-1)]$, then $a[i, c_3(i-1)]$, and so on through $a[i, c_n(i-1)$. To insert a particular entry $a[i,j]$ in the sorted list of previously inserted entries from row i, we first compare $a[i,j]$ to the largest previously inserted entry, then to the second largest, then to the third largest, and so on, until an entry smaller than $a[i,j]$ is found and $a[i,j]$'s place in the sorted list of previously inserted entries thereby ascertained.

Clearly, the order in which we insert the entries of row i affects the time spent sorting these entries. In particular, as noted in [Knu73], sorting N values with the insertion sort described above takes $\Theta(N + I)$ time, where I is the number of *inversions* separating the insertion order and the final sorted order for the values. An inversion is a pair of values (x, y) such that x is inserted before y but $x > y$. In the worst case, a sequence of N values may contain $\binom{N}{2} = \Omega(N^2)$ inversions; however, we will argue that the *total* number of inversions encountered in sorting all m rows of an $m \times n$ Monge array is $O(n^2)$.

Given the order in which we insert the entries of row i, an inversion encountered while sorting row i corresponds to a pair of columns j_1 and j_2, such that $a[i-1, j_1] < a[i-1, j_2]$ and $a[i, j_1] > a[i, j_2]$ (where, by convention, $a[0, j_1] < a[0, j_2]$ if and only if $j_1 < j_2$). Since A is Monge, Property 1.1 implies that for each pair of columns j_1 and j_2, there exists at most one row index i such that $a[i-1, j_1] < a[i-1, j_2]$ and $a[i, j_1] > a[i, j_2]$. (In fact, such a row index exists only if $j_1 < j_2$.) Thus, in sorting all the rows of A, we can encounter at most $O(n^2)$ total inversions, one for each pair of columns. This bound gives our basic algorithm a running time of $O(mn + n^2)$.

To obtain an algorithm that runs in $O(mn)$ time if $m \geq n$ and in $O(mn(1 + \lg(n/m)))$ time if $m < n$, we first note that if $m \geq n$, the basic algorithm described above already has the desired running time. On the other hand, if $m < n$, we need to modify the basic algorithm

as follows. First, we partition A into $\lceil n/m \rceil$ subarrays of size at most $m \times m$. Then, using our basic algorithm, we sort the rows of these subarrays in $O(m^2)$ time per subarray or $O(mn)$ total time. Finally, we merge the $\lceil n/m \rceil$ sorted subrows corresponding to each row of A in $O(n(1 + \lg(n/m)))$ time per row or $O(mn(1 + \lg(n/m)))$ total time. ∎

Note that the size of our algorithm's output, mn, is not necessarily a lower bound on the time required for the row-sorting problem. Just as the k smallest entries in each row of an $m \times n$ Monge array A can be specified in $o(km)$ space when k is large (see Subsection 3.1.2), there may be a more concise representation for the m permutations ordering by magnitude the entries in each of A's rows.

3.4 Array Sorting

As a final variation on this chapter's theme, we consider the problem of sorting all the entries of an $m \times n$ Monge array. Unlike the three array problems we considered in Sections 3.1 through 3.3, the array-sorting problem is not significantly easier than the general problem of sorting mn arbitrary values, which takes $\Theta(mn \lg mn)$ time. Specifically, we can prove the following theorem (a variant of which first appeared in [KP91]).

Theorem 3.7 All the entries in an $m \times n$ Monge array $A = \{a[i,j]\}$ can be sorted in $O(mn \lg mn)$ time. Furthermore, sorting all the entries of an $m \times n$ Monge array requires $\Omega(mn \lg t)$ comparisons and thus $\Omega(mn \lg t)$ time, where $t = \min\{m, n\}$.

Proof The upper bound follows immediately from any general $\Theta(N \lg N)$-time algorithm for sorting N arbitrary values. Note that first sorting the rows of A using the row-sorting algorithm of the previous section and then merging these sorted rows (in $O(mn \lg m)$ time) does not yield an asymptotically faster algorithm, since for $m \geq n$, $mn \lg m = \Theta(mn \lg mn)$, and for $m < n$, $mn(1 + \lg(n/m)) + mn \lg m = \Theta(mn \lg mn)$.

For the lower bound, first consider the $m \times n$ array $A = \{a[i,j]\}$ such that $a[i,j] = -(i+j)^2$. By Property 1.7, this array is Monge, since $f(x) = -x^2$ is concave. Moreover, for all i in the range $1 \leq i < m$ and all j in the range $1 \leq j < n$,

$$(a[i,j+1] + a[i+1,j]) - (a[i,j] + a[i+1,j+1]) = -2(i+j+1)^2 + (i+j)^2 + (i+j+2)^2$$

$$= 2.$$

This last observation is significant because it implies that A remains Monge even if we slightly perturb the entries of A. More precisely, the $m \times n$ array $A' = a'[i,j]$ given by $a'[i,j] = a[i,j] + \epsilon[i,j]$ is Monge so long as $-1/2 \leq \epsilon[i,j] \leq 1/2$ for all i and j.

Now for $1 \leq s \leq m+n-1$, consider the sth *diagonal* of A, which consists of those entries $a[i,j]$ such that $i+j = s-1$, and let d_s denote the number of entries in this diagonal, so that

$$d_s = \begin{cases} s & \text{if } 1 \leq s < t, \\ t & \text{if } t \leq s \leq m+n-t, \\ m+n-s & \text{if } m+n-t < s \leq m+n-1, \end{cases}$$

where $t = \min\{m,n\}$. All the entries in this diagonal have the same value, $-(s-1)^2$; thus, by perturbing these entries slightly, their relative magnitudes may be reordered in any of $d_s!$ different ways *without* destroying the Mongeness of A. Furthermore, since the entries in different diagonals can be reordered independently, we can obtain

$$\prod_{s=1}^{m+n-1} d_s = (t!)^{m+n-2t+1}((t-1)!)^2((t-2)!)^2 \cdots ((1)!)^2$$

different total orderings for all A's entries. Thus, in a linear-decision-tree model of computation,

$$\lg\left((t!)^{m+n-2t+1}((t-1)!)^2((t-2)!)^2 \cdots ((1)!)^2\right) = \Omega(mn \lg t)$$

comparisons are necessary (in the worst case) to solve the array-sorting problem for an $m \times n$ array Monge array. ■

3.5 Open Problems

In this chapter, we explored two fundamental comparison problems — selection and sorting — in the context of two-dimensional Monge arrays. We provided simple but efficient algorithms for the row-selection, array-selection, and row-sorting problems, algorithms that take advantage

of the structure of Monge arrays to obtain significantly better results than are possible for arbitrary arrays. We also showed that Mongeness does not help much with the array-sorting problem.

We conclude with a few of the more interesting questions left unresolved by this chapter:

1. In Subsection 3.1.2, we gave an algorithm that identifies the k smallest entries in each row of an $m \times n$ Monge array in $O((\sqrt{m}\lg m)(n\lg n) + m\lg n)$ time. For large values of k, this running time is $o(km)$, even though km entries are identified. This observation leads naturally to the question of whether there exists an algorithm for the row-selection problem that runs in $o(km)$ time when k is smaller.

2. In Section 3.2, we used the row-selection algorithm of Subsection 3.1.1 to obtain an efficient algorithm for the array-selection problem. It remains open whether our other row-selection algorithm can be used in a similar fashion.

3. The only array-searching problem considered in this chapter for which we obtain matching upper and lower bounds is the array-sorting problem discussed in Section 3.4. (The bounds for array-sorting are matching when $m = \Theta(n)$.) It remains open whether the algorithms for row selection, array selection, and row sorting given in Sections 3.1 through 3.3 can be improved or nontrivial lower bounds for these problems obtained. (Lower bounds might follow from the sizes of the various problems' search spaces. For example, a lower bound of $\Omega(S)$ on the number of different combinations of row ranks possible for the entries of a Monge array would imply an $\Omega(\lg S)$ lower bound on the time necessary to sort the array's rows in a linear-decision-tree model.)

Chapter 4

Parallel Algorithms

In this chapter, we present parallel algorithms for computing minimal entries in two- and three-dimensional Monge arrays. We consider three models of parallel computation: the CREW-PRAM model, the CRCW-PRAM model, and Valiant's comparison model. Algorithms in all three models are analyzed in terms of both the number of processors and the amount of time used. A PRAM ("Parallel Random-Access Machine") consists of a number of processors, each with its own local memory, and a shared global memory, accessible to all the processors. Time is measured as for sequential RAMs. In the CREW ("Concurrent-Read, Exclusive-Write") version of the PRAM model, concurrent reads of the same global memory location are allowed, but concurrent writes to the same location are not. In the CRCW ("Concurrent-Read, Concurrent-Write") version, both concurrent reads and writes are allowed, but if multiple processors attempt to write to the same location at the same time, only one of the processors will succeed, and it may be any one of the processors. In Valiant's comparison model, time is measured in terms of comparisons only. In other words, each processor may perform only one comparison per time step, but an unlimited number of noncomparison operations. Issues such as access to global memory and processor allocation can thus be ignored in this model.

This chapter is organized as follows. In Section 4.1, we mention several parallel algorithms for general minimization and merging that our Monge-array algorithms use as subroutines. Then, in Section 4.2, we discuss parallel algorithms for finding row minima in two-dimensional Monge arrays. Finally, Sections 4.3 and 4.4 consider parallel plane and tube minimization, respectively, in three-dimensional Monge arrays. With the except of three algorithms due to

Apostolico, Atallah, Larmore, and McFaddin [AALM90], Atallah and Kosaraju [AK91], and Atallah [Ata90] that are mentioned briefly in Sections 4.2 and 4.4, the results presented in this chapter represent joint work with Aggarwal that first appeared in [AP89a].

4.1 Preliminaries

In finding minimal entries in a Monge array, we often need to solve two more fundamental problems: given a list, compute its minimum element, and given two sorted lists, merge these lists into a single sorted list. Both of these problems are well understood in all three of the parallel models we consider. We summarize relevant results for these problems in the following four lemmas.

Lemma 4.1 (Kruskal [Kru83]) In the CREW-PRAM model, the minimum of n numbers can be computed in $\Theta(\lg n)$ time using $n/\lg n$ processors. Also, two sorted lists containing a total of n elements can be merged in $\Theta(\lg n)$ time using $n/\lg n$ processors. ■

Lemma 4.2 (Shiloach and Vishkin [SV81]) In the CRCW-PRAM model (and thus in Valiant's comparison model as well), the minimum of n numbers can be computed in $\Theta(\lg(1/\epsilon))$ time using $n^{1+\epsilon}/\lg(1/\epsilon)$ processors, for any $\epsilon > 0$. (In particular, the minimum of n numbers can be computed in $\Theta(\lg\lg n)$ time using $n/\lg\lg n$ processors.) Also, two sorted lists containing a total of n numbers elements can be merged in $\Theta(\lg(1/\epsilon))$ time using $n^{1+\epsilon}/\lg(1/\epsilon)$ processors, for any $\epsilon > 0$. ■

Lemma 4.3 In the CREW-PRAM model, the row minima of an arbitrary $r \times n$ array $A = \{a[i,j]\}$ can be computed in $O(r + \lg n)$ time using n processors.

Proof We first describe an $O(\lg n)$-time, n-processor algorithm for computing the minimum of n numbers $a_1, \ldots, a_n$ on a CREW-PRAM. Assuming (for the sake of simplicity) that $n = 2^s$ for some positive s, we consider the following n-leaf complete binary tree T. Associated with each leaf of T is one of the a_i, and associated with each internal node of T is a processor. Since T has $n - 1$ internal nodes, one processor is unused. The processor associated with node u of T computes b_u, the minimum of the a_i below u. If v and w are the children of u in T, then b_u is just the minimum of b_v and b_w (provided we define b_u to be the a_i associated with u if u

is a leaf). Thus, the processor associated with u can compute b_u in one time step once b_v and b_w have been computed. Since the depth of T is $\lg n$, this means that $\lg n$ time steps suffice to compute the minimum of a_1 through a_n.

Note that in the preceding algorithm, only those processors associated with nodes at height h in T are actually doing anything at time h. This means we can "pipeline" this algorithm to compute the row minima of A in $O(r + \lg n)$ time — we just "feed" a new row of A to the leaves of T every time step and the minimum entry in that row "pops out" at T's root $\lg n$ steps later. ■

Lemma 4.4 In the CRCW-PRAM model (and therefore, in Valiant's comparison model as well), the row minima of an arbitrary $r \times n$ array $A = \{a[i,j]\}$, can be computed in $O(r + \lg\lg n)$ time using n processors.

Proof We first describe an $O(\lg\lg n)$-time, n-processor algorithm for computing the minimum of n numbers $a_1, \ldots, a_n$ on a CRCW-PRAM. Assuming (for the sake of simplicity) that $n = 2^{2^s+s-1}$ for some positive s, we consider the following tree T of height s. Every node at height h in T has exactly $c(h) = 2^{2^{h-1}+1}$ children, which implies that the number of nodes at height h in T is

$$\prod_{i=h+1}^{s} 2^{2^{i-1}+1} = 2^{\sum_{i=h+1}^{s}(2^{i-1}+1)} = 2^{2^s-2^h+s-h}.$$

In particular, T has $2^{2^s+s-1} = n$ leaves. Associated with each leaf of T is one of the a_i, and associated with each internal node of T are $p(h) = 2^{2^h-1}$ processors, where h is the height of the internal node. Since

$$\sum_{h=1}^{s}\left(2^{2^s-2^h+s-h}\right)\left(2^{2^h-1}\right) = \sum_{h=1}^{s} 2^{2^s+s-h-1} = \sum_{h=1}^{s} n/2^h = n - n/2^s,$$

$n/2^s$ processors are unused. The processors associated with node u of T compute b_u, the minimum of the a_i below u. b_u is just the minimum of b_v over all children v of u (provided we define b_u to be the a_i associated with u if u is a leaf). Since $(c(h))^2 = 2^{2^h+2} = 8p(h)$, Lemma 4.2 tells us that the processors associated with u can compute b_u in constant time once b_v has been computed for all of u's children v. Since the depth of T is s, this means that $s = \Theta(\lg\lg n)$ time steps suffice to compute the minimum of a_1 through a_n.

Note that in the preceding algorithm, only those processors associated with nodes at height h in T are actually doing anything at time h. This means we can "pipeline" this algorithm to compute the row minima of A in $O(r + \lg\lg n)$ time — we just "feed" a new row of A to the leaves of T every time step and the minimum entry in that row "pops out" at T's root $\lg\lg n$ steps later. ■

We also make frequent use of the following theorem, due to Brent [Bre74], in reducing the processor requirements of our array-searching algorithms.

Theorem 4.5 (Bre74) Suppose $\mathcal{T}$ is a p-processor algorithm that runs in time t and performs a total of w operations in either the CREW-PRAM model, the CRCW-PRAM model, or Valiant's comparison model. Then we can simulate $\mathcal{T}$ in time $O(t)$ using w/t processors in the same model of parallel computation.

4.2 Two-Dimensional Monge Arrays

In this section, we consider the problem of finding the row minima of a two-dimensional Monge array. We begin with a simple result for Valiant's comparison model.

Theorem 4.6 In Valiant's comparison model, we can compute the row minima of an $n \times m$ Monge array $A = \{a[i,j]\}$ in $O(\lg n + \lg\lg m)$ time using $O((n + m\lg n)/(\lg n + \lg\lg m))$ processors.

Proof We begin by comparing the jth entry in row $\lceil n/2 \rceil$ of A to the $(j+1)$st entry of the same row, for all j between 1 and m. Now, if

$$a[\lceil n/2 \rceil, j] \geq a[\lceil n/2 \rceil, j+1],$$

then

$$a[i,j] \geq a[i,j+1]$$

for all i between 1 and $\lceil n/2 \rceil - 1$. This means the entries in column $j+1$ and rows 1 through $\lceil n/2 \rceil - 1$ no longer need be considered as candidates for the row minimum in their respective

rows. On the other hand, if

$$a[\lceil n/2 \rceil, j] < a[\lceil n/2 \rceil, j+1],$$

then

$$a[i, j] < a[i, j+1]$$

for all i between $\lceil n/2 \rceil + 1$ and n. This means the entries in column j and rows $\lceil n/2 \rceil + 1$ through n may be eliminated from consideration. Now, let A' denote the subarray formed by taking the top $\lceil n/2 \rceil - 1$ rows of A and all the columns of A that still contain candidates for the minima of these rows. Similarly, let A'' denote the subarray formed by taking the bottom $\lceil n/2 \rceil - 1$ rows of A and all the columns of A that still contain candidates for the minima of these rows. Note that if A' and A'' contain m' and m'' columns, respectively, then $m' \geq 1$, $m'' \geq 1$, and $m' + m'' = m + 1$.

To find the row minima of A, we now need to solve three separate problems in parallel: we need to find the row minima in the subarray A', the row minima in the subarray A'', and the minimum entry in row $\lceil n/2 \rceil$. The first two problems we solve recursively. The third we solve using Lemma 4.2 — this requires $O(\lg \lg m)$ time and $O(m)$ total comparisons. Thus, if $T(n, m)$ denotes the time required to solve the row-minima problem for A in Valiant's comparison model, then

$$T(n,m) \leq 1 + \min_{\substack{m', m'' \geq 1 \\ m' + m'' = m+1}} \{O(\lg \lg m), T(\lceil n/2 \rceil, m'), T(\lceil n/2 \rceil, m'')\},$$

where $m' + m'' = m + 1$ and $T(1, m) = O(\lg \lg m)$. The solution to this recurrence is $T(n, m) = O(\lg n + \lg \lg m)$. Similarly, if $W(n, m)$ denotes the total number of comparisons required, then

$$W(n,m) \leq O(m) + \min_{\substack{m', m'' \geq 1 \\ m' + m'' = m+1}} \{W(\lceil n/2 \rceil, m') + W(\lceil n/2 \rceil, m'')\},$$

where $W(1, m) = O(m)$. The solution to this recurrence is $W(n, m) = O(n + m \lg n)$. Thus, by Brent's theorem, $O((n + m \lg n)/(\lg n + \lg \lg m))$ processors suffice. ■

For $m = n$, Theorem 4.6 tells us that we can solve the row-minima problem for A with $O(n \lg n)$ total comparisons. However, the sequential algorithm of [AKM+87] uses $\Theta(n)$ comparisons; thus, it remains open whether there exists an $O(\lg n)$-time algorithm for Valiant's

comparison model using $o(n \lg n)$ comparisons. Furthermore, the only bound we have on the time to compute A's row minima in Valiant's comparison model is $\Omega(\lg\lg n)$ — this bound follows from the $\Omega(\lg\lg n)$ bound on the time to compute the minimum of n numbers [SV81]. Thus, it also remains open whether there exists an $o(\lg n)$-time algorithm for Valiant's comparison model.

We do not know how to convert the algorithm of Theorem 4.6 to a PRAM algorithm, as it is unclear how to perform processor allocation. However, using two different approaches, we can obtain both an $O(\lg n)$-time, n-processor CRCW-PRAM algorithm and an $O(\lg n)$-time, $(n \lg n)$-processor CREW-PRAM algorithm for computing the row minima of an $n \times n$ Monge array.

Before we present any of our PRAM algorithms, we first prove the following technical lemma.

Lemma 4.7 Given the minimum entry in every rth row of an $n \times m$ Monge array A such that $1 \le r \le n$, we can compute the remaining row minima in $O(r + \lg m)$ time on a CREW-PRAM and in $O(r + \lg\lg m)$ time on a CRCW-PRAM, using $(n+m)/r$ processors in both cases.

Proof For the sake of simplicity, we assume r divides n and m. For $1 \le i \le n/r$, let $k(i)$ denote the index of the column containing the minimum entry of row ir, and let $k(0) = 1$. Let A_i denote the subarray of A containing rows $(i-1)r+1$ through $ir-1$ and columns $k(i-1)$ through $k(i)$. Since A is Monge, the minima in rows $(i-1)r+1$ through $ir-1$ must lie in A_i.

Let $c(i) = \lceil (k(i) - k(i-1) + 1)/r \rceil$. We partition A_i into $c(i)$ subarrays. Specifically, for $1 \le j \le c(i)$, let $S_{i,j}$ denote the subarray of A_i containing rows $(i-1)r+1$ through $ir-1$ and columns $k(i-1)+(j-1)r$ through $\min\{k(i-1)+jr-1, k(i)\}$. This is suggested in Figure 4.1. Note that each of these subarrays has size at most $(r-1) \times r$ and is Monge. Moreover, since $\sum_{i=1}^{n/r} k(i) - k(i-1) \le m$, we have $\sum_{i=1}^{n/r} c(i) \le (n+m)/r$, i.e., the total number of subarrays $S_{i,j}$ is no more than $(n+m)/r$. Thus, we can assign one processor to each of these subarrays.

To assign processors to subarrays, we construct an (n/r)-element linear array $X = \{x_j\}$, where $x_j = k(j) - 1/2$ for $1 \le j \le n/r$, and an (m/r)-element linear array $Y = \{y_j\}$, where $y_j = jr$ for $1 \le j \le m/r$. Using Kruskal's techniques [Kru83], we can merge these two arrays in $O(\lg(n/r))$ time on a CREW-PRAM and in $O(\lg\lg(n/r))$ time on a CRCW-PRAM, using n/r processors in both cases. Let $Z = \{z_j\}$ be the array we obtain. We assign a processor to

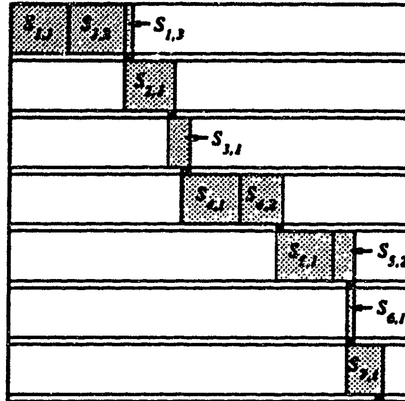

Figure 4.1: Suppose we know the minimum entry in every rth row of A, and that the positions of these minima are given by the small black squares. Then the remaining row minima of A must lie in the shaded regions. To find the row minima in these shaded regions, we partition the regions into a number of subarrays $S_{i,j}$ of size at most $r \times r$.

each element of Z. Now, for any j between 1 and $(m+n)/r$, consider the processor assigned to the element z_j. In order to determine its subarray, this processor first determines whether the element z_j was originally in X or Y. (We maintain pointers so that this can be done in constant time in either model.) If z_j corresponds to an element x_i from X, then the processor is assigned to the subarray $S_{i,1}$. If, on the other hand, z_j corresponds to an element y_k from Y, then the processor first computes $i = j - k$. x_i is the last element from X preceding y_k in Z. Next, the processor determines the element z_l in Z corresponding to x_i. The processor is then assigned to the subarray $S_{i,k-l}$. Note that this procedure results in either $c(i)$ or $c(i)+1$ processors being assigned to rows $(i-1)r+1$ through $ir-1$. If $c(i)+1$ processors are assigned, then the last one (the one assigned to the nonexistent subarray $S_{i,c(i)+1}$) does nothing.

Once the processors are assigned, each processor solves the row-minima problem for its subarray in $O(r)$ time using the sequential array-searching algorithm of [AKM+87]. Now, for $1 \le i \le n/r$, let R_i denote the $(r-1) \times c(i)$ array formed from the row minima of $S_{i,1}$ through $S_{i,c(i)}$. The row minima of A_i (which correspond to the minima in rows $(i-1)r+1$ through $ir-1$ of A) are just the row minima of R_i. Moreover, we have exactly $c(i)$ processors assigned to each R_i. Thus, using Lemmas 4.3 and 4.4, we can compute the row minima of R_i in $O(r + \lg c(i))$ time on a CREW-PRAM and in $O(r + \lg\lg c(i))$ time on a CRCW-PRAM. Since $c(i) \le m/r$

for all i, we obtain the specified time and processor bounds. ■

By applying Lemma 4.7 repeatedly, we obtain the following theorem.

Theorem 4.8 For $1 \leq r \leq n$, we can compute the row minima of an $n \times m$ Monge array A in $O((\lg n/\lg r)(r + \lg m))$ time on a CREW-PRAM or in $O((\lg n/\lg r)(r + \lg\lg m))$ time on a CRCW-PRAM, using $(n + m)/r$ processors in both cases.

Proof For the sake of simplicity, we again assume r divides n and m. Form an $n/r \times m$ array B by taking every rth row of A. Recursively compute the row minima of B, using $n/r^2 + m/r \leq (n + m)/r$ processors. Then, by invoking Lemma 4.7, compute the remaining row minima of A using $(n + m)/r$ processors.

If $T(n, m)$ denotes the time to solve the row-minima problem for A using a CREW-PRAM and if $T'(n, m)$ denotes the time to solve the row-minima problem for A using a CRCW-PRAM, then $T(n, m) \leq T(n/r, m) + O(r + \lg m)$ and $T'(n, m) \leq T(n/r, m) + O(r + \lg\lg m)$. Since $T(r, m) = O(r + \lg m)$ and $T'(r, m) = O(r + \lg\lg m)$, these recurrences yield the specified time bounds. ■

For $m = n$ and $r = n^{\epsilon}$, where $\epsilon > 0$ is a constant, this theorem yields an optimal processor-time product for both CREW- and CRCW-PRAMs. For $m = n$ and $r = \lg n$, we obtain an $O(\lg^2 n/\lg\lg n)$-time, $(n/\lg n)$-processor CREW-PRAM algorithm for computing A's row minima. Finally, for $m = n$ and $r = \lg\lg n$, we obtain an $O(\lg n \lg\lg n/\lg\lg\lg n)$-time, $(n/\lg\lg n)$-processor CRCW-PRAM algorithm.

We can obtain even better time bounds (but worse processor-time products) using a divide-and-conquer approach; for such an approach we need the following lemma.

Lemma 4.9 Given an $n \times m$ Monge array A such that $m \leq n$, suppose we know the minimum in every $\lfloor n/m \rfloor$th row of A. Then, we can compute the remaining row minima of A in $O(\lg m)$ time using $n/\lg m$ processors on a CREW-PRAM and in $O(\lg\lg m)$ time using $n/\lg\lg m$ processors on a CRCW-PRAM.

Proof In this proof, we *do not* assume m divides n — this complicates the proof a little bit, but it is worth going through these details at least once. For $1 \leq i \leq m$, let $k(i)$ denote

the index of the column containing the minimum entry of row $i\lfloor n/m \rfloor$. Also, let $k(0) = 1$ and let $k(m+1) = n$. Furthermore, for $1 \le i \le m+1$, let A_i denote the subarray of A containing rows $(i-1)\lfloor n/m \rfloor + 1$ through $\min\{i\lfloor n/m \rfloor - 1, n\}$ and columns $k(i-1)$ through $k(i)$. Since A is Monge, the minima in rows $(i-1)\lfloor n/m \rfloor + 1$ through $\min\{i\lfloor n/m \rfloor - 1, n\}$ must lie in A_i. Using the list merging technique from the proof of Lemma 4.7, we can assign $(\lfloor n/m \rfloor - 1)(k(i) - k(i-1) + 1)$ processors to A_i in the specified time and processor bounds. Furthermore, we can find the minimum entry in each row of A_i in $O(\lg(k(i) - k(i-1) + 1))$ time on a CREW-PRAM and in $O(\lg\lg(k(i) - k(i-1) + 1))$ time on a CRCW-PRAM, using $k(i) - k(i-1) + 1$ processors in both cases. Since $\sum_{i=1}^{m+1} k(i) - k(i-1) + 1 \le 2m + 1$, the total number of processors required and the total number of comparisons used are both

$$\sum_{i=1}^{m+1} (\lfloor n/m \rfloor - 1)(k(i) - k(i-1) + 1) = O(n).$$

Noting that $k(i) - k(i-1) + 1 \le m$ for all i, and applying Brent's theorem to reduce the number of processors, we obtain the specified time and processor bounds. ■

Theorem 4.10 We can compute the row minima of an $n \times n$ Monge array A in $O(\lg n)$ time using n processors on a CRCW-PRAM.

Proof For the sake of simplicity, we only prove this theorem for the case of $n = 2^{2^s}$, where s is some positive integer. Consider the $\sqrt{n} \times n$ array B formed by taking every $\sqrt{n}$th row of A. Partition this array into $\sqrt{n}$ subarrays such that the jth subarray B_j, $1 \le j \le \sqrt{n}$, contains columns $(j-1)\sqrt{n} + 1$ through $i\sqrt{n}$ of B. We assign $\sqrt{n}$ processors to each B_j and recursively compute its row minima. Now the row minima of the B_j form a $\sqrt{n} \times \sqrt{n}$ array B', such that the minimum entry in row i of B' is precisely the minimum entry in row $i\sqrt{n}$ of A. Thus, we can assign $\sqrt{n}$ processors to each row of B' and compute its row minima in $O(\lg\lg n)$ time.

For $1 \le i \le \sqrt{n}$, let $k(i)$ denote the index of the column containing the minimum entry of row $i\sqrt{n}$, and let $k(0) = 1$. Since A is Monge, the minimum entries in rows $(i-1)\sqrt{n} + 1$ through $i\sqrt{n} - 1$ of A must lie in columns $k(i-1)$ through $k(i)$. For $1 \le i \le \sqrt{n}$, let

$$c(i) = \left\lceil \frac{k(i) - k(i-1) + 1}{\sqrt{n}} \right\rceil .$$

For $1 \leq j \leq c(i)$, let $S_{i,j}$ be the subarray of A that contains rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ and columns $k(i-1)+(j-1)\sqrt{n}$ through $\min\{k(i-1)+j\sqrt{n}-1, k(i)-1\}$. Note that we define $S_{i,j}$ to contain part of row $i\sqrt{n}$, even though we already know that row's minimum — this is merely for convenience. Also note that the minimum entries in rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ of A are either in one of the $S_{i,j}$ or in column $k(i)$, and that $S_{i,1}, \ldots, S_{i,c(i)-1}$ are all $\sqrt{n} \times \sqrt{n}$ arrays. Let $d(i) = k(i) - k(i-1) - (c(i)-1)\sqrt{n}$, and let $S'_{i,c(i)}$ be the subarray formed by taking every $\lfloor \sqrt{n}/d(i) \rfloor$th row of $S_{i,c(i)}$. Note that $S'_{i,c(i)}$ is a $d(i) \times d(i)$ array.

For $1 \leq i \leq \sqrt{n}$ and $1 \leq j < c(i)$, we assign $\sqrt{n}$ processors to $S_{i,j}$, and for $1 \leq i \leq \sqrt{n}$, we assign $d(i)$ processors to $S'_{i,c(i)}$. This can be done in $O(\lg\lg n)$ time using the merging technique described in the proof of Lemma 4.7. Moreover, since

$$\sum_{i=1}^{\sqrt{n}}(c(i)-1)\sqrt{n}+d(i) \;=\; \sum_{i=1}^{\sqrt{n}} k(i)-k(i-1) \;\leq\; n,$$

we have enough processors. We then recursively solve the row-minima problem for these subarrays.

Next, we assign $\sqrt{n}$ processors to each of $S'_{1,c(1)}, S'_{2,c(2)}, \ldots, S'_{\sqrt{n},c(\sqrt{n})}$. Then for $1 \leq i \leq \sqrt{n}$, we use the row minima of $S'_{i,c(i)}$ and Lemma 4.9 to compute the row minima of $S_{i,c(i)}$ in $O(\lg\lg n)$ time.

Finally, we compute the minimum entry in row l, $(i-1)\sqrt{n}+1 \leq l \leq i\sqrt{n}$, by taking the minimum of the entry in column $k(i)$ and the $c(i)$ values obtained for row l in solving the row-minima problems for $S_{i,1}, S_{i,2}, \ldots, S_{i,c(i)}$. To do this, we first spread out the $(c(i)-1)\sqrt{n}+d(i)$ processors originally assigned to $S_{i,1}, S_{i,2}, \ldots, S_{i,c(i)}$, assigning at least $c(i)-1$ to each of rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$. The processors assigned to any particular row can then compute the minimum of the $c(i)$ candidates for the row's minimum obtained from $S_{i,1}, S_{i,2}, \ldots, S_{i,c(i)}$ in $O(\lg\lg c(i)) = O(\lg\lg n)$ time (if $c(i) = 1$, then no computation is necessary) — this gives the minimum entry in columns $k(i-1)$ through $k(i)-1$ of the row. Then we reshuffle the processors once more, assigning one processor to each row of A. For $(i-1)\sqrt{n}+1 \leq l \leq i\sqrt{n}$, the processor assigned to row l can then compute the minimum of the entry in column $k(i)$ of row l and the minimum of the entries in columns $k(i-1)$ through $k(i)-1$ of row l to obtain minimum entry in row l.

To analyze the time complexity of this algorithm, note that all steps of the algorithm, other than the recursive calls, can be done in $O(\lg\lg n)$ time. If $T(n)$ denotes the time complexity of solving the row-minima problem for A, then the two recursive calls take at most $T(\sqrt{n})$ time apiece. Consequently, $T(n) \le 2T(\sqrt{n}) + O(\lg\lg n)$, which, together with $T(1) = O(1)$, yields the required time bound. ■

Theorem 4.11 We can compute the row minima of an $n \times n$ Monge array A in $O(\lg n \lg\lg n)$ time using $n/\lg\lg n$ processors on a CREW-PRAM.

Proof We first show how to solve the row-minima problem for A in $O(\lg n \lg\lg n)$ time by using n processors on a CREW-PRAM. Then we use Brent's theorem to reduce the number of processors to $O(n/\lg\lg n)$.

Our algorithm for a CREW-PRAM with n processors is the same as that given for Theorem 4.10. A straightforward analysis shows that steps that are not recursive calls can now be executed in $O(\lg n)$ time. Consequently, if $T(n)$ denotes the time complexity of the row-minima problem for A, then $T(n) \le 2T(\sqrt{n}) + O(\lg n)$ and $T(1) = O(1)$. This recurrence yields $T(n) = O(\lg n \lg\lg n)$, which implies n processors can solve the row-minima problem in $O(\lg n \lg\lg n)$ time.

To reduce the number of processors, observe that if $W(n)$ denotes the number of operations required by this algorithm, then

$$W(n) \;=\; O(n) + \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{\sqrt{n}}\{(c(i)-1)W(\sqrt{n}) + W(d(i))\},$$

which, together with $W(1) = O(1)$, yields $W(n) = O(n\lg n)$. Consequently, by Brent's theorem, the number of processors required is $O(n/\lg\lg n)$. ■

A superlinear number of processors allows us to obtain even faster algorithms. Specifically, we have the following theorem.

Theorem 4.12 For any $\epsilon > 0$, we can compute the row minima of an $n \times m$ Monge array A in $O((1/\epsilon)\lg(1/\epsilon))$ time using $(n+m)^{1+\epsilon}$ processors on a CRCW-PRAM.

Proof Consider the $n^{\epsilon/2} \times m$ array B formed by taking every $n^{1-\epsilon/2}$th row of A. Since $n^{\epsilon/2}m^{1+\epsilon/2} \le (n+m)^{1+\epsilon}$, we can assign $m^{1+\epsilon/2}$ processors to each row of B and then use [SV81] to obtain B's row minima in $O(\lg(1/\epsilon))$ time.

Now, for $1 \le i \le n^{\epsilon/2}$, let $k(i)$ denote the index of the column containing the minimum entry in row $in^{1-\epsilon/2}$ of A, and let $k(0) = 1$. Let S_i denote the subarray of A containing rows $(i-1)n^{1-\epsilon/2}+1$ through $in^{1-\epsilon/2}-1$ and columns $k(i-1)$ through $k(i)$, and let $m(i) = k(i) - k(i-1) + 1$ be the number of columns in S_i. Since A is Monge, the minimum entries in rows $(i-1)n^{1-\epsilon/2}+1$ through $in^{1-\epsilon/2}-1$ must lie in S_i. Thus, we need only recursively solve the row-minima problems in these subarrays to find the remaining row minima.

To recursively solve the row-minima problem for S_i, we need

$$\left(n^{1-\epsilon/2} - 1 + m(i)\right)^{1+\epsilon}$$

processors. This implies we need a total of

$$\sum_{i=1}^{n^{\epsilon/2}} \left(n^{1-\epsilon/2} - 1 + m(i)\right)^{1+\epsilon}$$

processors. Since

$$\sum_{i=1}^{n^{\epsilon/2}} m(i) \le n^{\epsilon/2} + m,$$

this is less than

$$(n+m)^{1+\epsilon},$$

i.e., we have enough processors. We assign processors to subarrays using the merging technique given in the proof of Lemma 4.7.

To analyze the running time of this algorithm, we note that, except for the recursive call, all of the algorithm's steps take $O(\lg(1/\epsilon))$ time. Consequently, if $T(n,m)$ denotes the time necessary to solve the row-minima problem for an $n \times m$ Monge array using $(n+m)^{1+\epsilon}$ processors on a CRCW-PRAM, then

$$T(n,m) \le O(\lg(1/\epsilon)) + \sum_{i=1}^{n^{\epsilon/2}} T(n^{1-\epsilon/2}, m(i)).$$

Since $T(1,m) = O(\lg(1/\epsilon))$, this recurrence yields $T(n,m) = O(1/\epsilon \lg(1/\epsilon))$. ■

We conclude this section by mentioning briefly two more algorithms for the row-minimization problem. The first of these algorithms is due to Apostolico, Atallah, Larmore, and McFaddin [AALM90].

Theorem 4.13 (Apostolico, Atallah, Larmore, and McFaddin [AALM90]) The row minima of an $n \times n$ Monge array A can be computed in $O(\lg n)$ time using $n \lg n$ processors on a CREW-PRAM. ■

The second algorithm, due to Atallah and Kosaraju [AK91], reduces the processor complexity of Apostolico, Atallah, Larmore, and McFaddin's algorithm down to n with no degradation in the running time.

Theorem 4.14 (Atallah and Kosaraju [AK91]) In the CREW-PRAM model, the row minima of an $n \times n$ Monge array A can be computed in $O(\lg n)$ time using n processors. ■

We remark that in [AK91], Atallah and Kosaraju also gave a more complicated EREW-PRAM version of their algorithm with the same time and processor bounds. We also note that both of the two preceding algorithms use Property 1.9.

Table 4.1 summarizes the six algorithms described in the section for computing row minima in two-dimensional Monge arrays, along with the two additional algorithms from [AALM90] and [AK91].

4.3 Plane Minima in Three-Dimensional Monge Arrays

In this section, we consider the plane-minima problem for a three-dimensional Monge array in a parallel context. Specifically, we give CREW- and CRCW-PRAM algorithms that, given an $n \times n \times n$ Monge array $A = \{a[i,j,k]\}$, for each index i, $1 \leq i \leq n$, find two more indices $j(i)$ and $k(i)$, $1 \leq j(i) \leq n$ and $1 \leq k(i) \leq n$, such that

$$a[i, j(i), k(i)] = \min_{\substack{1 \leq j \leq n \\ 1 \leq k \leq n}} a[i,j,k] .$$

Model	*Time*	*Processors*	*Theorem*
Valiant's	$O(\lg n)$	n	4.6
CRCW	$O((\lg n/\lg r)(r+\lg\lg n))$	n/r	4.8
	$O(\lg n)$	n	4.10
	$O((1/\epsilon)\lg(1/\epsilon))$	$n^{1+\epsilon}$	4.12
CREW	$O((\lg n/\lg r)(r+\lg n))$	n/r	4.8
	$O(\lg n\lg\lg n)$	$n/\lg\lg n$	4.11
	$O(\lg n)$	$n\lg n$	4.13
	$O(\lg n)$	n	4.14

Table 4.1: The row minima of an $n \times n$ Monge array can be computed using time and processors as given by this table.

The natural divide-and-conquer approach yields a simple solution for the plane-minima problem: we merely find the minimum entry in the $\lfloor n/2 \rfloor$th plane (corresponding to those entries whose first index $i = \lfloor n/2 \rfloor$) and then recurse on the two smaller plane-minima problems that remain. This gives the following lemma.

Lemma 4.15 If we can compute the row minima of an $n \times n$ two-dimensional Monge array in $T(n)$ time using $P(n)$ processors in either the CREW-PRAM or the CRCW-PRAM model, we can compute the plane minima of an $n \times n \times n$ three-dimensional Monge array in $O(T(n)\lg n)$ time using $P(n)$ processors in the same model. ■

Combining this lemma with Theorem 4.14 gives an $O(\lg^2 n)$-time, $(n/\lg\lg n)$-processor CREW-PRAM algorithm for computing the plane minima of an $n \times n \times n$ Monge array. It remains open whether this time complexity can be improved, even for the stronger CRCW-PRAM model or Valiant's comparison model.

4.4 Tube Minima in Three-Dimensional Monge Arrays

In this section, we consider the tube-minima problem for a three-dimensional Monge array in a parallel context. Specifically, we describe optimal CREW- and CRCW-PRAM algorithms that, given an $n \times n \times n$ Monge array $A = \{a[i,j,k]\}$, for each pair of indices (i,k), $1 \le i \le n$ and $1 \le k \le n$, find a third index $j(i,k)$, $1 \le j(i,j) \le n$, such that $a[i,j(i,k),k] = \min_{1 \le j \le n} a[i,j,k]$.

We can obtain a naive solution for the tube-minima problem by noting that the tube minima of a three-dimensional array are just the row minima of the planes corresponding to fixed values of the first index. Thus, to compute the tube minima of a three-dimensional $n \times n \times n$ Monge array, we need only find the row minima of n two-dimensional $n \times n$ Monge arrays. By Theorems 4.11 and 4.10, this can be done in $O(\lg n \lg\lg n)$ time using $n^2/\lg\lg n$ processors on a CREW-PRAM and in $O(\lg n)$ time using n^2 processors on a CRCW-PRAM.

This naive approach gives us an $O(n^2 \lg n)$ processor-time product for both CREW- and CRCW-PRAMs. Using the following lemma, we can reduce this product to $O(n^2)$ in both models. (Note that this is optimal, since an $n \times n \times n$ arrays has n^2 tube minima.)

Lemma 4.16 Suppose $\mathcal{T}$ is a CREW-PRAM algorithm that computes the tube minima of an $n \times n \times n$ array $A = \{a[i,j,k]\}$ in $T(n)$ time using n^2 processors. Then we can use $\mathcal{T}$ to obtain another CREW-PRAM algorithm that computes the tube minima of A in $O(T(n) + \lg n)$ time using $n^2/T(n)$ processors. Similarly, suppose $\mathcal{T}'$ is a CRCW-PRAM algorithm that computes the tube minima of A in $T'(n)$ time using n^2 processors. Then we can use $\mathcal{T}'$ to obtain another CRCW-PRAM algorithm that computes the tube minima of A in $O(T'(n) + \lg\lg n)$ time using $n^2/T'(n)$ processors.

Proof We only prove this lemma for CREW-PRAMs; the proof for CRCW-PRAMs is similar. Also, for the sake of simplicity, we assume $T(n)$ divides n.

For $0 \leq \ell < T(n)$, let $B_\ell = \{b_\ell[i,j,k]\}$ be the $(n/T(n)) \times (n/T(n)) \times (n/T(n))$ subarray of A where $b_\ell[i,j,k] = a[iT(n), jT(n), k + \ell n/T(n)]$. Since there are $T(n)$ subarrays B_ℓ and we have $n^2/T(n)$ total processors, we can assign $n^2/(T(n))^2)$ processors to each B_ℓ. Moreover, the processors assigned to B_ℓ can solve the tube-minima problem for B_ℓ in $T(n/T(n)) \leq T(n)$ time using the algorithm $\mathcal{T}$.

Now consider the $(iT(n), jT(n))$-tube of A, where $1 \leq i \leq n/T(n)$ and $1 \leq j \leq n/T(n)$. The minimum entry in this tube is simply the minimum of the corresponding tube minima in $B_0, \ldots, B_{T(n)-1}$. Thus, we can apply Lemma 4.1 and obtain the minimum entry in every $(iT(n), jT(n))$-tube of A in $O(\lg n)$ time using $n^2/T(n)$ processors.

Next, consider the $n/T(n)$ planes of A corresponding to those values of A's second index that are multiples of $T(n)$. In each of these planes, we know the minimum entry in every $T(n)$th

row (it is one of the tube minima we have already computed). Thus, we can apply Lemma 4.7 and fill in the rest of these planes' row minima in $O(T(n) + \lg n)$ time using $n/T(n)$ processors per plane (i.e., $n^2/(T(n))^2$ total processors).

Finally, consider the n planes of A corresponding to all possible values of the first index. In each of these planes, we now know the minimum entry in every $T(n)$th row. Thus, by again applying Lemma 4.7, we can fill in the rest of these planes' row minima (corresponding to the remaining tube minima of A) in $O(T(n) + \lg n)$ time using $n/T(n)$ processors per plane (i.e., $n^2/T(n)$ total processors). ∎

Applying Lemma 4.16 to the naive algorithms for computing the tube minima of an $n \times n \times n$ array, we obtain an $O(\lg n \lg\lg n)$-time, $(n^2/\lg n \lg\lg n)$-processor CREW-PRAM algorithm and an $O(\lg n)$-time, $(n^2/\lg n)$-processor CRCW-PRAM algorithm. Both these algorithms have optimal processor-time products. However, as we show in the next two theorems, we can obtain better time bounds without any deterioration in the asymptotics of the processor-time product.

Theorem 4.17 In the CREW-PRAM model, we can compute the tube minima of an $n \times n \times n$ Monge array $A = \{a[i,j,k]\}$ in $\Theta(\lg n)$ time using $n^2/\lg n$ processors.

Proof For simplicity, we assume $n = 2^{2^s}$ for some positive integer s. By Lemma 4.16, it suffices to show that n^2 processors can compute the tube minima of A in $O(\lg n)$ time on a CREW-PRAM. We first show that $n^2 \lg n$ processors are sufficient for computing the minima in $O(\lg n)$ time and then use Brent's theorem to reduce the number of processors to n^2. We use a divide-and-conquer technique similar to that used for Theorem 4.10.

Let $B = \{b[i,j,k]\}$ be the $\sqrt{n} \times \sqrt{n} \times n$ subarray of A where $b[i,j,k] = a[i\sqrt{n}, j\sqrt{n}, k]$. Since B has n^2 entries, we can apply Lemma 4.1 and compute the tube minima of B in $O(\lg n)$ time using $n^2/\lg n$ processors. This yields the minimum entry in the $(i\sqrt{n}, j\sqrt{n})$-tube of A for all i and j between 1 and $\sqrt{n}$.

For $1 \le i \le \sqrt{n}$ and $1 \le j \le \sqrt{n}$, let $k(i,j)$ denote the third index of the minimum entry in the $(i\sqrt{n}, j\sqrt{n})$-tube of A. Let $k(0,j) = 1$ for $0 \le j \le \sqrt{n}$ and let $k(i,0) = 1$ for $0 \le i \le \sqrt{n}$. Furthermore, let $C_{i,j}$ denote the three-dimensional array formed by taking the entries $a[x,y,z]$

of A such that

$$\begin{aligned} (i-1)\sqrt{n}+1 &\leq x \leq i\sqrt{n}, \\ (j-1)\sqrt{n}+1 &\leq y \leq j\sqrt{n}, \text{ and} \\ k(i-1,j-1) &\leq z \leq k(i,j). \end{aligned}$$

Since A is Monge, the tube minima of these $C_{i,j}$ are precisely the tube minima of A.

Now partition each $C_{i,j}$ into $c(i,j)$ subarrays, where

$$c(i,j) = \left\lceil \frac{k(i,j)-k(i-1,j-1)+1}{\sqrt{n}} \right\rceil.$$

Specifically, for $1 \leq w \leq c(i,j)$, let $S_{i,j,w}$ be the subarray of $C_{i,j}$ that contains those entries $a[x,y,z]$ of A such that

$$\begin{aligned} (i-1)\sqrt{n}+1 &\leq x \leq i\sqrt{n}, \\ (j-1)\sqrt{n}+1 &\leq y \leq j\sqrt{n}, \text{ and} \\ k(i-1,j-1)+(w-1)\sqrt{n} &\leq z \leq \min\{k(i-1,j-1)+w\sqrt{n}-1, k(i,j)\}. \end{aligned}$$

Note that the arrays $S_{i,j,1}, S_{i,j,2}, \ldots, S_{i,j,c(i,j)-1}$ are all $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$. Let

$$d(i,j) = k(i,j) - k(i-1,j-1) - (c(i,j)-1)\sqrt{n},$$

and let $S'_{i,j,c(i,j)}$ denote the subarray formed by taking every (u,v)-tube of $S_{i,j,c(i,j)}$ where u and v are multiples of $\lfloor \sqrt{n}/d(i,j) \rfloor$; this implies $S'_{i,j,c(i,j)}$ is a $d(i,j) \times d(i,j) \times d(i,j)$ array.

For $1 \leq i \leq \sqrt{n}$, $1 \leq j \leq \sqrt{n}$, and $1 \leq w < c(i,j)$, we assign $(\sqrt{n})^2 \lg(\sqrt{n}) = (n \lg n)/2$ processors to $S_{i,j,z}$. For $1 \leq i \leq \sqrt{n}$ and $1 \leq j \leq \sqrt{n}$, we assign $(d(i,j))^2 \lg d(i,j)$ processors to $S'_{i,j,c(i,j)}$. This assignment of processors can be done using the merging technique described in the proof of Lemma 4.7; this takes $O(\lg n)$ time using $n^2 \lg n$ processors on a CREW-PRAM.

The total number of processors assigned is

$$\sum_{i=1}^{\sqrt{n}}\sum_{j=1}^{\sqrt{n}}\left((c(i,j)-1)\frac{n\lg n}{2}+(d(i,j))^2\lg d(i,j)\right) \le \frac{\sqrt{n}\lg n}{2}\sum_{i=1}^{\sqrt{n}}\sum_{j=1}^{\sqrt{n}}(k(i,j)-k(i-1,j-1)+1).$$

Since A is Monge, we know that for $-\sqrt{n} < \ell < \sqrt{n}$,

$$\sum_{\substack{1\le i\le\sqrt{n}\\ 1\le j\le\sqrt{n}\\ i-j=\ell}} (k(i,j)-k(i-1,j-1)+1) \le n+\sqrt{n}-\ell.$$

Summing over all ℓ, we obtain

$$\sum_{i=1}^{\sqrt{n}}\sum_{j=1}^{\sqrt{n}}(k(i,j)-k(i-1,j-1)+1) \le 2n\sqrt{n}.$$

Thus, $n^2\lg n$ processors suffice.

Once the processors are assigned, we recursively compute the tube minima of the arrays $S_{i,j,1},\ldots,S_{i,j,c(i,j)-1}$ and $S'_{i,j,c(i,j)}$ for all i and j. Then we use the tube minima of $S'_{i,j,c(i,j)}$ to compute the tube minima of $S_{i,j,c(i,j)}$. Specifically, we assign n processors to each $S'_{i,j,c(i,j)}$ and then consider the $d(i,j)$ planes of $S_{i,j,c(i,j)}$ corresponding to those values of the second index that are multiples of $\lfloor\sqrt{n}/d(i,j)\rfloor$. In each of these planes, we know the minimum entry in every $\lfloor\sqrt{n}/d(i,j)\rfloor$th row (it is one of the tube minima of $S'_{i,j,c(i,j)}$). Thus, we can apply Lemma 4.9 and fill in the rest of these planes' row minima in $O(\lg d(i,j)) = O(\lg n)$ time using $\sqrt{n}/\lg d(i,j)$ processors per plane (i.e., $\sqrt{n}d(i,j)/\lg d(i,j) \le n$ total processors). Then we consider the $\sqrt{n}$ planes of $S_{i,j,c(i,j)}$ corresponding to all possible values of the first index. In each of these planes, we now know the minimum entry in every $\lfloor\sqrt{n}/d(i,j)\rfloor$th row. Thus, by again applying Lemma 4.9, we can fill in the rest of these planes' row minima (corresponding to the remaining tube minima of $S_{i,j,c(i,j)}$) in $O(\lg n)$ time using $\sqrt{n}/\lg d(i,j)$ processors per plane (i.e., $n/\lg d(i,j) \le n$ total processors).

Finally, for $1 \le i \le \sqrt{n}$ and $1 \le j \le \sqrt{n}$, we compute the minimum entry in the (x,y)-tube of A, $(i-1)\sqrt{n}+1 \le x \le i\sqrt{n}$ and $(j-1)\sqrt{n}+1 \le y \le j\sqrt{n}$, by taking the minimum of the $c(i,j)$ entries obtained for this tube in the previous step. If $c(i,j) = 1$, then we already

know the minima for these tubes; otherwise, we spread out the processors originally assigned to $S_{i,j,1}, \ldots, S_{i,j,c(i,j)}$, assigning at least $c(i,j)-1$ processors to each of the n tubes of A associated with these arrays. The processors assigned to any particular tube can then compute that tube's minimum entry from the $c(i,j)$ candidates obtained from $S_{i,j,1}, \ldots, S_{i,j,c(i,j)}$ in $O(\lg c(i,j)) = O(\lg n)$ time using Lemma 4.1.

Clearly, the above algorithm uses only $n^2 \lg n$ processors. To analyze the time complexity, observe that all steps of our algorithm, except the recursive call, take $O(\lg n)$ time. Consequently, if $T(n)$ denotes the time to compute the tube minima of an $n \times n \times n$ Monge array, then the recursive step takes at most $T(\sqrt{n})$ time, which implies

$$T(n) \leq T(\sqrt{n}) + O(\lg n).$$

Since $T(1) = O(1)$, this recurrence has solution $T(n) = O(\lg n)$.

Now consider the number of operations required. All the steps of our algorithm, other than the recursive call, can be done with $O(n^2)$ operations. Thus, if $W(n)$ denotes the number of operations required to compute the tube minima of an $n \times n \times n$ Monge array,

$$W(n) \leq O(n^2) + \sum_{i=1}^{\sqrt{n}} \sum_{j=1}^{\sqrt{n}} ((c(i,j)-1)W(\sqrt{n}) + W(d(i,j))).$$

Using our bound on the sum over all i and j of $k(i,j) - k(i-1,j-1) + 1$ and noting that $W(1) = O(1)$, we obtain $W(n) = O(n^2 \lg n)$. Thus, by Brent's theorem, we can reduce the number of processors to $W(n)/T(n) = O(n^2)$. ■

Theorem 4.18 In the CRCW-PRAM model, we can compute the tube minima of an $n \times n \times n$ Monge array $A = \{a[i,j,k]\}$ in $O((\lg \lg n)^2)$ time using $n^2/(\lg \lg n)^2$ processors.

Proof For the sake of simplicity, we assume $n = 2^{2^s}$ for some positive integer s. Our algorithm to compute the tube minima of A consists of $\lg \lg n$ phases. Each phase will require $O(\lg \lg n)$ time; thus, the entire algorithm will run in $O((\lg \lg n)^2)$ time.

For $0 \leq \ell \leq \lg \lg n$, let $\alpha(\ell) = 2^{-\ell}$. Furthermore, let $B_\ell = \{b_\ell[i,j,k]\}$ denote the $n^{1-\alpha(\ell)} \times n^{1-\alpha(\ell)} \times n$ subarray of A where $b_\ell[i,j,k] = a[in^{\alpha(\ell)}, jn^{\alpha(\ell)}, k]$. In the first phase of our algorithm,

we compute the tube minima of B_1. Since B_1 contains only n^2 entries, we can apply Lemma 4.2 and compute the tube minima of B_1 in $O(\lg\lg n)$ time using $n^2/\lg\lg n$ processors.

In the ℓth phase of the algorithm, $1 < \ell \leq \lg\lg n$, we compute the tube minima of B_ℓ using the tube minima of $B_{\ell-1}$. We first consider the planes of B_ℓ corresponding to values of the second index that are multiples of $n^{\alpha(\ell-1)}$. In each of these $n^{1-\alpha(\ell)-\alpha(\ell-1)}$ planes, we already know the minimum entry in every rth row, where $r = n^{1-\alpha(\ell)}/n^{1-\alpha(\ell-1)} = n^{\alpha(\ell)}$ (it is one of the tube minima of $B_{\ell-1}$). Since B_ℓ is Monge, this means there are at most $(n^{\alpha(\ell)} - 1)(n + n^{1-\alpha(\ell-1)}) \leq n^{1+\alpha(\ell)}$ total entries that we need to check for the remaining row minima. Thus, we can assign a processor to each of these entries (using the merging technique given in the proof of Lemma 4.7) and compute the remaining row minima in $O(\lg\lg n)$ time using $n^{1+\alpha(\ell)}$ operations per plane. Furthermore, by applying Brent's theorem, we can reduce the number of processors required per plane to $n^{1+\alpha(\ell)}/\lg\lg n$. Since there are $n^{1-\alpha(\ell)-\alpha(\ell-1)}$ planes, only $n^{2-\alpha(\ell-1)}/\lg\lg n$ total processors are required.

Next, we consider the planes of B_ℓ corresponding to fixed values of the first index. In each of these $n^{1-\alpha(\ell)}$ planes, we now know the minimum entry in every $n^{\alpha(\ell)}$th row. Using the same technique we used for the planes corresponding to fixed values of the second index, we can fill in the remaining row minima in these planes (thereby obtaining the remaining tube minima of B_ℓ) in $O(\lg\lg n)$ time using $n^{1+\alpha(\ell)}/\lg\lg n$ processors per plane. Moreover, since there are $n^{1-\alpha(\ell)}$ such planes, $n^2/\lg\lg n$ total processors suffice.

Since $B_{\lg\lg n} = A$, after $\lg\lg n$ phases, we will have computed all of A's tube minima. ■

Note that we really do require two different techniques for computing the tube minima of a three-dimensional Monge array, one for CREW-PRAMs and one for CRCW-PRAMs — applying the approach of Theorem 4.17 to CRCW-PRAMs only gives us an $O((\lg\lg n)^2)$-time, $(n^2\lg n/(\lg\lg n)^2)$-processor CRCW-PRAM algorithm, and applying the approach of Theorem 4.18 to CREW-PRAMs only gives us an $O(\lg n\lg\lg n)$-time, $(n^2/\lg n\lg\lg n)$-processor CREW-PRAM algorithm.

Implicit in [AKL+89] and [AALM90] are algorithms for the tube-minima problem; our results improve their time bounds by factors of $\lg n$ and $\lg n/\lg\lg n$ without any deterioration in the processor-time product. In [AP88], we sketch an $O(\lg n)$-time, $(n^2/\lg n)$-processor CREW-

Model	*Time*	*Processors*	*Theorem*
CREW	$\Theta(\lg n)$	$n^2/\lg n$	4.17
CRCW	$O((\lg\lg n)^2)$	$n^2/(\lg\lg n)^2$	4.18
	$O(\lg\lg n)$	$n^2/\lg\lg n$	4.19

Table 4.2: The tube minima of an $n \times n \times n$ Monge array can be computed using time and processors as given by this table.

PRAM algorithm for the tube-minima problem that uses the cascading divide-and-conquer technique of [ACG87] — this result was obtained independently in [AALM90].

Note that the time complexity of our CREW-PRAM algorithm is optimal, since just computing the minimum of n numbers requires $\Omega(\lg n)$ time. The time complexity of our CRCW-PRAM algorithm, on the other hand, does not achieve the lower bound of $\Omega(\lg\lg n)$ we have on the time necessary to compute the minimum of n numbers with a CRCW-PRAM. However, this bound is achieved by a CRCW-PRAM tube-minima algorithm due to Atallah [Ata90], who proved the following theorem.

Theorem 4.19 (Atallah [Ata90]) In the CRCW-PRAM model, we can compute the tube minima of an $n\times n\times n$ Monge array $A = \{a[i,j,k]\}$ in $O(\lg\lg n)$ time using $n^2/\lg\lg n$ processors. ∎

Table 4.2 summarizes the two tube-minima algorithms described in this section, along with Atallah's improved CRCW-PRAM algorithm.

Part II

The Applications

Chapter 5

Convex-Polygon Problems

In this chapter, we present algorithms for a number of problems involving convex polygons in the plane.

5.1 Intervertex Distances

The vertices of a convex polygon are invariably easier to deal with than arbitrary points in the plane.

Consider a convex polygon P in the plane with vertices $p_1, \ldots, p_n$ in clockwise order. As Aggarwal, Klawe, Moran, Shor, and Wilber observed in [AKM+87], the distances separating pairs of vertices of P form an inverse-Monge array. Specifically, let $A = \{a[i,j]\}$ denote the $n \times (2n-1)$ array given by the equation

$$a[i,j] = \begin{cases} -\infty & \text{if } 1 \le j < i, \\ \mathrm{d}(p_i, p_j) & \text{if } i \le j \le n, \\ \mathrm{d}(p_i, p_{j-n}) & \text{if } n < j < i+n, \\ -\infty & \text{if } i+n \le j < 2n, \end{cases}$$

where $\mathrm{d}(p_i, p_j)$ denotes the Euclidean distance between p_i and p_j. (We call this array the *distance array* for P.) The finite entries in row i of A are precisely the n distances separating p_i from $p_1, \ldots, p_n$; moreover, A is Monge, as the following lemma shows.

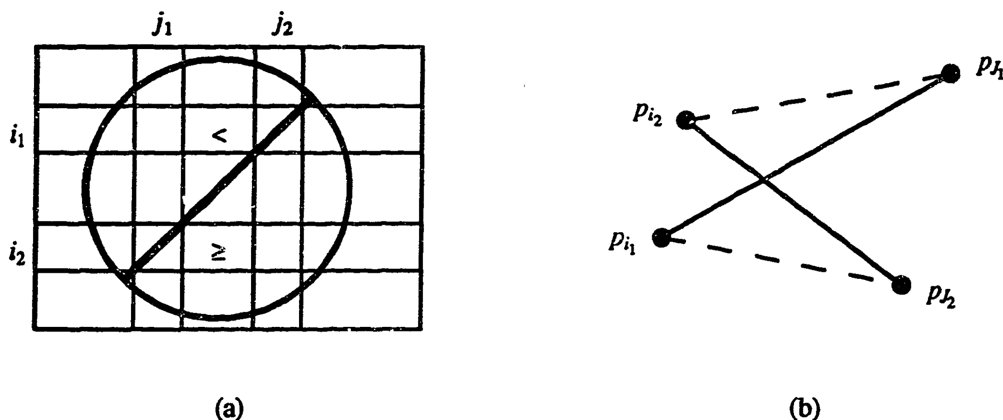

Figure 5.1: (a) In a totally monotone array, for no $i_1 < i_2$ and $j_1 < j_2$ is $a[i_1, j_1] < a[i_1, j_2]$ and $a[i_2, j_1] \geq a[i_2, j_2]$. (b) For any quadrilateral with vertices p_{i_1}, p_{i_2}, p_{J_1}, and p_{J_2} in clockwise order, $\mathrm{d}(p_{i_1}, p_{J_1}) + \mathrm{d}(p_{i_2}, p_{J_2}) > \mathrm{d}(p_{i_1}, p_{J_2}) + \mathrm{d}(p_{i_2}, p_{J_1})$.

Lemma 5.1 A is Monge.

Proof Let i_1 and i_2 denote any two rows of A, where $i_1 < i_2$, and let j_1 and j_2 denote any two columns of A, where $j_1 < j_2$. We must show that

$$a[i_1, j_1] + a[i_2, j_2] \leq a[i_1, j_2] + a[i_2, j_1] \ .$$

We consider three possibilities.

If $i_1 < i_2 \leq j_1 < j_2 < i_1 + n$, then all four entries $a[i_1, j_1]$, $a[i_1, j_2]$, $a[i_2, j_1]$, and $a[i_2, j_2]$ correspond to distances between pairs of vertices. Moreover, if we let $J_1 = ((j_1 - 1) \bmod n) + 1$ and $J_2 = ((j_2 - 1) \bmod n) + 1$, then p_{i_1}, p_{i_2}, p_{J_1}, and p_{J_2} are the vertices of a quadrilateral in clockwise order, as suggested in Figure 5.1(b). This means $a[i_1, j_1]$ and $a[i_2, j_2]$ are the lengths of the quadrilateral's two diagonals, and $a[i_1, j_2]$ and $a[i_2, j_1]$ are the lengths of two opposite sides. Now the *quadrangle inequality* tells us that the sum of the lengths of the diagonals of any quadrilateral is *strictly greater* than the sum of the lengths of two opposite sides. Thus, $a[i_1, j_1] + a[i_2, j_2] > a[i_1, j_2] + a[i_2, j_1]$.

If $j_1 < i_2$, then $a[i_2, j_1] = -\infty$. This implies $a[i_1, j_1] + a[i_2, j_2] \geq a[i_1, j_2] + a[i_2, j_1]$.

Finally, if $i_1 + n < j_2$, then $a[i_1, j_2] = -\infty$. This again implies $a[i_1, j_1] + a[i_2, j_2] \geq a[i_1, j_2] + a[i_2, j_1]$. ■

Using the row-selection algorithm of Section 3.1, we can solve two selection problems involving convex polygons in the plane. Given a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane and an integer k between 1 and n, the *k-farthest-neighbors* problem for S is that of computing k farthest neighbors for each point p_i. More precisely, for all i between 1 and n, we must find a subset $S_i \subset S$ such that $|S_i| = k$ and for all $q \in S_i$ and $q' \in S - S_i$, $\mathrm{d}(p_i, q) \geq \mathrm{d}(p_i, q')$. The *k-nearest-neighbors* problem for S is defined analogously. If the points $p_1, \ldots, p_n$ are the vertices of a convex n-gon in clockwise order, then using our algorithm for computing the k largest entries in each row of a totally monotone array, we can obtain efficient algorithms for both the k-farthest-neighbors problem and the k-nearest-neighbors problem.

To reduce the k-farthest-neighbors problem for $p_1, \ldots, p_n$ to a row-selection problem, we use the $n \times (2n-1)$ totally monotone distance array A defined at the beginning of this section. As the n largest entries in row i of A are the n distances $\mathrm{d}(p_i, p_1), \mathrm{d}(p_i, p_2), \ldots, \mathrm{d}(p_i, p_n)$, we can use our row-selection algorithm to solve the k-farthest-neighbors problem for $p_1, \ldots, p_n$ in $O(kn)$ time.

Theorem 5.2 (Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]) Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order, the farthest neighbor of each vertex v_i can be computed in $O(n)$ time. Moreover, this time bound is asymptotically optimal. ■

Theorem 5.3 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq n$, the kth farthest neighbor of each vertex v_i can be computed in $O(kn)$ time. ■

Theorem 5.4 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq n$, the kth farthest neighbor of each vertex v_i can be computed in $O(n^{3/2} \lg^2 n)$ time. ■

Theorem 5.5 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq \binom{n}{2}$, the kth farthest pair of vertices can be computed in $O(n + k \lg(n^2/k))$ time. ■

Theorem 5.6 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order, the neighbors of each v_i can be ranked by distance from v_i in $O(n^2)$ time. ■

Theorem 5.7 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k < n$, the kth farthest neighbor of each vertex v_i can be computed in $O(n^{3/2} \lg^2 n)$ time. ■

We need to mention Lee and Preparata [LP78].

To solve the k-nearest-neighbors problem for $p_1, \ldots, p_n$, we would like to use the distance array A again; however, to compute the k nearest neighbors of p_i, we need the $n - 1 + k$ smallest entries in row i, since the $n - 1$ smallest entries in this row are negative integers that do not correspond to distances. For $1 \leq k \leq \lfloor n$, our upper bound on the time to compute the $n - 1 + k$ smallest entries in A is $O(n^2)$. Thus, to obtain an $O(kn)$-time algorithm for the k-nearest-neighbors problem, we need a slightly more complicated reduction. (Note that we cannot circumvent this difficulty by replacing the negative integers in A with large positive integers, as this destroys the total monotonicity of A.)

In [LP78], Lee and Preparata consider the *nearest-neighbor* problem (the $k = 1$ special case of the k-nearest-neighbors problem) for the vertices of a convex n-gon. In obtaining an $O(n)$-time solution for this problem, they introduce an interesting property of certain convex polygons which they call the *semicircle property*. A convex polygon P with vertices $p_1, \ldots, p_n$ in clockwise order is said to possess the semicircle property if $p_2, \ldots, p_{n-1}$ lie inside the circle with diameter $\overline{p_1 p_n}$.

Lemma 5.8 (Lee and Preparata [LP78]) Let P denote a convex polygon with vertices $p_1, \ldots, p_n$ in clockwise order. If P satisfies the semicircle property, then for all i satisfying $1 \leq i \leq n$, the sequence of distances $\mathrm{d}(p_i, p_1), \mathrm{d}(p_i, p_2), \ldots, \mathrm{d}(p_i, p_n)$ is bitonic, i.e.,

$$\mathrm{d}(p_i, p_1) > \mathrm{d}(p_i, p_2) > \cdots > \mathrm{d}(p_i, p_{i-1})$$

and

$$\mathrm{d}(p_i, p_{i+1}) < \cdots < \mathrm{d}(p_i, p_{n-1}) < \mathrm{d}(p_i, p_n) .$$

■

Lee and Preparata also showed how to decompose an arbitrary convex n-gon into four convex polygons possessing the semicircle property. We use a slightly simpler decomposition,

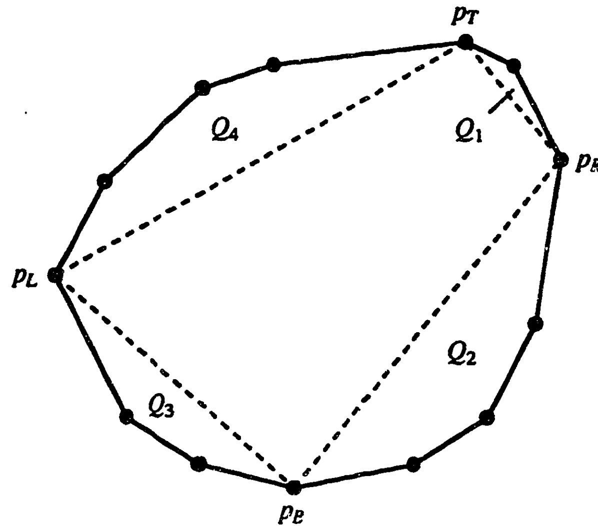

Figure 5.2: Q_1, Q_2, Q_3, and Q_4 have the semi-circle property.

due to Yang and Lee [YL79]:

Lemma 5.9 (Yang and Lee [YL79]) Let p_L and p_R denote vertices of P with minimum and maximum x-coordinates, respectively, and let p_B and p_T denote vertices of P with minimum and maximum y-coordinates, respectively. Let Q_1 denote the polygon formed by vertices p_T through p_R (i.e., p_T, p_R, and those vertices between p_T and p_R in the clockwise ordering of P's vertices). Similarly, let Q_2, Q_3, and Q_4 denote the polygons formed by vertices p_R through p_B, p_B through p_L, and p_L through p_T, respectively, as shown in Figure 5.2. Q_1, Q_2, Q_3, and Q_4 possess the semi-circle property. ■

Using this decomposition of P (which is easily computed in linear time), we can compute the k nearest neighbors of each vertex of P. We restrict our attention to the vertices of Q_1, showing that their k nearest neighbors in P can be computed in $O(kn)$ time — the computation of the k nearest neighbors of the vertices of Q_2, Q_3, and Q_4 is analogous. For each v in Q_1, the k nearest neighbors of v in Q_1 can be computed in $O(k)$ time, since by the semi-circle property, these k nearest neighbors must be within k of v in the original ordering of P's vertices. We can also compute for each v in Q_1 its k nearest neighbors in Q_2. To do this, we consider the $|Q_1| \times (|Q_2| - 1)$ array $A = \{a[i,j]\}$ where $a[i,j]$ is the distance from the i-th vertex of Q_1 to the $(j-1)$-st vertex of Q_2. (We ignore the first vertex of Q_2 since it is also the last vertex of

Q_1.) It is readily verified that A is totally monotone; moreover, the k smallest entries in row i of A correspond to the k nearest neighbors in Q_2 of the i-th vertex of Q_1. Thus, using our row-selection algorithm, we can find the k nearest neighbors in Q_2 of all the vertices in Q_1 in $O(kn)$ total time. In a similar manner, we can compute for each v in Q_1 its k nearest neighbors in Q_3 and its k nearest neighbors in Q_4. We now have $4k$ neighbors for each v in Q_1; using the linear-time selection algorithm of [BFP+73], we can select the k nearest of these neighbors in $O(k)$ additional time. This gives the k nearest neighbors in P of each v in Q_1 in $O(kn)$ total time.

Using the array-selection algorithm of Section 3.2, we can solve two more selection problems involving convex polygons in the plane. Given a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane and an integer k between 1 and $\binom{n}{2}$, the *k-farthest-pairs* problem for S is that of computing k largest values of $\mathrm{d}(p_i, p_j)$ over all unordered pairs (p_i, p_j) of points. The *k-nearest-pairs* problem for S is defined analogously. If the points $p_1, \ldots, p_n$ are the vertices of a convex n-gon in clockwise order, then using our algorithm for computing the k largest entries overall in a totally monotone array, we can obtain efficient algorithms for both the k-farthest-pairs problem and the k-nearest-pairs problem.

To reduce the k-farthest-pairs problem for $p_1, \ldots, p_n$ to a row-selection problem, we use a subarray of the $n \times (2n-1)$ distance array A defined at the beginning of this section. Specifically, we use the subarray corresponding to all n rows of A and its first n columns. Since the $\binom{n}{2}$ largest entries overall in this subarray are the $\binom{n}{2}$ distances corresponding to all unordered pairs of vertices, and since both the subarray and its transpose are totally monotone (because A and its transpose are totally monotone), we can use our array-selection algorithm to solve the k-farthest-pairs problem for $p_1, \ldots, p_n$ in $O(n + k \lg(t^2/k))$ time, where $t = \min\{n, k\}$.

Similarly, to solve the k-nearest-pairs problem for $p_1, \ldots, p_n$, we use nearly the same reduction that we used for the k-nearest-neighbors problem, except that here we must again insure that for all unordered pairs (p_i, p_j) of points, only one of $\mathrm{d}(p_i, p_j)$ and $\mathrm{d}(p_j, p_i)$ is among the distances we consider. Applying our array-selection algorithm then allows us to solve the k-nearest-pairs problem for $p_1, \ldots, p_n$ in $O(n + k \lg(t^2/k))$ time, where $t = \min\{n, k\}$.

As an application of the row-sorting algorithm given in Section 3.3, we consider the *neighbor-ranking* problem: given a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane, for each p_i, sort the

other vertices of S by distance from p_i.

If $p_1, \ldots, p_n$ are the vertices of a convex polygon P in clockwise order, then we can solve the neighbor-ranking problem for P using the $n \times (2n-1)$ totally monotone distance array $A = \{a[i,j]\}$ defined at the beginning of this section. Specifically, the i-th row of A contains the distances $d(p_i, p_1), \ldots, d(p_i, p_n)$, along with $n-1$ negative entries; thus, sorting the rows of A using our row-sorting algorithm gives an $O(n^2)$-time solution to the neighbor-ranking problem for P.

Theorem 5.10 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order, the nearest neighbor of each vertex v_i can be computed in $O(n)$ time. Moreover, this time bound is asymptotically optimal. ■

Theorem 5.11 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq n$, the kth nearest neighbor of each vertex v_i can be computed in $O(kn)$ time. ■

Theorem 5.12 Given a convex n-gon P with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq \binom{n}{2}$, the kth nearest pair of vertices can be computed in $O(n + k \lg(n^2/k))$ time. ■

5.2 Maximum-Perimeter Inscribed d-Gons

In this section, we apply the Monge-array abstraction to the problem of finding maximum-perimeter inscribed polygons. Given an n-vertex convex polygon P and an integer d between 3 and n, we want to find a maximum-perimeter convex d-gon Q contained in P. Note that each of Q's d vertices must be vertices of P.

The maximum-perimeter inscribed d-gon problem has been widely studied. We should mention Boyce, Dobkin, Drysdale, and Guibas [BDDG85] and Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]. We present the latter result as an algorithm for searching in cycle-decomposable Monge-composite arrays.

In describing our algorithm for finding maximum-perimeter inscribed d-gons, we use the following conventions. For any convex m-gon R, we let $v_1^R, \ldots, v_m^R$ and $e_1^R, \ldots, e_m^R$ denote R's

vertices and edges, respectively, in counterclockwise order, where e_i^R connects v_i^R and $v_{(i+1)\bmod m}^R$. We use the letter P for the convex n-gon given as input to the problem and the letter Q for convex d-gons inscribed in P.

We also need the following definition. Two inscribed polygons Q and Q' *interleave* if the vertices of Q and Q' alternate. In other words, between every two consecutive vertices of Q (in the counterclockwise ordering of vertices and edges of P) is a vertex of Q' (perhaps one of the two consecutive vertices of Q). Similarly, between every two consecutive vertices of Q' is a vertex of Q.

We will now describe an algorithm for finding a maximum-perimeter inscribed d-gon Q. Recall the second Monge distance array $A' = \{a'[i,j]\}$ defined in the previous section.

By summing d-dimensional extensions of A' and its transpose, we obtain the d-dimensional array $B = \{b[i_1, i_2, \ldots, i_d]\}$, where

$$b[i_1, i_2, \ldots, i_d] = a'[i_1, i_2] + a'[i_2, i_3] + \cdots + a'[i_{d-1}, i_d] + a'[i_1, i_d] .$$

B is clearly a cycle-decomposable Monge-composite array. Furthermore, B contains an entry corresponding to every possible inscribed d-gon. Specifically, the perimeter of the inscribed d-gon with vertices $v_{i_1}^P, v_{i_2}^P, \ldots, v_{i_d}^P$, where $i_1 < i_2 < \cdots < i_d$, is $b[i_1, i_2, \ldots, i_d]$. Moreover, only those entries corresponding to inscribed d-gons are finite; thus, to find a maximum-perimeter inscribed d-gon, we need only find a maximum entry in A.

By applying Theorem 2.10 directly, we can find this entry in $O(dn \lg n)$ time. However, this time complexity can be reduced to $O(dn + n \lg n)$ using a theorem concerning the perimeter of interleaving d-gons. For every i in the range $1 \le i \le n$, let Q_i denote a maximum-perimeter inscribed d-gon whose first vertex is v_i^P. (In other words, Q_i's perimeter is maximum among all inscribed d-gons whose first vertex is v_i^P.)

Theorem 5.13 (Boyce, Dobkin, Drysdale, and Guibas) For all i in the range $1 \le i \le n$ and all i' in the range $1 \le i' \le n$, Q_i and $Q_{i'}$ interleave. ■

Returning to the problem of finding a maximum-perimeter inscribed d-gon, note that finding a maximum-perimeter flush inscribed d-gon Q_1 whose first vertex is v_1^P is equivalent to finding a maximum entry in first plane of A. This can be done in $O(dn)$ time (since this plane is

path-decomposable). Let $v_{j_1}^P, \ldots, v_{j_d}^P$ be the vertices of Q_1 (by definition, $j_1 = 1$; by convention, $j_{d+1} = n$). These vertices define d intervals $I_1, \ldots, I_d$ of vertices from P, where $I_k = [v_{j_k}^P, v_{j_{k+1}}^P]$. Theorem 5.13 tells us that we need only consider vertices in I_k for the k-th vertex of a maximum-perimeter inscribed d-gon. In other words, if we let $n_k = j_{k+1} - j_k + 1$ for all k in the range $1 \leq k \leq d$, we need only search the $n_1 \times n_2 \times \cdots \times n_d$ subarray of B containing those entries $a[i_1, i_2, \ldots, i_d]$ where $j_k \leq i_k \leq j_{k+1}$ for all k between 1 and d. Since

$$\sum_{k=1}^{d} n_k = O(n)$$

and every subarray of a cycle-decomposable array is also cycle-decomposable, we can use Theorem 2.10 to find a maximum entry in this subarray, corresponding to a maximum-perimeter flush inscribed d-gon, in $O(n \lg n)$ additional time. This gives the entire algorithm a time complexity of $O(dn + n \lg n)$.

Theorem 5.14 (Aggarwal, Klawe, Moran, Shor, and Wilber) Given a convex n-gon P and an integer d in the range $3 \leq d \leq n$, a maximum-perimeter d-gon Q contained in P can be computed in $O(dn + n \lg n)$ time. ■

5.3 Minimum-Area Circumscribing d-Gons

In this section, we apply the Monge-array abstraction to the problem of finding minimum-area circumscribing polygons. Given an n-vertex convex polygon P and an integer d between 3 and n, we want to find a minimum-area d-gon Q containing P. Note that Q must clearly be convex, and each of its d edges must contact P. Also note that if we can find area-optimal d-gons circumscribing convex n-gons, then we can also find area-optimal convex d-gons containing arbitrary sets of points in the plane, since any convex polygon containing a set of points must contain the points' convex hull.

The minimum-area circumscribing d-gon problem has been widely studied. In [CY84], Chang and Yap showed that a minimum-area circumscribing d-gon can be found in $O(n^3 \lg d)$ time using dynamic programming. Aggarwal, Chang, and Yap [ACY85] then improved this result to $O(n^2 \lg d \lg n)$ time, and it is further improved to $O(n^2 \lg d)$ in [AKM+87]. We should also mention Klee and Laskowski [KL85] and O'Rourke, Aggarwal, Maddila, and Baldwin

[OAMB86]. We extend (in a non-trivial manner) the techniques of Boyce et al. [BDDG85] for finding maximum-perimeter inscribed d-gons to obtain an $O(dn + n \lg n)$ time algorithm for the minimum-area-circumscribing-d-gon problem.

In describing our algorithms for finding minimum-area circumscribing d-gons, we use the following conventions. For any convex m-gon R, we let $v_1^R, \ldots, v_m^R$ and $e_1^R, \ldots, e_m^R$ denote R's vertices and edges, respectively, in counterclockwise order, where e_i^R connects v_i^R and $v_{(i+1) \bmod m}^R$. We use the letter P for the convex n-gon to be circumscribed and the letter Q for convex d-gons circumscribing P.

We also need the following definitions.

1. If an edge e_i^Q of a circumscribing polygon Q touches P (which it must, if Q has minimal area or perimeter), then its *contact point* is the part of P that it touches. A contact point is always either an edge or a vertex of P.

2. Two circumscribing polygons Q and Q' *interleave* if the contact points of Q and Q' alternate. In other words, between every two consecutive contact points of Q (in the counterclockwise ordering of vertices and edges of P) is a contact point of Q' (perhaps one of the two consecutive contact points of Q). Similarly, between every two consecutive contact points of Q' is a contact point of Q.

3. An edge e_i^Q of Q is *flush* with P if its contact point is an edge. Q itself is flush with P if all its edges are flush with P.

4. An edge e_i^Q of Q is *balanced* if its midpoint lies on P.

5. An edge e_i^Q of Q determines two half-planes; let H_i denote the half-plane that does not contain P. e_i^Q is a *c-edge* if the lines containing its neighbors e_{i-1}^Q and e_{i+1}^Q converge (i.e., intersect) in H_i or are parallel; otherwise, e_i^Q is a *d-edge*. Equivalently, e_i^Q is a c-edge if the sum of the two internal angles of Q corresponding to e_i^Q's endpoints is greater than or equal to π, and e_i^Q is a d-edge if this sum is less than π.

Our algorithm for finding a minimum-area circumscribing d-gon has two parts. First, we restrict our attention to *flush* circumscribing d-gons and use the techniques of [BDDG85] to find one with minimal area. Then, we use this minimum-area flush d-gon (and a lemma due to

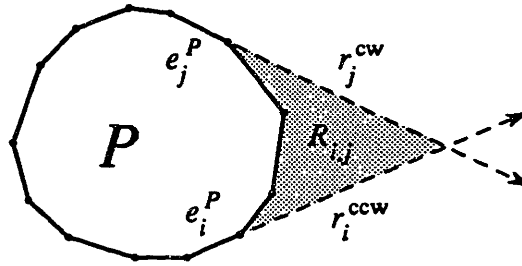

Figure 5.3: If r_i^{ccw} intersects r_j^{cw}, then $R_{i,j}$ is the shaded region between r_i^{ccw}, r_j^{cw}, and P.

DePano [DeP87]) to obtain a circumscribing d-gon, possibly not flush, whose area is minimal among *all* circumscribing d-gons.

5.3.1 Finding the Best Flush d-gon

The techniques given by Boyce et al. [BDDG85] for finding a maximum-perimeter inscribed d-gon can also be used to find a minimum-area flush circumscribing d-gon in $O(dn \lg n + n \lg^2 n)$ time. (This was pointed out in the concluding section of [BDDG85].) Furthermore, the techniques of [AKM+87] reduce the time complexity of this problem to $O(dn + n \lg n)$. For the sake of completeness, we will describe this result, recasting it in terms of multidimensional Monge arrays.

For $1 \leq i \leq n$, let r_i^{ccw} be the ray containing e_i^P with v_i^P as its origin, and let r_i^{cw} be the ray containing e_i^P with v_{i+1}^P as its origin. (The superscript of r_i^{ccw} indicates that it is a counterclockwise "extension" of e_i^P, and the superscript of r_i^{cw} indicates that it is a clockwise "extension.") If r_i^{ccw} intersects r_j^{cw}, let $R_{i,j}$ be the region outside P bounded by r_i^{ccw}, r_j^{cw}, and the edges $e_{i+1}^P, \ldots, e_{j-1}^P$ of P (the shaded region in Figure 5.3). Now consider the two-dimensional arrays $W = \{w[i,j]\}$ and $W' = \{w'[i,j]\}$, where

$$w[i,j] = \begin{cases} \text{area}(R_{i,j}) & \text{if } i < j \text{ and } r_i^{ccw} \text{ intersects } r_j^{cw}, \text{ and} \\ \infty & \text{otherwise,} \end{cases}$$

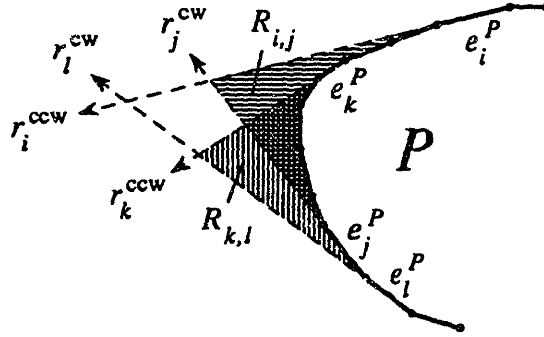

Figure 5.4: Suppose $w[i,j]$, $w[i,l]$, $w[k,j]$, and $w[k,l]$ all correspond to regions outside of P, i.e., $i < k < j < l$ and r_i^{ccw} intersects r_l^{cw}. The region $R_{i,l}$ bounded by r_i^{ccw} and r_l^{cw} contains both the region $R_{i,j}$ bounded by r_i^{ccw} and r_j^{cw} (shaded with horizontal lines) and the region $R_{k,l}$ bounded by r_k^{ccw} and r_l^{cw} (shaded with vertical lines). Moreover, the intersection of $R_{i,j}$ and $R_{k,l}$ is exactly the region $R_{k,j}$ bounded by r_k^{ccw} and r_j^{cw}.

and

$$w'[i,j] = \begin{cases} \text{area}(R_{i,j}) & \text{if } i > j \text{ and } r_i^{ccw} \text{ intersects } r_j^{cw}, \text{ and} \\ \infty & \text{otherwise.} \end{cases}$$

Lemma 5.15 Both W and W' satisfy the Monge condition.

Proof We only prove the lemma for W; the proof for W' is similar. For $1 \le i < k \le n$ and $1 \le j < l \le n$, we must show that $w[i,j] + w[k,l] \le w[i,l] + w[k,j]$. We consider two cases. If either $w[i,l] = \infty$ or $w[k,j] = \infty$, then the Monge condition follows immediately. If, on the other hand, $w[i,l]$ and $w[k,j]$ both correspond to regions outside of P ($R_{i,l}$ and $R_{k,j}$, respectively), then $w[i,j]$ and $w[k,l]$ must also correspond to regions outside of P ($R_{i,j}$ and $R_{k,l}$, respectively), and the four regions must overlap as in Figure 5.4. Now consider the region $R_{i,j} \cup R_{k,l}$. It has area $w[i,j] + w[k,l] - w[k,j]$, since the intersection of $R_{i,j}$ and $R_{k,l}$ is exactly $R_{k,j}$. Moreover, $R_{i,j} \cup R_{k,l}$ is contained in $R_{i,l}$, which implies $w[i,j] + w[k,l] - w[k,j] \le w[i,l]$ or $w[i,j] + w[k,l] \le w[i,l] + w[k,j]$. ■

By summing d-dimensional extensions of W and W', we obtain the d-dimensional array $A = \{a[i_1, i_2, \ldots, i_d]\}$, where

$$a[i_1, i_2, \ldots, i_d] = w[i_1, i_2] + w[i_2, i_3] + \cdots + w[i_{d-1}, i_d] + w'[i_d, i_1] .$$

A is clearly a Monge-composite cycle-decomposable array. Furthermore, A contains an entry corresponding to every possible flush circumscribing d-gon. Specifically, the area of the flush circumscribing d-gon with contact points $e^P_{i_1}, e^P_{i_2}, \ldots, e^P_{i_d}$, $i_1 < i_2 < \cdots < i_d$, is $a[i_1, i_2, \ldots, i_d] +$ area(P). Moreover, only those entries corresponding to circumscribing d-gons are less than ∞; thus, to find a minimum-area flush circumscribing d-gon, we need only find a minimum entry in A.

By applying Theorem 2.10 directly, we can find this entry in $O(dn \lg n)$ time. However, this time complexity can be reduced to $O(dn + n \lg n)$ using a theorem concerning the area of interleaving d-gons. (The theorem we prove is actually significantly more general than we need it to be in obtaining an $O(dn + n \lg n)$ time algorithm for the minimum-area flush circumscribing d-gon problem, but we will use it again later in this section in the context of two other related problems.)

Before we can prove this theorem, however, we first need a few more definitions. Let Q_a and Q_b denote circumscribing d-gons. We define an *edge exchange* operation as follows. Let E denote the union of Q_a and Q_b's edges. To exchange edges between Q_a and Q_b, we select a d-edge subset E' of E, and form two new circumscribing d-gons, the first consisting of the edges of E' (extended or shortened as necessary to form a circumscribing d-gon) and the second consisting of the edges of $E - E'$.

Now let $\mathcal{Q}_a$ and $\mathcal{Q}_b$ denote sets of circumscribing d-gons. We will say that $\mathcal{Q}_a$ and $\mathcal{Q}_b$ are *closed under edge exchange* if for any d-gon $Q_a \in \mathcal{Q}_a$ and any d-gon $Q_b \in \mathcal{Q}_b$, any edge exchange between Q_a and Q_b produces a d-gon in $\mathcal{Q}_a$ and a d-gon in $\mathcal{Q}_b$.

Theorem 5.16 Suppose two sets $\mathcal{Q}_a$ and $\mathcal{Q}_b$ of circumscribing d-gons are closed under edge exchange. Furthermore, suppose that Q_a has minimum perimeter among d-gons in $\mathcal{Q}_a$ and that Q_b has minimum perimeter among d-gons in $\mathcal{Q}_b$. Then Q_a and Q_b interleave.

Proof Suppose Q_a and Q_b do not interleave. Let $a_1, a_2, \ldots, a_d$ be the contact points of Q_a, and let $b_1, b_2, \ldots, b_d$ be the contact points of Q_b. Since Q_a and Q_b do not interleave, there exists at least one pair (a_{i-1}, a_i) of consecutive contact points of Q_a such that no contact points of Q_b lie between a_{i-1} and a_i (inclusive) in the counterclockwise ordering of P's vertices and edges. There also exists at least one pair (b_j, b_{j+1}) of consecutive contact points of Q_b such

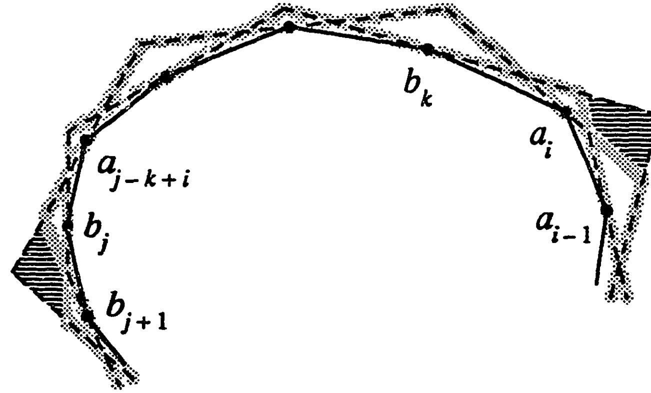

Figure 5.5: The sum of the areas of Q_a and Q_b (indicated by the dotted lines) is exactly the sum of the areas of Q'_a and Q'_b (indicated by the shaded lines) *plus* the area of the two shaded regions.

that no contact points of Q_a lie between b_j and b_{j+1} (inclusive). Moreover, there exists such a pair (a_{i-1}, a_i) and such a pair (b_j, b_{j+1}) separately only by alternating contact points. In other words, there exist i, j, and k, such that the contact points between a_{i-1} and b_{j+1} are (in order)

$$a_{i-1}, a_i, \underbrace{b_k, a_{i+1}, b_{k+1}, a_{i+2}, \ldots, b_{j-1}, a_{j-k+i}}_{\text{alternating contact points}}, b_j, b_{j+1} \ .$$

Now suppose we exchange edges between Q_a and Q_b and form a d-gon Q'_a with contact points

$$a_1, \ldots, a_{i-1}, b_k, b_{k+1}, \ldots, b_j, a_{j-k+i+1}, \ldots, a_d$$

and a d-gon Q'_b with contact points

$$b_1, \ldots, b_{k-1}, a_i, a_{i+1}, \ldots, a_{j-k+i}, b_{j+1}, \ldots, b_d \ .$$

For any two contact points c and c', let $R_{c,c'}$ denote the region outside P bounded by P and lines through the edges of Q_a or Q_b touching c and c'. As is suggested in Figure 5.5,

$$\begin{aligned} \text{area}(Q'_a) + \text{area}(Q'_b) &= \text{area}(Q_a) + \text{area}(Q_b) \\ &\quad - \text{area}(R_{a_{i-1},a_i}) - \text{area}(R_{a_{j-k+i},a_{j-k+i+1}}) \\ &\quad - \text{area}(R_{b_{k-1},b_k}) - \text{area}(R_{b_j,b_{j+1}}) \end{aligned}$$

$$+ \text{area}(R_{a_{i-1},b_k}) + \text{area}(R_{b_j,a_{j-k+i+1}})$$
$$+ \text{area}(R_{b_{k-1},a_i}) + \text{area}(R_{a_{j-k+i},b_{j+1}}) .$$

Since b_{k-1} precedes a_{i-1} in the counterclockwise ordering of P's vertices and edges,

$$\text{area}(R_{b_{k-1},a_i}) + \text{area}(R_{a_{i-1},b_k}) < \text{area}(R_{b_{k-1},b_k}) + \text{area}(R_{a_{i-1},a_i}) ,$$

by the same argument we used in the proof of Lemma 5.15. Similarly,

$$\text{area}(R_{a_{j-k+i},b_{j+1}}) + \text{area}(R_{b_j,a_{j-k+i+1}}) < \text{area}(R_{a_{j-k+i},a_{j-k+i+1}}) + \text{area}(R_{b_j,a_{j+1}}) .$$

Thus, $\text{area}(Q'_a) + \text{area}(Q'_b) < \text{area}(Q_a) + \text{area}(Q_b)$. Since $\mathcal{Q}_a$ and $\mathcal{Q}_b$ are closed under edge exchange, one of the new d-gons is in $\mathcal{Q}_a$ and the other is in $\mathcal{Q}_b$. Without loss of generality, we assume $Q'_a \in \mathcal{Q}_a$ and $Q'_b \in \mathcal{Q}_b$. Now either $\text{area}(Q'_a) < \text{area}(Q_a)$ or $\text{area}(Q'_b) < \text{area}(Q_b)$, both of which are contradictions. ■

Corollary 5.17 Let Q_i be a minimum-area flush circumscribing d-gon with e_i^P as a contact point. Every minimum-area flush circumscribing d-gon interleaves Q_i.

Proof Any edge exchange between a circumscribing d-gon with e_i^P as a contact point and an arbitrary circumscribing d-gon produces a circumscribing d-gon with e_i^P as a contact point and an arbitrary circumscribing d-gon. Thus, this corollary follows from Theorem 5.16. ■

Returning to the problem of finding a minimum-area flush circumscribing d-gon, note that finding a minimum-area flush circumscribing d-gon Q_1 with e_1^P as a contact point is equivalent to finding a minimum entry in first plane of A. This can be done in $O(dn)$ time (since this plane is path-decomposable). Let $e_{j_1}^P, \ldots, e_{j_d}^P$ be the contact points of Q_1 (by definition, $j_1 = 1$; by convention, $j_{d+1} = n$). These edges define d intervals $I_1, \ldots, I_d$ of edges from P, where $I_k = [e_{j_k}^P, e_{j_{k+1}}^P]$. Corollary 5.17 tells us that we need only consider edges in I_k for the k-th contact point of a minimum-area circumscribing d-gon. In other words, we need only search the $n_1 \times n_2 \times \cdots \times n_d$ subarray of A, $n_k = j_{k+1} - j_k + 1$, containing those entries $a[i_1, i_2, \ldots, i_d]$

where $j_k \le i_k \le j_{k+1}$ for all k between 1 and d. Since

$$\sum_{k=1}^{d} n_k = O(n)$$

and every subarray of a cycle-decomposable array is also cycle-decomposable, we can use Theorem 2.10 to find a minimum entry in this subarray, corresponding to a minimum-area flush circumscribing d-gon, in $O(n \lg n)$ additional time. This gives the entire algorithm a time complexity of $O(dn + n \lg n)$.

5.3.2 Using the Best Flush d-gon to Obtain the Best Arbitrary d-gon

In [DeP87], DePano provides the following geometric characterization of minimum-area circumscribing d-gons.

Lemma 5.18 ([DeP87]) Let P be any convex n-gon. For $3 \le d \le n$, if Q is a minimum-area d-gon Q circumscribing P, then either

1. all d edges of Q are flush with P, or
2. $d-1$ edges of Q are flush with P, and the non-flush edge is a balanced d-edge.

This lemma allows us to relate minimum-area flush circumscribing d-gons and minimum-area arbitrary circumscribing d-gons. Specifically, we have the following corollary to Theorem 5.16.

Corollary 5.19 Let Q' be a minimum-area flush circumscribing d-gon. Every minimum-area circumscribing d-gon Q interleaves Q'.

Proof Let $\mathcal{Q}'$ denote the set of all flush circumscribing d-gons, and let $\mathcal{Q}$ denote the set of all circumscribing d-gons whose first $d-1$ edges are flush with P and whose d-th edge is a balanced d-edge. By Lemma 5.18, every minimum-area circumscribing d-gon is in $\mathcal{Q} \cup \mathcal{Q}'$. Moreover, every minimum-area circumscribing d-gon has minimal area among d-gons in $\mathcal{Q} \cup \mathcal{Q}'$.

Now, $\mathcal{Q}'$ and $\mathcal{Q} \cup \mathcal{Q}'$ are clearly closed under edge exchange. Thus, by Theorem 5.16, every minimum-area circumscribing d-gon must interleave every minimum-area flush circumscribing d-gon. ■

Now suppose we have found a minimum-area flush circumscribing d-gon Q', using the techniques of [BDDG85]. Let $e^P_{j_1}, \ldots, e^P_{j_d}$ be the contact points of this d-gon. Without loss of generality, assume $j_1 = 1$ (if it is not, we can renumber the edges of P). Also, for notational convenience, let $j_{d+1} = n$. The edges $e^P_{j_1}, \ldots, e^P_{j_d}$ define d intervals $I_1, \ldots, I_d$ of edges from P, where $I_k = [e^P_{j_k}, e^P_{j_{k+1}}]$. Corollary 5.19 tells us that we need only consider edges or vertices in I_k for the k-th contact point of a minimum-area circumscribing d-gon Q. (A vertex is considered to be in I_k if *both* the edges incident to it are in I_k.) Furthermore, we can use Lemma 5.18 to show:

Lemma 5.20 There are at most three intervals that might contain the contact point of the non-flush edge of a minimum-area circumscribing d-gon Q. Moreover, we can identify these intervals in $O(d)$ time.

Proof For each edge e^P_i of P, let r^{ccw}_i be the ray containing e^P_i with v^P_i as its origin. Now for each interval I_k, consider the ray $r^{\text{ccw}}_{j_k}$ associated with the interval's first edge $e^P_{j_k}$. Let α_k be $r^{\text{ccw}}_{j_k}$'s angle with respect to r^{ccw}_1, measured in a counterclockwise direction. Clearly,

$$0 = \alpha_1 < \alpha_2 < \cdots < \alpha_d < 2\pi .$$

Now suppose the non-flush edge of Q is e^Q_k. It must contain an edge or vertex of P in the interval I_k. Moreover, the contact point of e^Q_{k-1} is an edge $e^P_i \in I_{k-1}$ and the contact point of e^Q_{k+1} is an edge $e^P_{i'} \in I_{k+1}$. Since e^P_i lies in I_{k-1}, the angle r^{ccw}_i forms with r^{ccw}_1 is at least α_{k-1}. Similarly, since $e^P_{i'}$ lies in I_{k+1}, the angle $r^{\text{ccw}}_{i'}$ forms with r^{ccw}_1 is at most α_{k+2}. Furthermore, DePano's lemma tells us that e^Q_k is a d-edge, i.e., the lines containing e^Q_{k-1} and e^Q_{k+1} intersect on P's side of the line containing e^Q_k. This implies $(\alpha_{k+2} - \alpha_{k-1}) \bmod 2\pi > \pi$, and this inequality can hold for at most three values of k.

To identify those intervals that might contain the contact point of the non-flush edge of Q, we need only compute α_k for each interval I_k and then $(\alpha_{k+2} - \alpha_{k-1}) \bmod 2\pi$ for each k, which may be done in $O(d)$ time. ■

We can check the possibility of the nonflush edge lying in the interval I_k as follows. Let e^P_i be any edge of P in I_{k-1}. Let e^P_j be any edge of P in I_{k+1}. Define $Q_{i,j}$ to be the set of all

circumscribing d-gons Q satisfying the following constraints:

1. Q's $(k-1)$-st contact point is the edge e_i^P,
2. Q's k-th contact point is an edge or vertex in I_k,
3. Q's k-th edge is a d-edge,
4. Q's $(k+1)$-st contact point is the edge e_j^P, and
5. for $1 \leq l < k-1$ and $k+1 < l \leq d$, Q's l-th contact point is an edge in I_l.

Also define

$$\mathcal{Q}_i = \bigcup_j \mathcal{Q}_{i,j} .$$

Lemma 5.21 For any $e_i^P \in I_{k-1}$, we can find a circumscribing d-gon $Q_i \in \mathcal{Q}_i$ of minimal area in $O(n)$ time.

Proof We begin with a few more definitions (borrowed from [ACY85]). An *h-sided* (i,j)*-chain* is a polygonal chain $C = \{e_1^C, \ldots, e_h^C\}$ such that

1. e_1^C is flush with e_i^P,
2. e_h^C is flush with e_j^P, and
3. for $1 < l < h$, e_l^C has a contact point in $I_{(k+l) \bmod d}$.

C is *flush* if all its edges are flush. The *extra area* of C is the area of the bounded (but perhaps disconnected) region between C and P.

For each edge $e_j^P \in I_{k+1}$, let $C_{i,j}$ be an optimal flush $(d-1)$-sided (j,i)-chain (i.e., its extra area is minimal), and let $C'_{i,j}$ be an optimal 3-sided (i,j)-chain. Combining $C_{i,j}$ and $C'_{i,j}$ gives an optimal circumscribing d-gon $Q_{i,j}$ from $\mathcal{Q}_{i,j}$. Moreover, given $Q_{i,j}$ for each $e_j^P \in I_{k+1}$, we can pick the best of these d-gons in $O(n)$ time to obtain an optimal circumscribing d-gon from $\mathcal{Q}_i$. Thus, if we can find an optimal flush $(d-1)$-sided (j,i)-chain and an optimal 3-sided (i,j)-chain for each $e_j^P \in I_{k+1}$ in $O(n)$ time, we will have established the lemma.

To find an optimal flush $(d-1)$-sided (j,i) chain for each $e_j^P \in I_{k+1}$, we first define the $(d-2)$-dimensional path-decomposable array $A = \{a[i_1, i_2, \ldots, i_{d-2}]\}$, where

$$a[i_1, i_2, \ldots, i_{d-2}] = w[i_1, i_2] + w[i_2, i_3] + \cdots + w[i_{d-k-1}, i_{d-k}] + w'[i_{d-k+1}, i_{d-k}] + w[i_{d-k+1}, i_{d-k+2}] + \cdots + w[i_{d-2}, i] .$$

(We defined the arrays $W = \{w[i,j]\}$ and $W' = \{w[i,j]\}$ earlier in this section, in the context of optimal flush circumscribing d-gons.) Now suppose $I_l = [j_l, j_{l+1}]$ for $1 \le l \le d$. Let $n_l = j_{l+1} - j_l + 1$, and let A' be the $n_1 \times n_2 \times \cdots \times n_{d-2}$ subarray of A containing entries $a[i_1, i_2, \ldots, i_{d-2}]$ that satisfy $j_{(l+k) \bmod d} \le i_l \le j_{(l+k+1) \bmod d}$ for $1 \le l \le d-2$. The optimal flush $(d-1)$-sided (j,i) chain corresponds to the minimum entry in the j-th plane of A' (the plane containing those entries of A' whose first coordinate is j). Since $\sum_{l=1}^{d-2} n_l = O(n)$, Theorem 2.9 tells us that we can compute these plane minima in $O(n)$ time.

To find an optimal 3-sided (i,j)-chain for each $e_j^P \in I_{k+1}$, we first recall that Lemma 5.18 tells us we need only consider 3-sided chains whose middle edge is a balanced d-edge. We will call this middle edge a *closing edge* for e_i^P and e_j^P.

Now suppose e_k^Q is a balanced closing d-edge for $e_i^P \in I_{k-1}$ and $e_j^P \in I_{k+1}$. e_k^Q's first endpoint must lie on the line L_i containing e_i^P, and its second endpoint must lie on the line L_j containing e_j^P. Moreover, since e_k^Q is balanced, its second endpoint must also lie on the chain C defined as follows. For each $e_l^P \in I_k$, let e_{2l-1}^C be the segment on the line containing e_l^P whose first and second endpoints are twice as far from L_i as are e_l^P's first and second endpoints, respectively. C consists of these segments, plus segments e_{2l}^C parallel to L_i connecting e_{2l-1}^C and e_{2l+1}^C for $1 \le l < |I_k|$. This is suggested in Figure 5.6. We will denote the endpoints of e_l^C by v_l^C and v_{l+1}^C.

Using what we know about the angles of rays containing $e_l^C \in C$ and $e_j^P \in I_{k+1}$ form with respect r_i^{ccw}, we can prove the following two lemmas about C and the lines L_j containing edges in I_{k+1}.

Lemma 5.22 For all $e_j^P \in I_{k+1}$, L_j intersects C at most once.

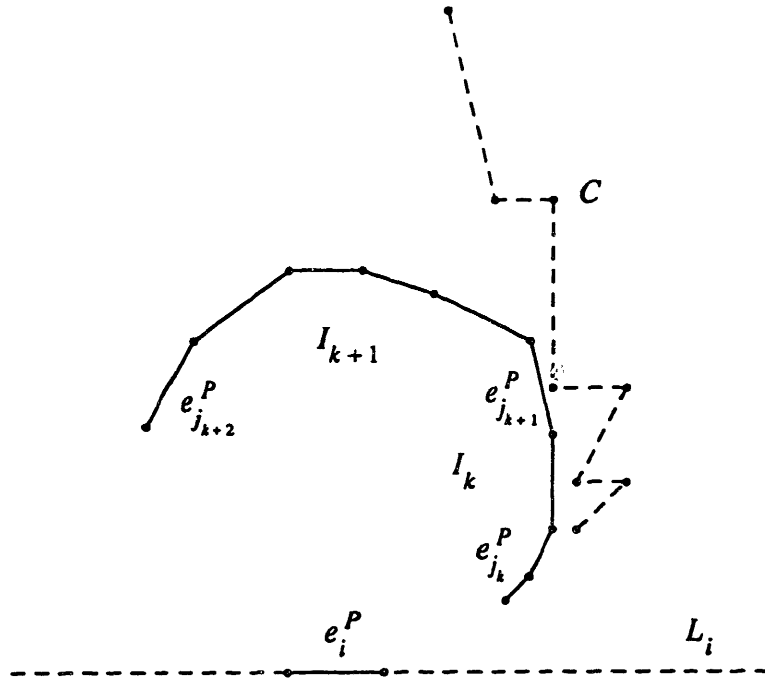

Figure 5.6: Every balanced closing d-edge for a particular $e_i^P \in I_{k-1}$ and any $e_j^P \in I_{k+1}$ must have one endpoint on the chain C.

Lemma 5.23 For all e_j^P and $e_{j'}^P$ in I_{k+1}, $j' > j$, L_j intersects C before $L_{j'}$ does, i.e., if L_j intersects e_l^C, then $L_{j'}$ intersects $e_{l'}^C$, $l' > l$.

The first lemma tells us that there is at most one balanced closing d-edge corresponding to each $e_j^P \in I_{k+1}$. The second allows us to find all these closing edges in $O(n)$ time. We begin by computing C, which can be done in $O(n)$ time. We then check whether the line L containing the first edge in I_{k+1} intersects e_1^C. If L does not intersect this segment, we check whether it intersects e_2^C. If again there is no intersection, we move on to e_3^C. We continue in this manner until we find the segment e_l^C that L intersects. This gives us the closing edge for the first edge of I_{k+1} Next, we check whether the line L' containing the second edge of I_{k+1} intersects e_l^C. If there is no intersection, we move on to segment e_{l+1}^C. We continue in this manner until all closing edges are found. It is easy to see that only $O(n)$ time is required. Also, Lemma 5.23 guarantees that this approach will find all of the desired intersections, which in turn gives us the optimal 3-sided (i, j)-chains. ■

To complete our algorithm, we require a third corollary to Theorem 5.16.

Corollary 5.24 For any e_i^P and $e_{i'}^P$ in I_{k-1}, every optimal $Q_i \in \mathcal{Q}_i$ interleaves every optimal $Q_{i'} \in \mathcal{Q}_{i'}$.

Proof This corollary follows immediately from Theorem 5.16, since $\mathcal{Q}_i$ and $\mathcal{Q}_{i'}$ are closed under edge exchange. ■

This corollary, together with Lemma 5.21, allows us to find an optimal circumscribing d-gons for each $\mathcal{Q}_i$ such that $e_i^P \in I_{k-1}$ in $O(n \lg n)$ total time. We use the natural divide-and-conquer approach of Theorem 2.7. Let $I_l = [j_l, j_{l+1}]$ for $1 \leq l \leq d$, and let $n_l = j_{l+1} - j_l + 1$. We first find an optimal circumscribing d-gon $Q_i \in \mathcal{Q}_i$ for $i = j_{k-1} + \lceil n_{k-1}/2 \rceil$. By Corollary 5.24, the contact point of Q_i in interval I_l, $1 \leq l \leq d$, splits that interval into two intervals I_l' and I_l'', such that the contact points of any optimal circumscribing d-gon for $\mathcal{Q}_{i'}$, $j_{k-1} \leq i' < i$, must lie in the intervals $I_1', \ldots, I_d'$, and the contact points of any optimal circumscribing d-gon for $\mathcal{Q}_{i'}$, $i < i' < j_k$, must lie in the intervals $I_1'', \ldots, I_d''$. If we recursively solve the two subproblems associated with the intervals $I_1', \ldots, I_d'$ and the intervals $I_1'', \ldots, I_d''$, we obtain a recurrence with solution $O(n \lg n_{k-1}) = O(n \lg n)$ for the time required to find an optimal circumscribing d-gon for each $\mathcal{Q}_i$ such that $e_i^P \in I_{k-1}$.

By choosing the best of the n_{k-1} circumscribing d-gons obtained in this manner (which can be done in $O(n)$ time), we obtain a minimum-area circumscribing d-gon; thus, we have the following theorem.

Theorem 5.25 Given a convex n-gon P and an integer d in the range $3 \leq d \leq n$, a minimum-area d-gon Q containing P can be computed in $O(dn + n \lg n)$ time. ■

Chapter 6

Two Dynamic-Programming Applications

In this chapter, we present two applications of the Monge-array abstraction to problems that can be solved using dynamic programming. (Additional dynamic-programming applications are described in the following chapter.) These applications show how the on-line LIEBER algorithm of Section 2.2 can be used to speed up dynamic-programming algorithms.

The first of these applications, which we discuss in Section 6.1, involves a special case of the n-vertex traveling-salesman problem that can be solved in $O(n^2)$ time using dynamic programming. We show that, under certain circumstances, this running time can be reduced to $O(n)$ using the LIEBER algorithm. This result was first presented in [Par91].

This chapter's second application, which is described in Section 6.2, concerns a dynamic-programming recurrence studied by Yao in [Yao80]. Yao identified certain general conditions under which the $O(n^3)$ running time of the straightforward method for solving the recurrence can be reduced to $O(n^2)$. (These conditions are satisfied by many of the problems giving rise to her recurrence, including the problem of constructing an optimal binary search tree.) In this chapter, we describe both Yao's conditions and her $O(n^2)$-time algorithm, reformulated in terms of two- and three-dimensional Monge arrays. We also give an alternate $O(n^2)$-time algorithm for her problem based on the LIEBER algorithm. This latter result represents joint work with Aggarwal [AP89b].

6.1 A Special Case of the Traveling-Salesman Problem

This section presents a special case of the n-vertex traveling-salesman problem that can be solved in $O(n)$ time using the on-line LIEBER algorithm of Section 2.2. We obtain this result by speeding up a quadratic-time dynamic-programming algorithm for a more general special case of the traveling-salesman problem.

Given an n-vertex complete directed graph G whose vertices are labeled $1, \ldots, n$ and an $n \times n$ cost array $C = \{c[i,j]\}$ such that the cost of traversing arc (i,j) of G is $c[i,j]$, the ***traveling-salesman problem*** is that of computing a minimum-cost tour of G that visits each vertex exactly once. Though this famous problem is NP-complete for arbitrary C, there exist several special cases of the traveling-salesman problem, corresponding to restricted sets of cost arrays, that can be solved in polynomial time. Many of these special cases are listed in a survey article written by Gilmore, Lawler, and Shmoys [GLS85].

In this section, we will focus on one of the special cases described by Gilmore, Lawler, and Shmoys. This special case was first considered by V. M. Demidenko; he identified a set Δ of cost arrays, such that for any $C \in \Delta$, a minimum-cost traveling-salesman tour through the directed graph corresponding to C can be computed in $O(n^2)$ time[1]. The set Δ consists of all cost arrays satisfying the following conditions: if $1 \leq i < j$ and $j+1 < k \leq n$, then

$$\begin{aligned}
c[i,j] + c[j,j+1] + c[j+1,k] &\leq c[i,j+1] + c[j+1,j] + c[j,k] \\
c[j,i] + c[j+1,j] + c[k,j+1] &\leq c[j+1,i] + c[j,j+1] + c[k,j] \\
c[i,j] + c[k,j+1] &\leq c[i,j+1] + c[k,j] \\
c[j,i] + c[j+1,k] &\leq c[j+1,i] + c[j,k] \,.
\end{aligned}$$

These conditions, which Gilmore, Lawler, and Shmoys call the *Demidenko conditions*, are depicted graphically in Figure 6.1. Note that every square (i.e. $s \times s$ for some s) Monge array satisfies the Demidenko conditions, as the following lemma shows.

Lemma 6.1 If an $n \times n$ array A is Monge, then it satisfies the Demidenko conditions.

[1] As I do not have easy access to Demidenko's 1979 Russian-language paper describing his result (see [GLS85] for the reference) nor do I read Russian, this section is based solely on Gilmore, Lawler, and Shmoys's presentation of the result.

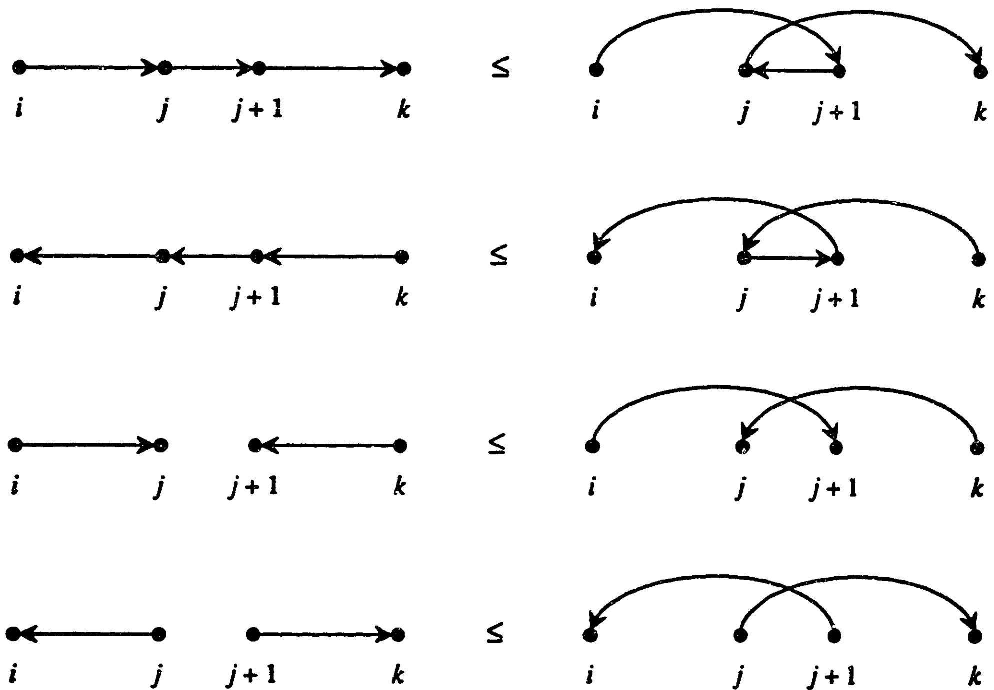

Figure 6.1: The Demidenko conditions require that if $1 \leq i < j$ and $j+1 < k \leq n$, then in each of the four "comparisons" depicted above, the total cost of the arcs on the left is at most the total cost of the corresponding arcs on the right.

Proof Consider any i, j, and k such that $1 \leq i < j$ and $j+1 < k \leq n$. The third and fourth Demidenko conditions follow immediately from the definition of a Monge array. As for the first Demidenko condition, A's Mongeness implies

$$a[j,j] + a[j+1,j+1] \leq a[j,j+1] + a[j+1,j],$$

$$a[i,j] + a[j,j+1] \leq a[i,j+1] + a[j,j],$$

and

$$a[j,j+1] + a[j+1,k] \leq a[j,k] + a[j+1,j+1].$$

Summing these three inequalities and canceling yields the first Demidenko condition:

$$a[i,j] + a[j,j+1] + a[j+1,k] \leq a[i,j+1] + a[j+1,j] + a[j,k].$$

The second Demidenko condition follows in a similar fashion. ∎

To explain why the Demidenko conditions are relevant to the traveling-salesman problem, we first need to introduce the notion of a *pyramidal* traveling-salesman tour. A traveling-salesman tour T of the graph G is said to be pyramidal if (1) the vertices on the path T follows from vertex n to vertex 1 have monotonically decreasing labels, and (2) the vertices on the path T follows from vertex 1 to vertex n have monotonically increasing labels. For example, if G has fives vertices labeled 1 through 5, then the tours $5 \to 2 \to 1 \to 3 \to 4 \to 5$ and $5 \to 4 \to 3 \to 2 \to 1 \to 5$ are pyramidal, but the tour $5 \to 2 \to 3 \to 1 \to 4 \to 5$ is not.

Pyramidal tours are interesting because a minimum-cost pyramidal tour through G can always be computed in $O(n^2)$ time using dynamic programming. Gilmore, Lawler, and Shmoys obtain this result as follows. For $1 \le i \le n$, $1 \le j \le n$, and $i \ne j$, let $E(i,j)$ denote the cost of a minimum-cost pyramidal path from vertex i to vertex j that passes through each vertex in $\{1, \ldots, \max\{i,j\}\}$ exactly once. (A pyramidal path, by analogy with a pyramidal tour, is a path P that can be decomposed into two subpaths P_1 and P_2 such that (1) the vertices on P_1 have monotonically decreasing labels, and (2) the vertices on P_2 have monotonically increasing labels.) Clearly, $E(1,2) = c[1,2]$, $E(2,1) = c[2,1]$, and the cost of a minimum-cost pyramidal tour is

$$\min\{\, E(n-1,n) + c[n,n-1],\ E(n,n-1) + c[n-1,n] \,\}\ .$$

Furthermore, it is not difficult to see that for $i \ne j$ and $\max\{i,j\} > 2$,

$$E(i,j) = \begin{cases} E(i,j-1) + c[j-1,j] & \text{if } i < j-1, \\ \min_{1 \le k < i}\{E(i,k) + c[k,j]\} & \text{if } i = j-1, \\ \min_{1 \le k < j}\{E(k,j) + c[i,k]\} & \text{if } i = j+1, \\ E(i-1,j) + c[i,i-1] & \text{if } i > j+1. \end{cases}$$

This recurrence can be used to compute all the $E(i,j)$ (and hence the cost of a minimum-cost pyramidal tour) in $O(n^2)$ time; moreover, a minimum-cost tour (and not just its cost) is easily extracted from this computation.

Minimum-cost pyramidal tours and the Demidenko conditions are related by the following

theorem, which Gilmore, Lawler, and Shmoys attribute to Demidenko.

Theorem 6.2 (Demidenko) Let C denote an $n \times n$ cost array, and let G denote the n-vertex complete directed graph corresponding to C. If $C \in \Delta$, then some minimum-cost traveling-salesman tour through G is pyramidal. ■

This theorem, together with the aforementioned dynamic-programming algorithm for computing a minimum-cost pyramidal tour, gives an $O(n^2)$-time algorithm for any instance of the traveling-salesman problem whose cost array C is a member of Δ.

With this background behind us, we can now describe this section's Monge-array result. We will identify another set of cost arrays, denoted Γ, for which minimum-cost pyramidal tours can be computed quite quickly. Specifically, for any $n \times n$ cost array C in Γ, the running time of the dynamic-programming algorithm for computing a minimum-cost pyramidal tour through the n-vertex graph G corresponding to C can be reduced from $O(n^2)$ to $O(n)$. We obtain this speedup using the on-line array-searching techniques of Section 2.2. An immediate consequence of this pyramidal-tour result is an $O(n)$-time algorithm for any instance of the n-vertex traveling-salesman problem whose cost array C is a member of $\Lambda = \Gamma \cap \Delta$.

The set Γ consists of all cost arrays $C = \{c[i,j]\}$ satisfying the following condition: if $1 \le i < n$, $1 \le j < n$, and either $i \le j - 3$ or $i \ge j + 3$, then

$$c[i,j] + c[i+1,j+1] \le c[i,j+1] + c[i+1,j]$$

Note that Γ is a superset of the set of all square Monge arrays, since an $n \times n$ Monge array C satisfies the above inequality for *all* i and j satisfying $1 \le i < n$ and $1 \le j < n$, including those i and j such that $j - 2 \le i \le j + 2$.

Our algorithm for computing a minimum-cost pyramidal tour is based on a slight variation of Gilmore, Lawler, and Shmoys' dynamic-programming formulation for the problem. For $1 \le j < n$, let $F(j)$ denote the cost of a minimum-cost pyramidal path from vertex j to vertex $j+1$ that passes through each vertex in $\{1, \ldots, j+1\}$ exactly once. (In terms of Gilmore, Lawler, and Shmoys' notation, $F(j) = E(j, j+1)$.) Similarly, for $1 \le j < n$, let $G(j)$ denote the cost of a minimum-cost pyramidal path from vertex $j+1$ to vertex j that again passes through each vertex in $\{1, \ldots, j+1\}$ exactly once. (In terms of Gilmore, Lawler, and Shmoys' notation,

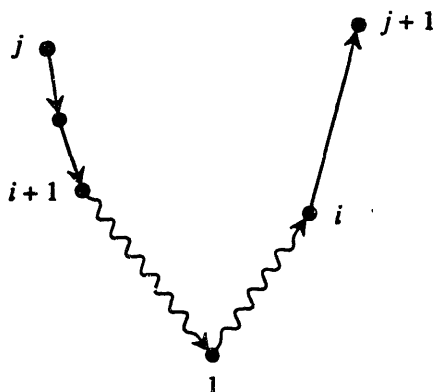

Figure 6.2: The shortest pyramidal path from vertex j to vertex $j+1$ passing through vertices $1, \ldots, j+1$ can be decomposed into three parts: (1) an edge $(i, j+1)$ such that $1 \leq i < j$, (2) a path from vertex j to vertex $i+1$ passing through vertices $i+1, \ldots, j$ *in strictly descending order*, and (3) the shortest pyramidal path from vertex $i+1$ to vertex i passing through vertices $1, \ldots, i+1$.

$G(j) = E(j+1, j)$.) Clearly, $F(1) = c[1,2]$, $G(1) = c[2,1]$, and the cost of a minimum-cost pyramidal tour through G is

$$\min\{\, F(n-1) + c[n, n-1],\, G(n-1) + c[n-1, n]\,\} .$$

Now consider any pyramidal path P from j to $j+1$ that achieves $F(j)$, and let $(i, j+1)$ denote the last arc traversed by P. We must have $1 \leq i < j$, as suggested in Figure 6.2. Moreover, if $i+1 < j$, then, since P is pyramidal, the first $j-(i+1)$ arcs traversed by P must be $(j, j-1), (j-1, j-2), \ldots, (i+2, i+1)$. Thus,

$$F(j) = \min_{1 \leq i < j} \left\{ G(i) + c[i, j+1] + \sum_{\ell=i+1}^{j-1} c[\ell+1, \ell] \right\} .$$

By a similar argument, we must also have

$$G(j) = \min_{1 \leq i < j} \left\{ F(i) + c[j+1, i] + \sum_{\ell=i+1}^{j-1} c[\ell, \ell+1] \right\} .$$

Using this recurrence to compute $F(n-1)$ and $G(n-1)$ in the naive fashion takes $O(n^2)$ time. However, if the cost array C is a member of Γ, then we can apply the on-line array-searching techniques mentioned in the previous section. To see why these techniques are applicable,

consider the $(n-1) \times (n-1)$ array $A = \{a[i,j]\}$ where

$$a[i,j] = \begin{cases} G(i) + c[i,j+1] + \sum_{\ell=i+1}^{j-1} c[\ell+1,\ell] & \text{if } i < j, \\ +\infty & \text{if } i \geq j, \end{cases}$$

and the $(n-1) \times (n-1)$ array $B = \{b[i,j]\}$ where

$$b[i,j] = \begin{cases} F(i) + c[j+1,i] + \sum_{\ell=i+1}^{j-1} c[\ell,\ell+1] & \text{if } i < j, \\ +\infty & \text{if } i \geq j. \end{cases}$$

Clearly,

$$F(j) = \min_{1 \leq i \leq m} a[i,j] ,$$

i.e., $F(j)$ is the jth column minimum of A, and

$$G(j) = \min_{1 \leq i \leq m} b[i,j] ,$$

i.e., $G(j)$ is the jth column minimum of B. Moreover, $C \in \Gamma$ implies both A and B are Monge, as the following lemma shows. (In fact, both A and B are Monge if and only if $C \in \Gamma$.)

Lemma 6.3 If C is a member of Γ, then both A and B are Monge.

Proof To show that A is Monge, first let $C_1 = \{c_1[i,j]\}$ denote the $(n-1) \times (n-1)$ array where

$$c_1[i,j] = \begin{cases} c[i,j+1] & \text{if } i < j, \\ +\infty & \text{if } i \geq j. \end{cases}$$

This array is Monge. To see why, consider any i in the range $1 \leq i < n-1$ and any j in the range $1 \leq j < n-1$. If $i+1 \geq j$, then $c_1[i+1,j] = +\infty$, which implies

$$c_1[i,j] + c_1[i+1,j+1] \leq c_1[i,j+1] + c_1[i+1,j] .$$

On the other hand, if $i+1<j$, then since C is a member of Γ and since $i \leq j-2$,

$$\begin{aligned} c_1[i,j]+c_1[i+1,j+1] &= c[i,j+1]+c[i+1,j+2] \\ &\leq c[i,j+2]+c[i+1,j+1] \\ &= c_1[i,j+1]+c_1[i+1,j]\,. \end{aligned}$$

Next, consider the $(n-1)\times(n-1)$ array $A'=\{a'[i,j]\}$ where

$$\begin{aligned} a'[i,j] &= G(i)+c_1[i,j]+\sum_{\ell=1}^{j-1}c[\ell+1,\ell]-\sum_{\ell=1}^{i}c[\ell+1,\ell] \\ &= \left[G(i)-\sum_{\ell=1}^{i}c[\ell+1,\ell]\right]+\left[\sum_{\ell=1}^{j-1}c[\ell+1,\ell]\right]+[c_1[i,j]]\,. \end{aligned}$$

Since

$$a[i,j]=\begin{cases} a'[i,j] & \text{if } i<j, \\ +\infty & \text{if } i\geq j, \end{cases}$$

every 2×2 subarray of A is either a 2×2 subarray of A' or its left- and bottommost entry is a $+\infty$; thus, if we can show that A' is Monge, then A must also be Monge.

To show that A' is Monge, note that the term $G(i)-\sum_{\ell=1}^{i}c[\ell+1,\ell]$ in the definition of A' depends only on i, and the term $\sum_{\ell=1}^{j-1}c[\ell+1,\ell]$ depends only on j. Furthermore, as we showed earlier, C_1 is Monge. Thus, by Properties 1.5 and 1.6, A' is Monge.

A similar argument shows that $C\in\Gamma$ also implies B are Monge. ■

Now suppose we precompute

$$\sum_{\ell=1}^{j-1}c[\ell+1,\ell]$$

and

$$\sum_{\ell=1}^{j-1}c[\ell,\ell+1]$$

for all j in the range $2\leq j<n$. This preprocessing requires $O(n)$ time. Moreover, it allows any entry $a[i,j]$ of A to be computed in constant time from $G(i)$, the ith column minimum of B, and any entry $b[i,j]$ of B to be computed in constant time from $F(i)$, the ith column minimum

of A. Thus, by interleaving the computation of A's column minima with the computation of B's column minima, as discussed in Section 2.2, we can use the LIEBER algorithm algorithm to compute $F(2), \ldots, F(n-1)$ and $G(2), \ldots, G(n-1)$ in $O(n)$ time.

Since a minimum-cost pyramidal tour (and not just its cost) is easily extracted from the computation of $F(2), \ldots, F(n-1)$ and $G(2), \ldots, G(n-1)$, we have the following theorem and corollary.

Theorem 6.4 Let C denote an $n \times n$ cost array, and let G denote the n-vertex complete directed graph corresponding to C. If $C \in \Gamma$, then a minimum-cost pyramidal tour through G can be computed in $O(n)$ time. ■

Corollary 6.5 Let C denote an $n \times n$ cost array, and let G denote the n-vertex complete directed graph corresponding to C. If $C \in \Gamma \cap \Delta$, then a minimum-cost traveling-salesman tour through G can be computed in $O(n)$ time. ■

We conclude this section by noting that the proof of Theorem 6.2 given in [GLS85] is, as the authors admit, rather long and tedious. However, a weaker Monge-array version of theorem is quite easy to prove. Specifically, Gilmore, Lawler, and Shmoys give a simple one-paragraph argument in [GLS85] showing that if C is an $n \times n$ Monge array, then some minimum-cost traveling-salesman tour through the n-vertex complete directed graph corresponding to C is pyramidal. So why mention Demidenko's stronger but harder-to-prove result in this section? We mention it because the intersection $\Lambda = \Gamma \cap \Delta$ contains potentially interesting nonMonge cost arrays. For example, consider a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order. Corresponding to P is a complete directed graph G on P's vertices with cost array $C = \{c[i,j]\}$ where $c[i,j]$ is the Euclidean distance between vertices v_i and v_j of P. It is not hard to verify that C is a nonMonge member of Λ. Thus, the traveling-salesman problem corresponding to P can be solved in $O(n)$ time. However, for cost arrays of this form, an $O(n)$-time algorithm is not particularly impressive, as $1 \to 2 \to \cdots \to n \to 1$ (i.e., the tour traversing the perimeter of P in clockwise order) is always a minimum-cost traveling-salesman tour for G.

6.2 Yao's Dynamic-Programming Problem

In [Yao80], Yao developed some very general techniques for speeding up dynamic programming. (These techniques are also discussed in [Yao82].) She considered dynamic-programming recurrences of the following form: for $1 \le i \le j \le n$,

$$E(i,j) = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) + \min_{k \text{ s.t. } i \le k < j} \{E(i,k) + E(k+1,j)\} & \text{if } i < j. \end{cases} \quad (6.1)$$

In this recurrence, we assume the interval function $w(i,j)$ can be evaluated in constant time for all i and j satisfying $1 \le i < j \le n$, and we want to compute $E(i,j)$ for all i and j satisfying $1 \le i < j \le n$. (In order to be consistent with notation used in the next chapter, our notation differs slightly from the notation used in [Yao80] and [AP89b].)

The $E(i,j)$ given by the above recurrence are easily computed in $O(n^3)$ time. We merely compute $E(i,j)$ for those i and j such that $j-i=1$, then for those i and j such that $j-i=2$, then for those i and j such that $j-i=3$, and so on, until we finally obtain $E(1,n)$. However, in [Yao80], Yao identified a special case of this dynamic programming problem that can be solved in significantly less time. (We call this special case *Yao's problem* in her honor.) Specifically, she showed that if the interval function $w(\cdot,\cdot)$ satisfies the quadrangle inequality (as defined in Section 1.3) *and* it is monotonically increasing on the lattice of intervals (i.e., $w(i',j') \le w(i,j)$ if $[i',j'] \subseteq [i,j]$), then all the $E(i,j)$ can be computed in $O(n^2)$ time.

In this section, we reformulate Yao's dynamic programming problem and her algorithm for solving the problem in terms of multidimensional Monge arrays. We also use the on-line array-searching algorithm of Section 2.2. to obtain an alternate $O(n^2)$-time solution for the problem.

6.2.1 Optimal Binary Search Trees

In this subsection, we present an example of Yao's problem, the optimal-binary-search-tree problem. (Yao gives several more examples in [Yao80].) Before we can describe this problem, however, we need a few definitions.

A binary search tree T is a special kind of labeled binary tree. Associated with each node x

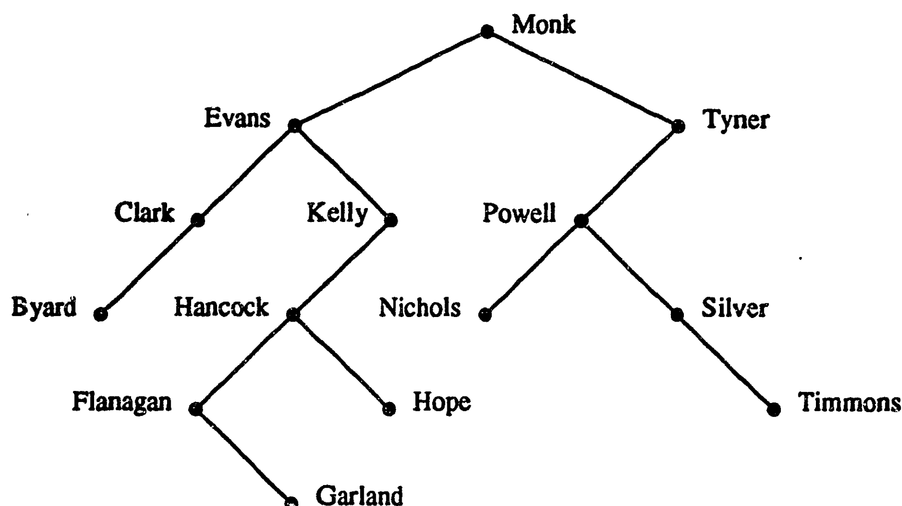

Figure 6.3: An example of a binary search tree. Locating "Sonny Clark" requires three comparisons, while determining that the tree contains neither "Carl Perkins" nor "George Wallington" requires four and two comparisons, respectively.

in T is a unique key $a[x]$, such that for any node x in T, any node y in the left subtree of x, and any node z in the right subtree of x, $a[y] < a[x] < a[z]$. Figure 6.3 gives an example of binary tree whose nodes are labeled with the names of modern jazz pianists, ordered lexicographically. Note that there are many different binary search trees associated with any particular set of n keys.

The binary search tree is a very useful data structure. Given a binary search tree T and a value A, we can locate a node x in T such that $A = a[x]$ (or determine that no such x exists) using a simple binary search starting at the T's root. If $A = a[x]$ for some node x, then this search requires $1 + \text{depth}(x)$ comparisons, where $\text{depth}(x)$ denotes the depth of x in T.

Given probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$, the *optimal-binary-search-tree* problem is that of constructing a minimum-cost n-node binary search tree T for keys $a_1, \ldots, a_n$ such that $a_1 < a_2 < \cdots < a_n$. The cost of T is the expected number of comparisons required to locate some random value A, where

$$\Pr\{A = a_i\} = p_i$$

and

$$\Pr\{a_i < A < a_{i+1}\} = q_i .$$

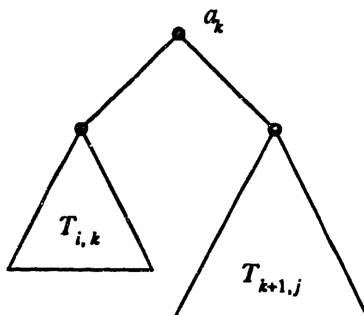

Figure 6.4: If $T_{i,j}$ is an optimal binary search tree for keys $a_i, \ldots, a_{j-1}$ and a_k is the key associated with $T_{i,j}$'s root, where $i \leq k < j$, then $T_{i,j}$'s left subtree $T_{i,k}$ is an optimal binary search tree for keys $a_i, \ldots, a_{k-1}$, and $T_{i,j}$'s right subtree $T_{k+1,j}$ is an optimal binary search tree for keys $a_i, \ldots, a_{k-1}$.

(By convention, $a_0 = -\infty$ and $a_{n+1} = +\infty$.) Equivalently, the cost of T is the tree's *weighted path length.*

For $1 \leq i < j \leq n+1$, let $E(i,j)$ denote the cost of a minimum-cost binary-search tree for keys $a_i, \ldots, a_{j-1}$. Furthermore, let $T_{i,j}$ denote any binary-search tree achieving $E(i,j)$ and let a_k denote the key associated with $T_{i,j}$'s root. As suggested in Figure 6.4, $T_{i,j}$'s left and right subtrees — denoted $T_{i,k}$ and $T_{k+1,j}$, respectively — must be minimum-cost binary search trees for keys $a_i, \ldots, a_{k-1}$ and $a_{k+1}, \ldots, a_{j-1}$, respectively, with costs $E(i,j)$ and $E(k+1,j)$, respectively. Thus, we must have

$$\begin{aligned} E(i,j) &= p_k \\ &\quad + E(i,k) + q_i + p_i + q_{i+1} + \cdots + q_{k-1} + p_{k-1} + q_k \\ &\quad + E(k+1,j) + q_{k+1} + p_{k+1} + q_{k+2} + \cdots + q_{j-1} + p_{j-1} + q_j \\ &= \left(\sum_{\ell=i}^{j-1} p_\ell\right) + \left(\sum_{\ell=i}^{j} q_\ell\right) + E(i,k) + E(k+1,j)\,. \end{aligned}$$

Setting

$$w(i,j) = \sum_{\ell=i}^{j-1} p_\ell + \sum_{\ell=i}^{j} q_\ell\,,$$

we then obtain (6.1). Moreover, it is not difficult to verify that this weight function satisfies the quadrangle inequality and that it is monotonically increasing on the lattice of intervals.

6.2.2 Yao's Algorithm

In this subsection, we reformulate Yao's $O(n^2)$-time algorithm for solving (6.1) in terms of the Monge arrays defined above. (Mention Knuth [Knu71].)

Thus, Yao's general algorithm solves the optimal-binary-search-tree problem in $O(n^2)$ time; this result matches the best previous result for the problem, an $O(n^2)$-time algorithm due to Knuth [Knu71].

We begin with a lemma proved by Yao in obtaining her $O(n^2)$-time bound.

Lemma 6.6 (Yao [Yao80]) If the interval function $w(\cdot,\cdot)$ both satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals, then the interval function $E(\cdot,\cdot)$ satisfies the quadrangle inequality. ■

Corollary 6.7 If the weight function $w(\cdot,\cdot)$ both satisfies the inverse quadrangle inequality and is monotonically decreasing on the lattice of intervals (i.e., $w(i',j') \geq w(i,j)$ if $[i',j'] \subseteq [i,j]$), then the interval function $E(\cdot,\cdot)$ satisfies the inverse quadrangle inequality. ■

Yao's quadrangle inequality is precisely the Monge condition, except that the functions $w(i,j)$ and $E(i,j)$ do not correspond to complete arrays. However, if we let $W = \{w[i,j]\}$ denote the $n \times n$ array where

$$w[i,j] = \begin{cases} w(i,j) & \text{if } i \leq j, \\ \infty & \text{otherwise,} \end{cases}$$

and we let $E = \{e[i,j]\}$ denote the $n \times n$ array where

$$e[i,j] = \begin{cases} E(i,j) & \text{if } i \leq j, \\ \infty & \text{otherwise,} \end{cases}$$

then both W and E satisfy the Monge condition. (Note that it is important that we define $w[i,j]$ and $e[i,j]$ to be ∞ when $i > j$ — if we instead define $w[i,j]$ and $e[i,j]$ to be $-\infty$ when $i > j$, as we might want to do if we were maximizing instead of minimizing, then W and E would not satisfy the Monge condition or the inverse Monge condition.) Yao's dynamic

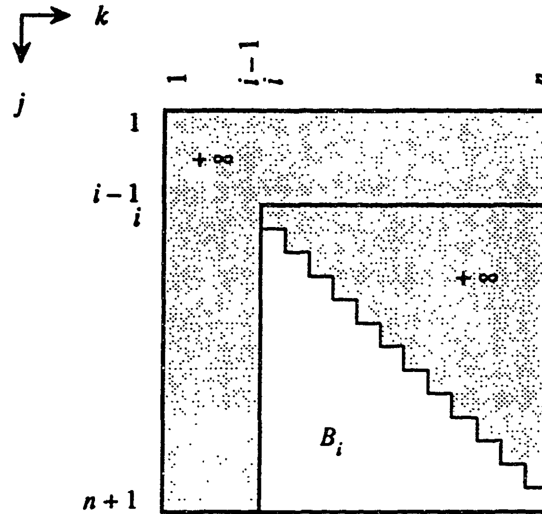

Figure 6.5: For any i in the range $1 \leq i \leq n$, $E(i, i+1)$ through $E(i, n+1)$ are the minimum entries in rows $i+1$ through $n+1$ of the plane A_i, which correspond to the minimum entries in rows 2 through $n-i+2$ of the subarray B_i.

programming problem then boils down to computing the entries of E. Unfortunately, the results of [AKM+87] seems inapplicable at this point, at least in a straightforward manner, as we are neither interested in the row minima of E nor are the entries of E readily available.

Let $A = \{a[i,j,k]\}$ denote the $n \times (n+1) \times n$ array where $a[i,j,k] = w[i,j] + e[i,k] + e[k+1,j]$. Furthermore, for $1 \leq i \leq n$, let $A_i = \{a_i[j,k]\}$ denote the $(n+1) \times n$ two-dimensional plane of A corresponding to those entries whose first coordinate is i, and let $B_i = \{b_i[s,t]\}$ denote the $(n-i+2) \times (n-i+1)$ subarray of A_i consisting of rows i through $n+1$ and columns i through n of A_i, so that $b_i[s,t] = a_i[s+i-1, t+i-1]$. (One such plane A_i and its subarray B_i are depicted in Figure 6.5.) Then for $1 \leq i < j \leq n$,

$$\begin{aligned} E(i,j) &= \min_{k \text{ s.t. } 1 \leq k \leq n} a[i,j,k] \\ &= \min_{t \text{ s.t. } 1 \leq t \leq n-i+1} b_i[j-i+1, t] , \end{aligned}$$

i.e., $E(i, i+1), \ldots, E(i, n+1)$ are simply row minima of B_i. (To be precise, they are the minimum entries in rows 2 through $n-i+2$ of B_i.) Moreover, since W and E are Monge arrays, A is a cycle-decomposable Monge-composite array, which implies both A_i and B_i are Monge for all i in the range $1 \leq i \leq n$.

Yao's algorithm consists of n stages, each requiring $O(n)$ time. In the ℓth stage, we locate the minimum entry in tube (i,j) of A for all i and j such that $1 \le i < j \le n+1$ and $j - i = \ell$. In other words, we compute $E(1,\ell+1), E(2,\ell+2), \ldots, E(n+1-\ell, n+1)$.

The first stage is easily completed in $O(n)$ time, since $e[i,i+1] = w[i,i+1]$ for all i in the range $1 \le i \le n$. For $\ell \ge 2$, we compute those $e[i,j]$ on the ℓth diagonal of E as follows. Since A is Monge, Property 1.12 implies

$$k(i,i+\ell-1) \;\le\; k(i,i+\ell) \;\le\; k(i+1,i+\ell)$$

for all i in the range $1 \le i \le n+1-\ell$. Furthermore, $k(i+1,i+\ell)$ and $k(i,i+\ell-1)$ are known from the previous stage. Thus, for $1 \le i \le n+1-\ell$, we can compute $k(i,i+\ell)$ in $O(k(i+1,i+\ell) - k(i,i+\ell-1) + 1)$ time, which implies the ℓth stage takes

$$\begin{aligned}\sum_{i=1}^{n+1-\ell} O\left(k(i+1,i+\ell) - k(i,i+\ell-1) + 1\right) &= O\left(k(n+2-\ell,n+1) - k(1,\ell) + n + 1 - \ell\right) \\ &= O(n)\end{aligned}$$

total time.

6.2.3 An Alternate Quadratic-Time Algorithm

Our algorithm consists of n stages. In the ith stage, we compute the row minima of B_i, which gives us $E(i,j)$ for all j such that $i < j \le n$. To compute these row minima, we first observe that for $1 \le t < s \le n - i + 2$,

$$b_i[s,t] \;=\; w[i,s+i-1] + e[i,t+i-1] + e[t+i,s+i-1]$$

can be computed in constant time from $e[i,t+i-1]$ and $e[t+i,s+i-1]$. Since $t \ge 1$, we have already computed $e[t+i,s+i-1] = E(t+i,s+i-1)$. Moreover, $e[i,t+i-1] = E(i,t+i-1)$ is the minimum entry in row t of B_i. Thus, we can compute the row minima of B_i (and hence $E(i,i+1),\ldots,E(i,n+1)$) in $O(n-i)$ time using the on-line LIEBER algorithm of Section 2.2.

Since the total running time of the above algorithm is $O(n^2)$ time, we have the following theorem.

Theorem 6.8 If the weight function $w(\cdot,\cdot)$ both satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals, then Yao's dynamic programming problem can be solved in $O(n^2)$ time. ∎

As a final observation, suppose the weight function $w(\cdot,\cdot)$ satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals, as before, but we are interested in maximizing $E(i,j)$ rather than minimizing it. In other words, for $1 \le i \le j \le n$,

$$E(i,j) = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) + \max\limits_{k \text{ s.t. } i \le k < j} \{E(i,k) + E(k+1,j)\} & \text{if } i < j. \end{cases}$$

For example, we might want to construct a binary search tree that *maximizes* the expected number of comparisons performed by a find operation. Since for $i < j$,

$$\begin{aligned} -E(i,j) &= -\left(w(i,j) + \max_{k \text{ s.t. } i \le k < j} \{E(i,k) + E(k+1,j)\}\right) \\ &= (-w(i,j)) + \min_{k \text{ s.t. } i \le k < j} \{(-E(i,k)) + (-E(k+1,j))\}, \end{aligned}$$

this maximizing variant of Yao's problem is equivalent to solving the original minimizing recurrence when $-w(\cdot,\cdot)$ satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals, or, equivalently, when $w(\cdot,\cdot)$ satisfies the inverse quadrangle inequality and is monotonically decreasing on the lattice of intervals.

By combining the approach of Subsection 6.2.3 with Klawe and Kleitman's on-line algorithm for computing the row minima of a partial inverse-Monge array of the staircase variety, we can obtain an $O(n^2\alpha(n))$-time algorithm for the maximizing variant of Yao's problem.

Theorem 6.9 If the weight function $w(\cdot,\cdot)$ both satisfies the inverse quadrangle inequality and is monotonically decreasing on the lattice of intervals, then Yao's dynamic programming problem can be solved in $O(n^2\alpha(n))$ time.

Proof This proof is identical to that given above for Theorem 6.8 except that the arrays W, E, and B_i defined above are no longer Monge arrays; instead, they are partial inverse-Monge arrays. For $1 \le i \le n$, the row-minima problem for B_i is now equivalent to the *convex* least-weight subsequence problem considered in [EGG88]. As we mentioned in Section 2.4, Klawe

and Kleitman [KK90] have shown that this problem can be solved in $O(n\alpha(n))$ time. Thus, by applying Klawe and Kleitman's algorithm n times, we can obtain the entries of E in $O(n^2\alpha(n))$ time. ■

It remains open whether the time complexity given in Theorem 6.8 or that given in Theorem 6.9 can be improved.

Chapter 7

Dynamic Programming and Economic Lot Sizing

This chapter presents efficient algorithms for problems related to economic lot-size models. These algorithms use Monge-array techniques to speed up classical dynamic-programming algorithms for production scheduling. The results covered in this chapter represent joint work with Aggarwal that was first described in [AP91].

Economic lot-size models typically deal with production and/or inventory systems. A product (which could be a raw material, a purchased part, or a semifinished or finished product in manufacturing or retailing) is produced or purchased in batch quantities and placed in stock. As the stock is depleted by demands for the product, more of the product must be produced or purchased. The object of production planning is to minimize the cost of this cycle of filling and depleting the stock. Since the number of variables affecting production planning is usually quite large (for example, these variables may include work-force levels, physical resources of the firm, and external variables such as federal regulations), economic lot-size models typically make certain simplifying assumptions. Some researchers have studied models with the assumption that the demands on the inventory follow a given probabilistic distribution, while others have assumed that these demands are *deterministic and known in advance.* In this chapter, we study models based on the latter assumption.

The study of economic lot-size models that assume deterministic demands dates to at least

1915 with F. W. Harris [Har15], who considered a model that assumes demands occur continuously over time. About three decades ago, a different approach was independently provided by Manne [Man58] and by Wagner and Whitin [WW58]; they divided time into discrete *periods* and assumed that the demand in each period is known in advance. Since 1958, the Manne-Wagner-Whitin model has received considerable attention, and several hundred papers have directly or indirectly discussed this model; most of these papers have either extended this model or provided efficient algorithms for production problems that arise in it. (Indeed, Lee and Denardo [LD86] have provided convincing reasons why the Manne-Wagner-Whitin model is a reasonable one.) The references given here and those given in [BRG87] provide only some of the papers related to the Manne-Wagner-Whitin model. Today, even an introductory course in operations research for managers and economists is likely to include a chapter on the Manne-Wagner-Whitin model and on some of its extensions. (See, for example, the following books: [Den82, HC84, JM74, Wag75].) Because of the immense interest in economic lot-size models, a considerable amount of research effort has been focussed on establishing the computational complexity of various problems in these models. (In particular, see Florian, Lenstra, and Rinnooy Kan [FLR80], Bitran and Yanasse [BY82], Luss [Lus82], Erickson, Monma, and Veinott [EMV87], and Chung and Lin [CL88].)

This chapter reviews the Manne-Wagner-Whitin model and some of its extensions. It also provides efficient algorithms for several production planning problems expressed in terms of this model, all of which assume concave costs. We focus on uncapacitated economic lot-size problems, i.e., problems without bounds on production, inventory, or backlogging; similar results for capacitated problems, as well as related problems involving negative demands and shelf-life bounds, are given in [AP90].

Our algorithms use dynamic programming [Bel57] and the on-line array-searching techniques described in Chapter 2, and they typically improve the running times of previous algorithms by factors of n and $n/\lg n$, where n is the number of time periods under consideration; these improvements are listed in Tables 7.1, 7.2, and 7.3. In many cases, the running times of these algorithms are optimal to within a constant factor or to within a factor of $\lg n$.

One of the critical contributions of this chapter is our identification of the Monge arrays that arise in connection with the economic lot-size model; it is these arrays that allow us to apply

the techniques of Chapter 2 and improve the time bounds of previous algorithms for economic lot-size problems so dramatically. We also raise several unresolved questions regarding the time complexities of various problems formulated in terms of the economic lot-size model. It is our hope that these open questions will stimulate interest in the economic lot-size model among researchers in theoretical computer science and related areas.

Recently, two groups of researchers from the operations research community — Federgruen and Tzur [FT89, FT90] and Wagelmans, van Hoesel, and Kolen [WvHK89] — have independently obtained some of the results presented in this chapter using different techniques. We will briefly describe their work and contrast it with our own in the final section of this chapter.

The remainder of this chapter is organized as follows. In Section 7.1, we review the Manne-Wagner-Whitin model and list the main results of this chapter. In Section 7.2, we discuss the dynamic programming techniques developed by previous researchers for solving economic lot-size problems with concave costs, and then in the Sections 7.3–7.5, we combine these techniques with the array-searching techniques of Chapter 2 to obtain algorithms for three different types of economic lot-size problems. Finally, in Section 7.6 we discuss several extensions to our work, relate our results to the aforementioned work of Federgruen and Tzur [FT89, FT90] and of Wagelmans, van Hoesel, and Kolen [WvHK89], and present some open problems.

7.1 Background and Definitions

7.1.1 The Basic Model

To describe the basic model given in [Man58, WW58], we use the notation employed by Denardo in [Den82]. Demand for the product in question occurs during each of n consecutive time *periods* (i.e., intervals of time) numbered 1 through n. The demand that occurs during a given period can be satisfied by production during that period or during any earlier period, as inventory is carried forward in time. (This basic model differs from the backlogging model described in Subsection 7.1.2 in that demand is not allowed to accumulate and be satisfied by future production.) Without loss of generality, we assume both the initial inventory (at the beginning of first period) and the final inventory (at the end of period n) are zero. The model includes production costs and inventory carrying costs, and the objective is to schedule production so

as to satisfy demand at minimum total cost.

The data in this model are the demands, the production cost functions, and the inventory-carrying cost functions. In particular, for $1 \le i \le n$,

$$\begin{aligned} d_i &= \text{the demand during period } i, \\ c_i(x) &= \text{the cost of producing } x \text{ units during period } i, \text{ and} \\ h_i(y) &= \text{the cost of storing } y \text{ units of inventory from period } i-1 \text{ to period } i, \end{aligned}$$

where for the duration of this chapter, we assume $d_i \ge 0$ for all i in the range $1 \le i \le n$. Furthermore, the model has $2n+1$ decision variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_{n+1}$, where for $1 \le i \le n$,

$$x_i = \text{the production during period } i,$$

and for $1 \le i \le n+1$,

$$y_i = \text{the inventory stored from period } i-1 \text{ to period } i.$$

Demand, production, and inventory occur in real quantities, and the problem of meeting demand at minimal total cost has the following mathematical representation:

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} \{c_i(x_i) + h_i(y_i)\} & \\ \text{subject to the constraints} & y_1 = y_{n+1} = 0 & \\ & x_i \ge 0 & \text{for } 1 \le i \le n, \\ & y_i \ge 0 & \text{for } 1 \le i \le n, \text{ and} \\ & y_i + x_i = d_i + y_{i+1} & \text{for } 1 \le i \le n. \end{array} \tag{7.1}$$

The first constraint of (7.1) assures that the initial and final inventories are zero, while the second and third constraints limit production and inventory to nonnegative values. (Requiring inventory to be nonnegative insures that the demand in period i is satisfied by production

during that period or during earlier periods.) Finally, matter must be conserved, so the fourth constraint requires that the sum of the inventory at the start of a period and the production during that period equals the sum of the demand during that period and the inventory at the start of the next period.

The production and inventory levels are, of course, interrelated. If one knew the inventory levels $y_1, \ldots, y_n$ at the beginning of all periods, one could determine the production levels $x_1, \ldots, x_n$ from the conservation-of-matter constraint. Conversely, if one knew the production levels $x_1, \ldots, x_n$, one could determine the inventory levels $y_1, \ldots, y_n$ from the equation

$$y_i = (d_1 + \cdots + d_{i-1}) - (x_1 + \cdots + x_{i-1}) . \quad (7.2)$$

To interpret (7.2), note that the inventory y_i at the beginning of period i equals the total production during periods 1 through $i-1$ less the total demand during these periods.

The production levels $x_1, \ldots, x_n$ give a *production plan* or *production schedule.* We will say that a particular schedule is *feasible* if it and the inventory levels determined by (7.2) satisfy the constraints of (7.1). Moreover, we will say that a particular schedule is *optimal* if it is a feasible production schedule that minimizes $\sum_{i=1}^{n} \{c_i(x_i) + h_i(y_i)\}$ over all feasible production schedules.

The basic economic lot-size problem can also be formulated as a network-flow problem. (This formulation was first proposed by Zangwill in [Zan68].) Consider the directed graph depicted in Figure 7.1. This graph consists of a single source, capable of generating a net outflow of $\sum_{i=1}^{n} d_i$, and n sinks, such that the ith sink requires d_i units of net inflow. Furthermore, for $1 \le i \le n$, there is an arc from the source to ith sink with associated cost function $c_i(\cdot)$, and for $2 \le i \le n$, there is an arc from the $(i-1)$st sink to the ith sink with associated cost function $h_i(\cdot)$. A minimum-cost flow for this graph corresponds to an optimal production schedule for the associated economic lot-size problem.

If $c_i(\cdot)$ and $h_i(\cdot)$ are arbitrary functions, then the basic economic lot-size problem is NP-hard[1], as Florian, Lenstra, and Rinnooy Kan showed in [FLR80]. In view of this difficulty, cer-

[1]Note that if production levels are restricted to integer values, then dynamic programming does yield a weakly-polynomial algorithm for computing an optimal production schedule, even for arbitrary production and inventory cost functions. The algorithm's running time is polynomial in n and the total demand $D = \sum_{i=1}^{n} d_i$

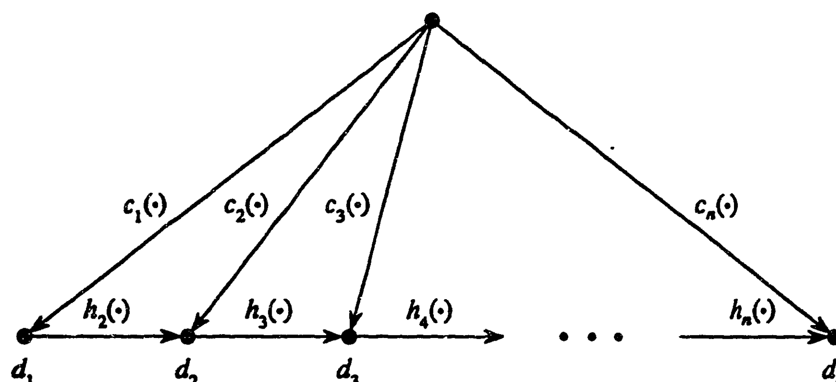

Figure 7.1: The basic economic lot-size problem can be formulated as a network-flow problem.

tain assumptions are often made regarding the structure of the basic economic lot-size model's costs; we review some of these assumptions below.

1. In their pioneering papers, Manne [Man58] and Wagner and Whitin [WW58] assumed that for $1 \leq i \leq n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

   and $h_i(y) = h_i^1 y$, where the c_i^0, c^1, and h_i^1 are all nonnegative constants[2]. (The assumption that $c^1 \geq 0$ can be dropped, as changing c^1 affects only the cost of the optimal production schedule and not its structure.) Wagner and Whitin [WW58] also provided an $O(n^2)$-time algorithm for computing an optimal production plan. Note that the *set-up costs* c_i^0 are what make this problem interesting; if $c_i^0 = 0$ for all i, then the problem can be solved trivially.

2. A function $f(\cdot)$ whose domain is the real line is called *concave* if for all real numbers x, y, and z such that $x \geq y$ and $z \geq 0$, we have

$$f(x+z) - f(x) \leq f(y+z) - f(y) . \tag{7.3}$$

but potentially exponential in the size of the input.

[2]We include subscripts on the constants $c_1^0, c_2^0, \ldots, c_n^0$ and $h_1^1, h_2^1, \ldots, h_n^1$ (but not on the constant c^1) to indicate that every period's cost functions are defined in terms of a (potentially) different pair of constants c_i^0 and h_i^1 but the same constant c^1.

Furthermore, $f(\cdot)$ is called concave on an interval I of the real line if (7.3) holds for all $x \geq y$ and for all $z \geq 0$ *such that* both y and $x+z$ are contained in the interval I. Finally, a function $g(\cdot)$ whose domain is some interval J of the real line is called concave if it is concave on its domain J. In [Wag60], Wagner showed that the $O(n^2)$-time algorithm given in [WW58] can still be used if for $1 \leq i \leq n$, $c_i(x)$ is concave (or more precisely, it is concave on $[0, +\infty)$, the relevant portion of its domain), and $h_i(y) = h_i^1 y$, where the h_i^1 are again nonnegative constants.

3. Zabel [Zab64] and Eppen, Gould, and Pashigian [EGP69] considered a somewhat simpler cost structure; for $1 \leq i \leq n$, they assumed that

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

and $h_i(y) = h_i^1 y$, where the c_i^0, c_i^1, and h_i^1 are all nonnegative constants. (The assumption that $c_i^1 \geq 0$ for $1 \leq i \leq n$ can be dropped, as changing all the c_i^1 by the same amount affects only the cost of the optimal production schedule and not its structure.) For this cost structure, both Zabel and Eppen, Gould, and Pashigian provided some additional properties of an optimal production schedule. Both papers also exploited these properties to obtain algorithms for computing an optimal schedule that run faster in practice but which still require quadratic time in the worst case.

4. In [Zan69], Zangwill again assumed that

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

for $1 \leq i \leq n$, but he allowed the $h_i(\cdot)$ to be arbitrary concave functions (on $[0, +\infty)$). For this cost structure, he showed that Wagner and Whitin's approach still yields an $O(n^2)$-time algorithm for computing an optimal production schedule. (See also Subsection 7.1.2.)

5. Finally, Veinott [Vei63] showed that even if both the $c_i(\cdot)$ and the $h_i(\cdot)$ are arbitrary concave functions, Wagner and Whitin's approach gives an $O(n^2)$-time algorithm.

cost structure	*previous result*	*new result*
$c_i(0) = 0$ $c_i(x) = c_i^0 + c_i^1 x$ for $x > 0$ $c_i^0 \geq 0$ $h_i(y) = h_i^1 y$ $c_i^1 \leq c_{i-1}^1 + h_i^1$	$O(n^2)$ [WW58] assumed $c_i^1 = c^1$	$O(n)$ Theorem 7.4
$c_i(0) = 0$ $c_i(x) = c_i^0 + c_i^1 x$ for $x > 0$ $c_i^0 \geq 0$ $h_i(y) = h_i^1 y$	$O(n^2)$ [Zab64, EGP69]	$O(n \lg n)$ Theorem 7.5
$c_i(\cdot)$ and $h_i(\cdot)$ concave	$O(n^2)$ [Vei63]	no improvement

Table 7.1: A summary of our results for the basic economic lot-size problem. The results are bounds on the time to find an optimal production schedule, where n is the number of periods.

Observe that if we interpret $f(x)$ as the cost of producing (or storing) x items, then a concave $f(\cdot)$ implies *decreasing marginal costs*, or equivalently, *economies of scale*. Since microeconomic theories often assume economies of scale, the concave cost structure assumed by Veinott seems reasonable, which is one of the reasons why the economic lot-size model with linear or concave costs has received so much attention.

In Section 7.3, we provide efficient algorithms for several of the cost structures discussed above. The time complexities of these algorithms are listed in Table 7.1. The new algorithms use dynamic programming and Monge-array techniques.

7.1.2 The Backlogging Model

Until now, we have assumed that the demand for a particular period is satisfied by production during that period or during earlier periods. In 1966, Zangwill [Zan66] extended the basic model by allowing demand to go unsatisfied during some period, provided it is satisfied eventually by production in some subsequent period. (Satisfying demand with future production is known as *backlogging* demand.) Zangwill's extension changes the formulation of the economic lot-size problem given in Subsection 7.1.1 in that it allows the variables y_2 through y_n in (7.1) to be negative. Equation (7.2) still identifies y_i as the total production during periods 1 through $i-1$

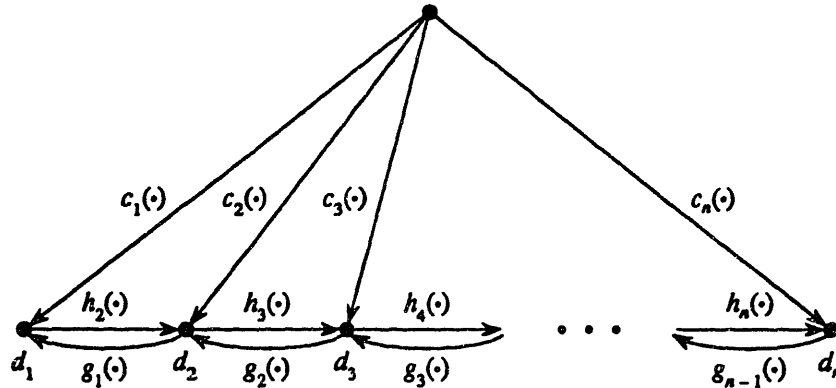

Figure 7.2: The backlogging economic lot-size problem can be formulated as a network-flow problem.

less the total demand during those periods; however, when y_i is negative, it now represents a ***shortage*** of $-y_i$ units of unfulfilled (backlogged) demand that must be satisfied during periods i through n. Furthermore, when y_i is nonnegative, $h_i(y_i)$ remains equal to the cost of y_i units of inventory at the start of period i, but when y_i is negative, $h_i(y_i)$ becomes the cost of having a shortage of y_i units at the start of period i. For the sake of clarity, we let $g_{i-1}(-y_i) = h_i(y_i)$ in this latter case.

The backlogging economic lot-size problem, like the basic problem, can also be formulated as a network-flow problem. We use the same single-source, n-sink directed graph as for the basic economic lot-size problem, except that for $2 \leq i \leq n$, we add an arc from the ith sink to the $(i-1)$st sink with associated cost function $g_{i-1}(\cdot)$. This new graph is depicted in Figure 7.2. Again, a minimum-cost flow for this graph corresponds to an optimal production schedule for the associated economic lot-size problem with backlogging.

As is the case for the basic economic lot-size problem (given in Subsection 7.1.1), the backlogging economic lot-size problem is NP-hard if arbitrary cost functions are allowed. For this reason, researchers have studied a fair number of restricted cost structures, some of which are listed below.

1. In [Zan66], Zangwill assumed that the $c_i(\cdot)$, $h_i(\cdot)$, and $g_i(\cdot)$ are all arbitrary concave functions (or more precisely, they are all concave on $[0, +\infty)$), and provided an $O(n^3)$-time dynamic programming algorithm for computing an optimal production plan.

2. In [Zan69], Zangwill assumed that

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

for $1 \leq i \leq n$ (where the c_i^0 and c^1 are nonnegative constants) and that the $h_i(\cdot)$ and $g_i(\cdot)$ are arbitrary concave functions. For this cost structure, he provided an $O(n^2)$-time algorithm for computing an optimal production plan.

3. Blackburn and Kunreuther [BK74] and Lundin and Morton [LM75] assumed that

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

$g_i(z) = g_i^1 z$, and $h_i(y) = h_i^1 y$, where $c_i^0, c_i^1, g_i^1, h_i^1 \geq 0$ for $1 \leq i \leq n$. For this case, they obtained some characteristics of optimal production schedules; these characteristics are generalizations of those given by Eppen et al. [EGP69] for the basic model (i.e., the one without backlogging). Both [BK74] and [LM75] also gave algorithms for determining an optimal production plan, but these algorithms again take quadratic time in the worst case. Like Eppen et al. [EGP69], however, Lundin and Morton [LM75] argued that their algorithm runs faster in practice than that of Zangwill.

4. Finally, Morton [Mor78] considered a very simple cost structure in which

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

$g_i(z) = g^1 z$, and $h_i(y) = h^1 y$, where $c_i^0, c^1, g^1, h^1 \geq 0$ for $1 \leq i \leq n$. For this case, Morton provided a very simple $O(n^2)$-time algorithm, which seems to run quite efficiently in practice.

In Section 7.4, we provide asymptotically faster algorithms for most of the cost structures discussed above. The time complexities of these algorithms are listed in Table 7.2. We again

cost structure	*previous result*	*new result*
$c_i(0) = 0$ $c_i(x) = c_i^0 + c_i^1 x$ for $x > 0$ $c_i^0 \geq 0$ $g_i(z) = g_i^1 z$ $h_i(y) = h_i^1 y$ $c_i^1 \leq c_{i+1}^1 + g_i^1$ $c_i^1 \leq c_{i-1}^1 + h_i^1$	$O(n^2)$ [Mor78] assumed $c_i^1 = c^1$ $g_i^1 = g^1$ $h_i^1 = h^1$	$O(n)$ Theorem 7.8
$c_i(0) = 0$ $c_i(x) = c_i^0 + c_i^1 x$ for $x > 0$ $c_i^0 \geq 0$ $g_i(z) = g_i^1 z$ $h_i(y) = h_i^1 y$	$O(n^2)$ [BK74, LM75]	$O(n \lg n)$ Theorem 7.9
$c_i(0) = 0$ $c_i(x) = c_i^0 + c^1 x$ for $x > 0$ $c_i^0 \geq 0$ $h_i(\cdot)$ and $g_i(\cdot)$ concave	$O(n^2)$ [Zan69]	no improvement
$c_i(\cdot)$, $h_i(\cdot)$, and $g_i(\cdot)$ concave	$O(n^3)$ [Zan66, EMV87]	$O(n^2)$ Theorem 7.12

Table 7.2: A summary of our results for the economic lot-size problem with backlogging. The results are bounds on the time to find an optimal production schedule, where n is the number of periods.

use both paradigms, dynamic programming and searching in Monge arrays.

7.1.3 Two Periodic Models

Since market demands often display periodic behavior (which may arise, for example, because of the inherent cyclicity in seasonal demands), Erickson, Monma, and Veinott [EMV87] and Graves and Orlin [GO85] have studied two different variants of the backlogging economic lot-size problem that assume the planning horizon is infinite, i.e., we are planning for an infinite number of periods, but the costs and demands are periodic with period n.

Erickson et al. [EMV87] consider the problem of finding an infinite production schedule with minimum average cost per period, subject to the constraint that the production schedule also have period n. Equivalently, they want a minimum-cost n-period production schedule for periods i through $i+n-1$, where i is allowed to vary between 1 and n. Their model can be interpreted in a graph-theoretic sense as the backlogging flow network (given in Figure 7.2) with two additional arcs — one corresponding to inventory and the other corresponding to backlogging — between the first sink and the nth sink. For this problem, Erickson et al. obtained an $O(n^3)$-time algorithm.

The second periodic variant of the backlogging problem, considered by Graves and Orlin [GO85], is also concerned with finding an infinite production schedule with minimum average cost per period. However, the schedule is not restricted to have period n; instead, an assumption is made about the limiting behavior of the $g_i(\cdot)$ and $h_i(\cdot)$. (See Section 7.5 for more details.) For this problem, Graves and Orlin [GO85] give an $O(p^3n^3)$-time algorithm, where p is a parameter that depends upon production, inventory, and backlogging costs.

In Section 7.5, we give efficient algorithms for both Erickson, Monma, and Veinott's problem and Graves and Orlin's problem. The time complexities of these algorithms are given in Table 7.3.

7.2 Arborescent Flows and Dynamic Programming

As we mentioned in Subsections 7.1.1 and 7.1.2, both the basic and backlogging variants of the economic lot-size problem can be formulated as network-flow problems. Moreover, if the

problem and cost structure	*previous result*	*new result*
Erickson, Monma, and Veinott's problem $c_i(0) = 0$ $c_i(x) = c_i^0 + c^1 x$ for $x > 0$ $c_i^0 \geq 0$ $h_i(\cdot)$ and $g_i(\cdot)$ concave and nondecreasing	$O(n^3)$ [EMV87]	$O(n^2)$ Theorem 7.13
Erickson, Monma, and Veinott's problem $c_i(\cdot)$, $h_i(\cdot)$, and $g_i(\cdot)$ concave	$O(n^3)$ [EMV87]	no improvement
Graves and Orlin's problem $c_i(\cdot)$, $h_i(\cdot)$, and $g_i(\cdot)$ concave	$O(p^3 n^3)$ [GO85]	$O(p^2 n^3)$ Theorem 7.15

Table 7.3: A summary of our results for the two periodic economic lot-size problems. The results are bounds on the time to find an optimal production schedule, where n is the periodicity and p is a function of the $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$.

cost functions $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$ assigned to these networks' edges are all concave, then we need only consider flows of a certain type in finding a minimum-cost flow. Specifically, a flow in an uncapacitated directed graph G is called *arborescent* if the directed edges of G carrying nonzero flow, when viewed as undirected edges, form an undirected acyclic graph on the vertices of G. As the following theorem shows, we can restrict our attention to arborescent flows in network-flow problems with concave edge cost functions.

Theorem 7.1 (folklore; see [Zan68, EMV87]) Consider the flow problem associated with a directed graph G, where each arc e of G is assigned a cost function $c_e(\cdot)$ and the only constraint on the flow f_e on arc e is $f_e \geq 0$. If $c_e(\cdot)$ is concave on $[0, +\infty)$ for all arcs e, then some minimum-cost flow in G is arborescent. ■

This theorem appears (in one form or another) in all of the papers dealing with the economic lot-size problem that we consider. It is important because it implies that we need only consider production schedules that supply the demand for period i from at most one of the following sources: production during period i, inventory from period $i-1$, or, in the case of the backlogging model, demand backlogged to period $i+1$. Consequently, the basic and backlogging economic lot-size problems can be formulated in terms of dynamic programming. Specifically, let $E(1) = 0$, and for $1 < j \leq n+1$, let $E(j)$ denote the minimum cost of supplying the demands of periods

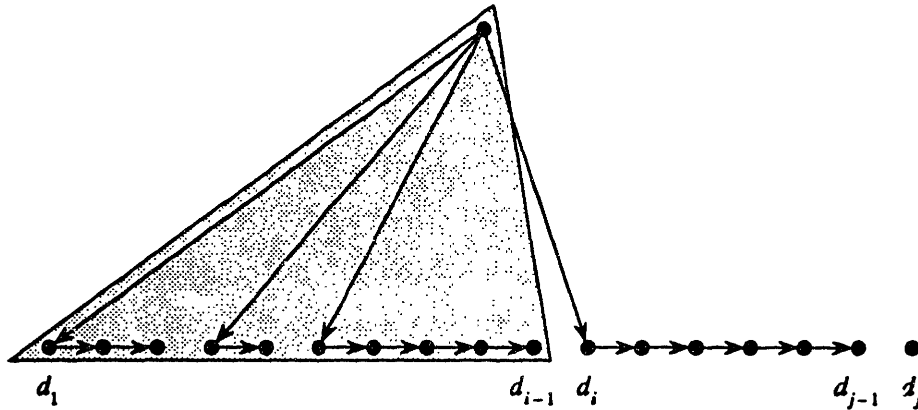

Figure 7.3: Consider any instance of the basic economic lot-size problem, and suppose P_j is a minimum-cost arborescent production schedule satisfying the demands of periods 1 through $j-1$ such that no inventory is carried forward to period j. Furthermore, suppose P_j's last production occurs during period i. Since P_j is arborescent, the demands of periods i through $j-1$ must all be satisfied by the production during period i. Moreover, the subschedule of P_j corresponding to periods 1 through $i-1$ (indicated by the shaded region) must be a minimum-cost arborescent production schedule satisfying the demands of periods 1 through $i-1$ such that no inventory is carried forward to period i.

1 through $j-1$ such that the inventory y_j carried forward to (or backlogged from) period j is zero. This definition implies that $E(n+1)$ is the cost of the desired optimal production schedule for periods 1 through n. Moreover, as suggested in Figures 7.3 and 7.4, if P_j is an optimal production schedule achieving $E(j)$, then there exists an i in the range $1 \leq i < j$ such that P_j can be decomposed into (1) a single period of production satisfying the demands of periods i through $j-1$, and (2) an optimal production schedule achieving $E(i)$. Thus, if we let $d_{i,j} = \sum_{m=i}^{j-1} d_m$ for $1 \leq i < j \leq n+1$, then for the basic problem,

$$E(j) = \min_{1 \leq i < j} \left\{ E(i) + c_i(d_{i,j}) + \sum_{m=i+1}^{j-1} h_m(d_{m,j}) \right\} ,$$

and for the backlogging problem,

$$E(j) = \min_{1 \leq i \leq k < j} \left\{ E(i) + c_k(d_{i,j}) + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j}) \right\} ,$$

provided we view summations of the form $\sum_{m=i}^{j}(\ldots)$ as evaluating to 0 if $i > j$.

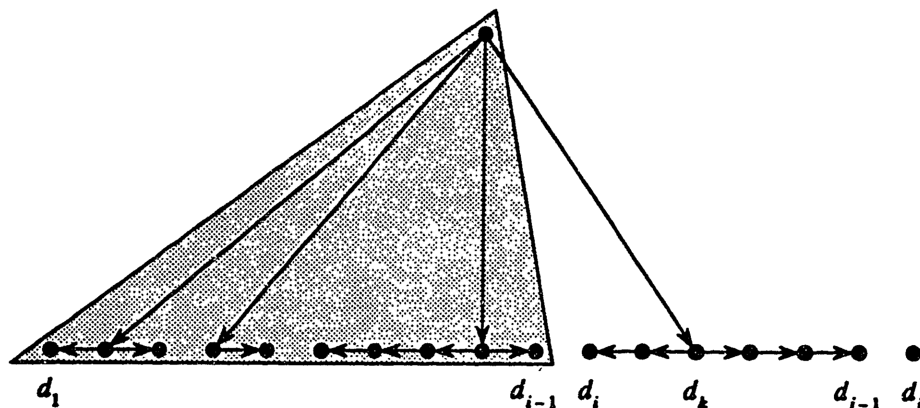

Figure 7.4: Consider any instance of the backlogging economic lot-size problem, and suppose P_j is a minimum-cost arborescent production schedule satisfying the demands of periods 1 through $j-1$ such that no inventory is carried forward to or backlogged from period j. Furthermore, suppose P_j's last production occurs during period k and that i is first period whose demand is satisfied by this production. Since P_j is arborescent, the demands of periods i through $j-1$ must all be satisfied by the production during period k. Moreover, the subschedule of P_j corresponding to periods 1 through $i-1$ (indicated by the shaded region) must be a minimum-cost arborescent production schedule satisfying the demands of periods 1 through $i-1$ such that no inventory is carried forward to or backlogged from period i.

Note that these dynamic programming formulations for the basic and backlogging economic lot-size problems give $O(n^2)$-time and $O(n^3)$-time algorithms, respectively, for computing the cost of an optimal production schedule; we merely evaluate $E(1), E(2), \ldots, E(n+1)$ in the naive fashion. Furthermore, we can extract an optimal production schedule (not just its cost) in $O(n)$ additional time, provided for each $E(j)$ we remember the i such that

$$E(j) \;=\; E(i) + c_i(d_{i,j}) + \sum_{m=i+1}^{j-1} h_m(d_{m,j})$$

or the i and k such that

$$E(j) \;=\; E(i) + c_k(d_{i,j}) + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j})\,.$$

7.3 The Basic Problem

This section investigates the time complexity of the basic economic lot-size problem under several different assumptions about the production and inventory cost functions. In Sub-

section 7.3.1, we consider nearly linear production costs and linear inventory costs, while in Subsection 7.3.2, we discuss other concave production and inventory cost functions.

7.3.1 Nearly Linear Costs

In this subsection, we give results for instances of the basic economic lot-size problem with what we will call *nearly linear* costs. Specifically, for $1 \leq i \leq n$, we assume

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

and $h_i(y) = h_i^1 y$, where c_i^0, c_i^1, and h_i^1 are constants and $c_i^0 \geq 0$. (The restriction on c_i^0 is necessary to insure that $c_i(x)$ is concave, so that the techniques of Section 7.2 can be applied.) In the operations-research literature, this cost structure is often described as consisting of *fixed-plus-linear* production costs and linear inventory costs.

We begin with a special case in Subsubsection 7.3.1: for $1 < i \leq n$, we assume $c_i^1 \leq c_{i-1}^1 + h_i^1$. For this special case of the basic lot-size problem, we give a linear-time algorithm for computing the optimal production schedule. Then, in Subsubsection 7.3.1, we remove this constraint on the coefficients of the cost functions, at the expense of an increase in our algorithm's running time by factor of $\lg n$.

Restricted Coefficients

In this subsubsection, we consider a nearly linear cost structure where the cost coefficients satisfy $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$. In other words, we assume that the marginal cost of producing during period i is at most the marginal cost of producing during period $i-1$ plus the marginal cost of storing inventory from period $i-1$ to period i. This particular cost structure subsumes those considered by Manne [Man58] and Wagner and Whitin [WW58]. The latter paper gave an $O(n^2)$-time algorithm for computing an optimal production schedule; we improve this time bound to $O(n)$ for our slightly more general cost structure.

Recall the dynamic programming formulation of the basic economic lot-size problem given in Section 7.2: if we let $E(j)$ denote the minimum cost of satisfying the demands of periods 1

through $j-1$ such that the inventory y_j carried forward from period $j-1$ to period j is 0, then $E(1)=0$ and for $2 \le j \le n+1$,

$$E(j) = \min_{1 \le i < j} \left\{ E(i) + c_i(d_{i,j}) + \sum_{m=i+1}^{j-1} h_m(d_{m,j}) \right\}$$

where $d_{i,j} = d_i + d_{i+1} + \cdots + d_{j-1}$. Solving this dynamic program in the naive fashion gives the $O(n^2)$-time algorithm presented in [WW58].

To compute $E(2), \ldots, E(n+1)$ in linear time, we consider the $n \times (n+1)$ array $A = \{a[i,j]\}$ where

$$a[i,j] = \begin{cases} E(i) + c_i^0 + c_i^1 d_{i,j} + \sum_{m=i+1}^{j-1} h_m^1 d_{m,j} & \text{if } i < j, \\ +\infty & \text{if } i \ge j. \end{cases}$$

(Instead, one is tempted to use the $n \times (n+1)$ array $B = \{b[i,j]\}$ where

$$b[i,j] = \begin{cases} E(i) + c_i(d_{i,j}) + \sum_{m=i+1}^{j-1} h_m^1 d_{m,j} & \text{if } i < j, \\ +\infty & \text{if } i \ge j, \end{cases}$$

but this array may not be Monge; for example, if $d_1 > 0$, $d_2 = 0$, and $d_3 > 0$, then

$$b[1,3] + b[2,4] - b[1,4] - b[2,3] = c_1^0 + (c_2^1 - (c_1^1 + h_2^1))d_3 ,$$

which may be positive if c_1^0 is sufficiently large.) Now if $d_{j-1} = 0$, then $E(j) = E(j-1)$. On the other hand, if $d_{j-1} > 0$, then $d_{m,j} > 0$ for all $m < j$, which implies

$$E(i) + c_i(d_{i,j}) + \sum_{m=i+1}^{j-1} h_m(d_{m,j}) = a[i,j]$$

for all $i < j$, and

$$E(j) = \min_{1 \le i \le n} a[i,j] .$$

Combining these two observations gives the following recurrence for $E(j)$ when $2 \leq j \leq n+1$:

$$E(j) = \begin{cases} E(j-1) & \text{if } d_{j-1} = 0, \\ \min_{1 \leq i \leq n} a[i,j] & \text{if } d_{j-1} > 0. \end{cases}$$

At this point, we would like to apply the on-line LIEBER algorithm described in Section 2.2 to compute the column minima of A (and hence $E(2), \ldots, E(n+1)$). Since $a[i,j]$ depends only on the minimum entries in columns 1 through i of A, all that remains to be shown is that the array A is Monge and that any entry $a[i,j]$ of A can be computed in constant time given $E(i)$.

Lemma 7.2 A is Monge.

Proof For $1 \leq i < j \leq n+1$,

$$\begin{aligned} a[i,j] &= E(i) + c_i^0 + c_i^1 d_{i,j} + \sum_{m=i+1}^{j-1} h_m^1 d_{m,j} \\ &= E(i) + c_i^0 + c_i^1 (d_{1,j} - d_{1,i}) + \sum_{m=1}^{j-1} h_m^1 d_{m,j} - \sum_{m=1}^{i} h_m^1 (d_{1,j} - d_{1,m}) \\ &= \left[E(i) + c_i^0 - c_i^1 d_{1,i} + \sum_{m=1}^{i} h_m^1 d_{1,m} \right] + \left[\sum_{m=1}^{j-1} h_m^1 d_{m,j} \right] + \left[\left(c_i^1 - \sum_{m=1}^{i} h_m^1 \right) d_{1,j} \right] . \end{aligned}$$

Now consider the $n \times (n+1)$ array $A' = \{a'[i,j]\}$ where

$$a'[i,j] = \left[E(i) + c_i^0 - c_i^1 d_{1,i} + \sum_{m=1}^{i} h_m^1 d_{1,m} \right] + \left[\sum_{m=1}^{j-1} h_m^1 d_{m,j} \right] + \left[\left(c_i^1 - \sum_{m=1}^{i} h_m^1 \right) d_{1,j} \right]$$

for $1 \leq i \leq n$ and $1 \leq j \leq n+1$. If we can show that A' is Monge, then A must also be Monge, since every 2×2 subarray of A is either a 2×2 subarray of A' or its left- and bottommost entry is a $+\infty$.

To show that A' satisfies the Monge condition, note that the first bracketed term in its definition depends only on i, the second bracketed term depends only on j, and the third bracketed term is the product of

$$c_i^1 - \sum_{m=1}^{i} h_m^1 ,$$

which depends only on i, and $d_{1,j}$, which depends only on j. Furthermore,

$$c_1^1 - \sum_{m=1}^{1} h_m^1 \geq c_2^1 - \sum_{m=1}^{2} h_m^1 \geq \cdots \geq c_n^1 - \sum_{m=1}^{n} h_m^1$$

(since, by assumption, $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$) and

$$0 = d_{1,1} \leq d_{1,2} \leq \cdots \leq d_{1,n+1}$$

(since, by assumption, $d_i \geq 0$ for $1 \leq i \leq n$). Thus, by Lemma 1.2, A' is Monge. ∎

Lemma 7.3 Given $O(n)$ preprocessing time, we can compute $a[i,j]$ from $E(i)$ in constant time, for all i and j.

Proof If $i \geq j$, then $a[i,j] = +\infty$, i.e., computing the entry is easy. If, on the other hand, $i < j$, then

$$a[i,j] = E(i) + c_i^0 + c_i^1 d_{i,j} + \sum_{m=i+1}^{j-1} h_m^1 d_{m,j} .$$

Now suppose we precompute $d_{1,i}$ for $1 \leq i \leq n$, which takes $O(n)$ time. This preprocessing gives us any $d_{i,j}$ in constant time, since $d_{i,j} = d_{1,j} - d_{1,i}$. Suppose we also precompute

$$\sum_{m=1}^{j-1} h_m^1$$

for $2 \leq j \leq n+1$. This preprocessing again takes $O(n)$ time, and it allows us to precompute

$$\sum_{m=1}^{j-1} h_m^1 d_{m,j}$$

for $2 \leq j \leq n+1$ in an additional $O(n)$ time, since

$$\sum_{m=1}^{j-1} h_m^1 d_{m,j} = \sum_{m=1}^{j-2} h_m^1 d_{m,j-1} + \left(\sum_{m=1}^{j-1} h_m^1\right) d_{j-1} .$$

Moreover, since

$$\sum_{m=i+1}^{j-1} h_m^1 d_{m,j} = \sum_{m=1}^{j-1} h_m^1 d_{m,j} - \sum_{m=1}^{i} h_m^1 d_{m,i+1} - \left(\sum_{m=1}^{i} h_m^1\right) d_{i+1,j} ,$$

these precomputations allow us to compute $a[i,j]$ from $E(i)$ in constant time. ■

Lemmas 7.2 and 7.3 allow us to use the LIEBER algorithm described in Section 2.2 to compute the column minima of A and hence $E(2),\ldots,E(n+1)$ in linear time. Consequently, we have the following theorem.

Theorem 7.4 Given an n-period instance of the basic economic lot-size problem such that

- for $1 \le i \le n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

  where c_i^0 and c_i^1 are constants and $c_i^0 \ge 0$,
- for $1 \le i \le n$, $h_i(y) = h_i^1 y$, where h_i^1 is a constant, and
- for $1 < i \le n$, $c_i^1 \le c_{i-1}^1 + h_i^1$,

we can find an optimal production schedule in $O(n)$ time. ■

Arbitrary Coefficients

In this subsubsection, we remove the constraint that $c_i^1 \le c_{i-1}^1 + h_i^1$ for $1 < i \le n$ and allow the c_i^1 and h_i^1 to be arbitrary constants. This cost structure is the one considered by Zabel [Zab64] and by Eppen, Gould, and Pashigian [EGP69]. Both papers gave $O(n^2)$-time algorithms for this variant of the basic economic lot-size problem; we improve this time bound to $O(n \lg n)$.

With arbitrary coefficients c_i^1 and h_i^1, the array A defined in the last subsection no longer satisfies the Monge condition, since we no longer have

$$c_1^1 - \sum_{m=1}^{1} h_m^1 \ge c_2^1 - \sum_{m=1}^{2} h_m^1 \ge \cdots \ge c_n^1 - \sum_{m=1}^{n} h_m^1 .$$

However, we can circumvent this difficulty by reordering the rows of A. Intuitively, we sort the n quantities $r_1,\ldots,r_n$, where

$$r_i = c_i^1 - \sum_{m=1}^{i} h_m^1 ,$$

i.e., we find a permutation γ such that $r_{\gamma(1)} \ge r_{\gamma(2)} \ge \cdots \ge r_{\gamma(n)}$. If we then use γ to permute the rows of A, we obtain a new array that is Monge-like.

We will now give a precise description of our $O(n \lg n)$-time algorithm for the basic economic lot-size problem with nearly linear costs. The algorithm uses a divide-and-conquer approach, and it involves solving several subproblems, each corresponding to a range of consecutive periods. These subproblems are slightly more general than the basic economic lot-size problem, in that solving the subproblem corresponding to periods s through $t-1$ involves computing $E(j)$ for $s < i \leq t$, where $E(j)$ corresponds to an optimal production schedule for periods 1 through $j-1$ (rather than periods s through $j-1$). In particular, the schedule corresponding to $E(j)$ may have its last nonzero production occur in some period $i < s$.

To describe our algorithm in detail, we must first introduce some new notation. For $1 \leq s \leq n$ and $s < j \leq n+1$, let

$$F_s(j) = \min_{1 \leq i < s} a[i,j] .$$

Roughly speaking, $F_s(j)$ is the cost of the minimum-cost production schedule satisfying the demands of periods 1 through $j-1$ such that the inventory y_j carried forward from period $j-1$ to period j is 0 *and* the schedule's last nonzero production occurs in some period $i < s$. For a subproblem corresponding to periods $s, \ldots, t-1$ and for $s < j \leq t$, we then have

$$E(j) = \begin{cases} E(j-1) & \text{if } d_{j-1} = 0, \\ \min\left\{F_s(j), \min_{s \leq i < t} a[i,j]\right\} & \text{if } d_{j-1} > 0. \end{cases}$$

Note that so long as $E(s)$ and $F_s(s+1), \ldots, F_s(t)$ are known, the only entries of A that we need to consider in computing $E(s+1), \ldots, E(t)$ are those lying in the subarray of A consisting of rows s through $t-1$ and columns $s+1$ through t. This subarray is depicted in Figure 7.5.

For $1 \leq s < t \leq n+1$, we can now define the subproblem corresponding to periods s through $t-1$ as follows. Given $E(s)$ and $F_s(s+1), \ldots, F_s(t)$, solving this subproblem entails

1. computing $E(j)$ for $s+1 \leq j \leq t$, and

2. sorting $r_s, \ldots, r_{t-1}$, where

$$r_i = c_i^1 - \sum_{m=1}^{i} h_m^1 ,$$

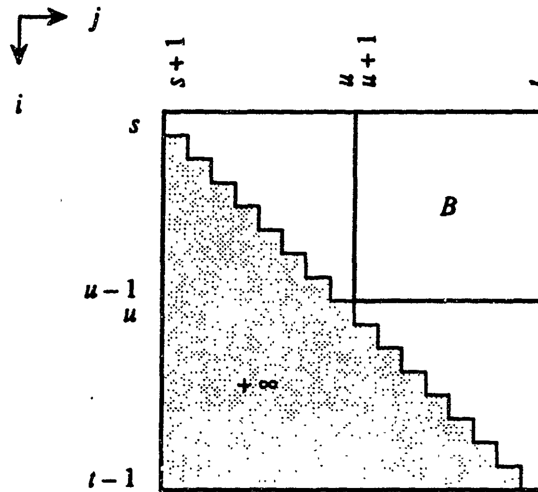

Figure 7.5: Given $E(s)$ and $F_s(s+1), \ldots, F_s(t)$, where $1 \leq s < t \leq n+1$, we can compute $E(s+1), \ldots, E(t)$ from the entries in rows s through t and columns $s+1$ through $t+1$ of A.

i.e., finding a permutation $\gamma_{s,t}$ such that

$$r_{\gamma_{s,t}(1)+s-1} \geq r_{\gamma_{s,t}(2)+s-1} \geq \cdots \geq r_{\gamma_{s,t}(t-s)+s-1} \,.$$

(Our reason for including the computation of $\gamma_{s,t}$ as part of the subproblem will become apparent in a moment.) Since $E(1) = 0$ and $F_1(j) = \infty$ for $2 \leq j \leq n+1$, solving the subproblem corresponding to periods 1 through $n+1$ gives us a solution for the original n-period economic lot-size problem.

To solve the subproblem corresponding to periods s through $t-1$ given $E(s)$ and $F_s(s+1)$ through $F_s(t)$, we first recursively solve the subproblem corresponding to periods s through $u-1$, where $u = \lfloor (s+t)/2 \rfloor$. This recursive computation is possible because $E(s)$ and $F_s(s+1), \ldots, F_s(u)$ are known. Solving this subproblem gives us $E(s+1)$ through $E(u)$ and the permutation $\gamma_{s,u}$.

Next, we compute the column minima of the subarray B consisting of rows s through $u-1$ and columns $u+1$ through t of A. (See Figure 7.5.) To find these column minima, we first permute the rows of B according to the permutation $\gamma_{s,u}$ obtained by solving the subproblem corresponding to periods s through $u-1$. This permutation gives the $(u-s) \times (t-u)$ array

$B' = b'[i,j]$ where

$$b'[i,j] = \left[E(\gamma_{s,u}(i)+s-1) + c^0_{\gamma_{s,u}(i)+s-1} - c^1_{\gamma_{s,u}(i)+s-1} d_{1,\gamma_{s,u}(i)+s-1} + \sum_{m=1}^{\gamma_{s,u}(i)+s-1} h^1_m d_{1,m}\right] + \left[\sum_{m=1}^{j+u-1} h^1_m d_{m,j+u}\right] + \left[r_{\gamma_{s,u}(i)+s-1} d_{1,j+u}\right] .$$

The column minima of B' are the column minima of B. Moreover, the first term in the sum defining $b'[i,j]$ depends only on i, the second term depends only on j, and the third term is the product $r_{\gamma_{s,u}(i)+s-1} d_{1,j+u}$, where

$$r_{\gamma_{s,u}(1)+s-1} \geq r_{\gamma_{s,u}(2)+s-1} \geq \cdots \geq r_{\gamma_{s,u}(u-s)+s-1}$$

and

$$d_{1,u+1} \leq d_{1,u+2} \leq \cdots \leq d_{1,t} .$$

Thus, by Lemma 1.2, B' is Monge. Furthermore, using the $O(n)$-time preprocessing described in the previous subsubsection, we can compute any entry of B' in constant time, since $E(s+1)$ through $E(u)$ are known. Thus, we can apply the off-line SMAWK algorithm of Aggarwal et al. and obtain the column minima of B in $O(t-s)$ time.

Given the column minima of B, we can now compute $F_u(u+1), \ldots, F_u(t)$, since for $u < j \leq t$,

$$F_u(j) = \min\left\{F_s(j), \min_{s \leq i < u} a[i,j]\right\}$$

and

$$\min_{s \leq i < u} a[i,j]$$

is the minimum entry in the $(j-u)$th column of B. This computation requires only $O(t-u) = O(t-s)$ additional time.

Once $F_u(u+1)$ through $F_u(t)$ are known, we recursively solve the subproblem corresponding to periods u through $t-1$ using $E(u)$ and $F_u(u+1), \ldots, F_u(t)$. This recursive computation gives $E(u+1)$ through $E(t)$ and the permutation $\gamma_{u,t}$.

As the final step of our algorithm, we compute the permutation $\gamma_{s,t}$ from the permutations $\gamma_{s,u}$ and $\gamma_{u,t}$. This computation can be accomplished in $O(t-s)$ time by merging the two sorted lists of r_i's corresponding to $\gamma_{s,u}$ and $\gamma_{u,t}$. (We assume that for $1 \leq i \leq n$,

$$\sum_{m=1}^{i} h_m^1$$

has been precomputed, so that any r_i can be computed in constant time; this preprocessing requires only $O(n)$ time.)

The running time $T(s,t)$ of this algorithm for the subproblem corresponding to periods s through $t-1$ is governed by the recurrence

$$T(s,t) = \begin{cases} T(s, \lfloor (s+t)/2 \rfloor) + T(\lfloor (s+t)/2 \rfloor, t) + O(t-s) & \text{if } t-s > 1, \\ O(1) & \text{if } t-s = 1, \end{cases}$$

which has as its solution $T(s,t) = O((t-s)\lg(t-s))$. Thus, $T(1, n+1) = O(n \lg n)$, which gives the following theorem.

Theorem 7.5 Given an n-period instance of the basic economic lot-size problem such that

- for $1 \leq i \leq n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

  where c_i^0 and c_i^1 are constants and $c_i^0 \geq 0$, and

- for $1 \leq i \leq n$, $h_i(y) = h_i^1 y$, where h_i^1 is a constant,

we can find an optimal production schedule in $O(n \lg n)$ time. ∎

7.3.2 Other Cost Structures

In the previous subsection, we assumed nearly linear production costs and linear inventory costs. These assumptions allowed us to prove that certain arrays arising in the context of the basic economic lot-size problem were Monge, and it was the Mongeness of these arrays that allowed us to give improved algorithms for the basic problem with nearly linear costs. If one

tries to generalize this approach to arbitrary concave production and inventory cost functions (and improve upon the $O(n^2)$-time algorithm of Veinott [Vei63]), however, one notes that the corresponding arrays need not be Monge. Consequently, the question of whether it is possible to obtain a subquadratic algorithm for the basic economic lot-size problem with arbitrary concave costs remains open. (See Section 7.6.)

Note that even if the array $A = \{a[i,j]\}$ defined in the last subsection were Monge under less restrictive assumptions about the production and inventory cost functions, the computation of its column minima might still take $\Omega(n^2)$ time. This possibility stems from our need to be able to compute any entry $a[i,j]$ in constant time, given the minimum entries in columns 1 through i of A. In fact, there are cost structures where this entry computation time turns out to be a time bottleneck. Specifically, consider the cost structure studied by Zangwill in [Zan69]. (See also Subsection 7.4.3 and Section 7.6.) Zangwill assumed that for $1 \le i \le n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

where c_i^0 and c^1 are constants, and $h_i(\cdot)$ is a nondecreasing concave function. (Note that the marginal cost of production c^1 is the same for all time periods; this assumption is needed to insure that the array A defined below is Monge.) If we consider the $n \times (n+1)$ array $A = \{a[i,j]\}$ where

$$a[i,j] = \begin{cases} E(i) + c_i^0 + c^1 d_{i,j} + \sum_{m=i+1}^{j-1} h_m(d_{m,j}) & \text{if } i < j, \\ +\infty & \text{if } i \ge j \end{cases}$$

then A is Monge. This claim follows because

$$a[i,j] + a[i+1,j+1] - a[i,j+1] - a[i+1,j] = h_{i+1}(d_{i+1,j}) - h_{i+1}(d_{i+1,j+1})$$

for $1 < i+1 < j < n+1$, and the right-hand side of this equation is nonpositive so long as $h_{i+1}(\cdot)$ is a nondecreasing function. However, it is unclear how to compute $a[i,j]$ in constant time, given the minimum entries in columns 1 through i of A, as $a[i,j]$ depends on $\sum_{m=i+1}^{j-1} h_m^1(d_{m,j})$ and we do not know of any $o(n^2)$-time preprocessing that would allow us to compute this sum for

any i and j in constant time.

7.4 The Backlogging Problem

This section investigates the time complexity of the backlogging economic lot-size problem under several different assumptions about the production, inventory, and backlogging cost functions. In Subsection 7.4.1, we consider nearly linear production costs and linear inventory and backlogging costs. In Subsection 7.4.2, we focus on arbitrary concave production, inventory, and backlogging cost functions. Finally, in Subsection 7.4.3, we discuss arbitrary concave inventory and backlogging cost functions together with nearly linear production cost functions such that the marginal cost of production is the same for all periods.

7.4.1 Nearly Linear Costs

In this subsection, we give results for instances of the backlogging economic lot-size problem with nearly linear costs. Specifically, for $1 \le i \le n$, we assume

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

$h_i(y) = h_i^1 y$, and $g_i(z) = g_i^1 z$, where the c_i^0, c_i^1, h_i^1, and g_i^1 are constants and c_i^0 is restricted to be nonnegative for $1 \le i \le n$. This problem is similar to the basic problem with nearly linear costs considered in Subsection 7.3.1, except that here we are faced with a three-dimensional Monge array rather than a two-dimensional Monge array.

We begin with a special case in Subsubsection 7.4.1: for $1 \le i < n$, we assume $c_i^1 \le c_{i+1}^1 + g_i^1$ for $1 \le i < n$ and $c_i^1 \le c_{i-1}^1 + h_i^1$ for $1 < i \le n$. For this special case of the backlogging problem, we give an $O(n)$-time algorithm for computing the optimal production schedule. Then, in Subsubsection 7.4.1, we remove the constraint on the coefficients of the cost functions and give an $O(n \lg n)$-time algorithm for the backlogging problem with nearly linear costs.

Restricted Coefficients

In this subsubsection, we consider a nearly linear cost structure where the cost coefficients satisfy $c_i^1 \leq c_{i+1}^1 + g_i^1$ for $1 \leq i < n$ and $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$. This particular cost structure subsumes the cost structure considered by Morton [Mor78]. Morton gave an $O(n^2)$-time algorithm for his problem; we improve this time bound to $O(n)$ for our more general cost structure.

Recall the dynamic programming formulation of the backlogging economic lot-size problem given in Section 7.2: if we let $E(j)$ denote the minimum cost of satisfying the demands of periods 1 through $j-1$ such that $y_j = 0$ (i.e., no inventory is stored from period $j-1$ to period j, nor is any demand backlogged from period $j-1$ to period j), then $E(1) = 0$ and for $2 \leq j \leq n+1$,

$$E(j) = \min_{1 \leq i \leq k < j} \left\{ E(i) + c_k(d_{i,j}) + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j}) \right\}$$

where $d_{i,j} = d_i + d_{i+1} + \cdots + d_{j-1}$.

To compute $E(2), \ldots, E(n+1)$ in $O(n)$ time, we consider the $n \times (n+1) \times n$ array $A = \{a[i,j,k]\}$ where

$$a[i,j,k] = \begin{cases} E(i) + c_k^0 + c_k^1 d_{i,j} + \sum_{m=i}^{k-1} g_m^1 d_{i,m+1} + \sum_{m=k+1}^{j-1} h_m^1 d_{m,j} & \text{if } i \leq k < j, \\ +\infty & \text{otherwise.} \end{cases}$$

Now if $d_{j-1} = 0$, then either some optimal production schedule produces during period $j-1$, in which case

$$E(i) + c_{j-1}(d_{i,j}) + \sum_{m=i}^{j-2} g_m(d_{i,m+1}) + \sum_{m=j}^{j-1} h_m(d_{m,j}) = a[i,j,j-1]$$

for $i < j$ and

$$E(j) = \min_{1 \leq i \leq n} a[i,j,j-1] \leq E(j-1),$$

or some optimal schedule does not produce during period j, in which case

$$E(j) = E(j-1) \leq \min_{1 \leq i \leq n} a[i, j, j-1] .$$

On the other hand, if $d_{j-1} > 0$, then $d_{i,j} > 0$ for all $i < j$, which implies

$$E(i) + c_k(d_{i,j}) + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j}) = a[i, j, k]$$

for $i \leq k < j$ and

$$E(j) = \min_{\substack{1 \leq i \leq n \\ 1 \leq k \leq n}} a[i, j, k] .$$

These observations give the following recurrence for $E(j)$ when $2 \leq j \leq n+1$:

$$E(j) = \begin{cases} \min\{E(j-1), \min_{1 \leq i \leq n} a[i, j, j-1]\} & \text{if } d_{j-1} = 0, \\ \min_{\substack{1 \leq i \leq n \\ 1 \leq k \leq n}} a[i, j, k] & \text{if } d_{j-1} > 0. \end{cases}$$

Now observe that the three-dimensional array A define above can be decomposed into two two-dimensional arrays S and T. (Zangwill uses essentially this same decomposition in [Zan69] to obtain an $O(n^2)$-time algorithm for a variant of this problem.) Specifically, let $S = \{s[i, k]\}$ denote the $n \times n$ array given by the equation

$$s[i, k] = \begin{cases} E(i) + c_k^0 + c_k^1 d_{i,k} + \sum_{m=i}^{k-1} g_m^1 d_{i,m+1} & \text{if } i < k, \\ +\infty & \text{if } i \geq k, \end{cases}$$

and let $T = \{t[k, j]\}$ denote the $n \times (n+1)$ array given by the equation

$$t[k, j] = \begin{cases} F(k) + c_k^1 d_{k,j} + \sum_{m=k+1}^{j-1} h_m^1 d_{m,j} & \text{if } k < j, \\ +\infty & \text{if } k \geq j. \end{cases}$$

where

$$F(k) = \min\{E(k), \min_{1 \leq i \leq n} s[i, k]\} .$$

Since

$$\begin{aligned}
\min_{\substack{1\le i\le n\\1\le k\le n}} a[i,j,k]
&= \min_{1\le k<j}\left\{\min_{1\le i\le k}\left\{E(i)+c_k^0+c_k^1 d_{i,k}+\sum_{m=i}^{k-1} g_m^1 d_{i,m+1}\right\}+c_k^1 d_{k,j}+\sum_{m=k+1}^{j-1} h_m^1 d_{m,j}\right\}\\
&= \min_{1\le k<j}\left\{F(k)+c_k^1 d_{k,j}+\sum_{m=k+1}^{j-1} h_m^1 d_{m,j}\right\}\\
&= \min_{1\le k\le n} t[k,j]
\end{aligned}$$

and

$$\min_{1\le i\le n} a[i,j,j-1] = \min_{1\le i\le n} s[i,j-1]\ ,$$

we have

$$E(j) = \begin{cases} F(j-1) & \text{if } d_{j-1}=0,\\ \min_{1\le k\le n} t[k,j] & \text{if } d_{j-1}>0. \end{cases}$$

Thus, to compute $E(2),\ldots,E(n+1)$, we need merely compute the column minima of S and T. (In terms of the definitions of Section 1.2, the three-dimensional array A is *path-decomposable*; this structure is what allows the plane-minima problem for A to be decomposed into two column-minima problems for two-dimensional arrays.)

Using arguments similar to those used in proving Lemma 7.3, it is not hard to show that, after linear preprocessing time, any entry $s[i,k]$ of S can be computed in constant time from the minimum entries in columns 1 through i of T and the minimum entries in columns 1 through $i-1$ of S, and any entry $t[k,j]$ of T can be computed in constant time from the minimum entries in columns 1 through k of S and the minimum entries in columns 1 through k of T. Furthermore, both S and T are Monge, as the following two lemmas show.

Lemma 7.6 S is Monge.

Proof For $1\le i\le k\le n$,

$$s[i,k] = E(i)+c_k^1 d_{i,k}+\sum_{m=i}^{k-1} g_m^1 d_{i,m+1}$$

$$= E(i) + c_k^1(d_{1,k} - d_{1,i}) + \sum_{m=1}^{k-1} g_m^1(d_{1,m+1} - d_{1,i}) - \sum_{m=1}^{i-1} g_m^1 d_{i,m+1}$$

$$= \left[E(i) - \sum_{m=1}^{i-1} g_m^1 d_{i,m+1}\right] + \left[c_k^1 d_{1,k} + \sum_{m=1}^{k-1} g_m^1 d_{1,m+1}\right] + \left[-\left(c_k^1 + \sum_{m=1}^{k-1} g_m^1\right) d_{1,i}\right] .$$

Now consider the $n \times n$ array $S' = \{s'[i,k]\}$ where

$$s'[i,k] = \left[E(i) - \sum_{m=1}^{i-1} g_m^1 d_{i,m+1}\right] + \left[c_k^1 d_{1,k} + \sum_{m=1}^{k-1} g_m^1 d_{1,m+1}\right] + \left[-\left(c_k^1 + \sum_{m=1}^{k-1} g_m^1\right) d_{1,i}\right]$$

for $1 \le i \le n$ and $1 \le k \le n$. If we can show that S' is Monge, then S must also be Monge, since every 2×2 subarray of S is either a 2×2 subarray of S' or its left- and bottommost entry is a $+\infty$.

To show that S' is Monge, note that the first bracketed term in its definition depends only on i, the second bracketed term depends only on k, and the third bracketed term is the product of $d_{1,i}$, which depends only on i, and

$$-\left(c_k^1 + \sum_{m=1}^{k-1} g_m^1\right) ,$$

which depends only on k. Furthermore,

$$0 = d_{1,1} \le d_{1,2} \le \cdots \le d_{1,n}$$

(since, by assumption, $d_i \ge 0$ for $1 \le i \le n$), and

$$c_1^1 + \sum_{m=1}^{0} g_m^1 \le c_2^1 + \sum_{m=1}^{1} g_m^1 \le \cdots \le c_n^1 + \sum_{m=1}^{n-1} g_m^1$$

(since, by assumption, $c_i^1 \le c_{i+1}^1 + g_i^1$ for $1 \le i < n$). Thus, by Lemma 1.2, S' is Monge. ∎

Lemma 7.7 T is Monge.

Proof The proof for this lemma is very similar to that for Lemma 7.2. ∎

At this point, we would like to apply the on-line LIEBER algorithm as we did in Subsubsection 7.3.1. The only complication here is that we now have two arrays, not one, and the arrays'

entries depend on each other's column minima. We can get around this difficulty, however, by interleaving the computation of S's column minima and the computation of T's column minima, just as we interleaved the computation of A's column minima and the computation of B's column minima in Section 6.1. Thus, we have the following theorem.

Theorem 7.8 Given an n-period instance of the backlogging economic lot-size problem such that

- for $1 \leq i \leq n$,
$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$
where c_i^0 and c_i^1 are constants and $c_i^0 \geq 0$,
- for $1 \leq i \leq n$, $h_i(y) = h_i^1 y$, where h_i^1 is a constant,
- for $1 \leq i \leq n$, $g_i(z) = g_i^1 z$, where g_i^1 is a constant,
- for $1 < i \leq n$, $c_i^1 \leq c_{i-1}^1 + h_i^1$, and
- for $1 \leq i < n$, $c_i^1 \leq c_{i+1}^1 + g_i^1$,

we can find an optimal production schedule in $O(n)$ time. ■

Arbitrary Coefficients

In this subsubsection, we allow the c_i^1, g_i^1, and h_i^1 to be arbitrary constants, i.e., we no longer assume that $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$ and that $c_i^1 \leq c_{i+1}^1 + g_i^1$ for $1 \leq i < n$. This cost structure was considered by Blackburn and Kunreuther [BK74] and by Lundin and Morton [LM75]. Both papers gave $O(n^2)$-time algorithms for this variant of the backlogging economic lot-size problem; we improve this time bound to $O(n \lg n)$.

As in Subsubsection 7.3.1, if we allow arbitrary coefficients c_i^1, g_i^1, and h_i^1, then it is easy to verify that the arrays S and T defined in the previous subsubsection no longer satisfy the Monge condition. However, we can circumvent this difficulty by reordering the rows of S and T, just as we reordered the rows of A in Subsubsection 7.3.1. Specifically, let

$$q_k = -\left(c_k^1 + \sum_{m=1}^{k} g_m^1\right)$$

for $1 \le k \le n$, and let

$$r_i = c_i^1 - \sum_{m=1}^{i} h_m^1$$

for $1 \le i \le n$. If we sort $q_1, \ldots, q_n$ and $r_1, \ldots, r_n$, obtaining permutations β and γ such that $q_{\beta(1)} \ge q_{\beta(2)} \ge \cdots \ge q_{\beta(n)}$ and $r_{\gamma(1)} \ge r_{\gamma(2)} \ge \cdots \ge r_{\gamma(n)}$, then the finite entries of $S' = \{s'[i,k]\}$ where $s'[i,k] = s[i,\beta(k)]$ and $T' = \{t'[k,j]\}$ where $t'[k,j] = t[\gamma(k),j]$ satisfy the Monge condition. Combining this observation with the divide-and-conquer approach of Subsubsection 7.3.1, it is straightforward to obtain the following theorem.

Theorem 7.9 Given an n-period instance of the backlogging economic lot-size problem such that

- for $1 \le i \le n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

  where c_i^0 and c_i^1 are constants and $c_i^0 \ge 0$,
- for $1 \le i \le n$, $g_i(y) = g_i^1 y$, where g_i^1 is a constant, and
- for $1 \le i \le n$, $h_i(y) = h_i^1 y$, where h_i^1 is a constant,

we can find an optimal production schedule in $O(n \lg n)$ time.

7.4.2 Concave Costs

In this subsection, we consider the backlogging economic lot-size problem with arbitrary concave costs, i.e., we assume only that the cost functions $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$ are concave. In [Zan66], Zangwill gave an $O(n^3)$-time algorithm for this problem; we reduce this time bound to $O(n^2)$.

Our algorithm for this variant of the backlogging economic lot-size problem is reminiscent of the algorithm for Yao's problem described in Section 6.2, as we again define a three-dimensional array whose tube minima we compute one plane at a time. However, this subsection's three-dimensional array is not Monge (though certain of its two-dimensional planes are), and we use the off-line SMAWK algorithm (rather than the on-line LIEBER algorithm) to compute the planes' minima.

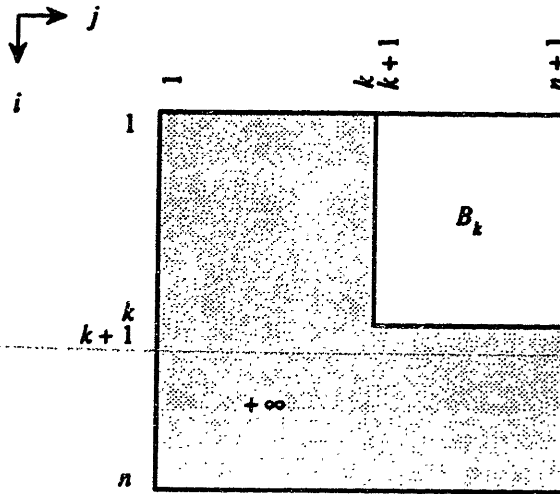

Figure 7.6: For any k in the range $1 \le k \le n$, the only finite entries in the plane A_k lie in the subarray B_k consisting of rows 1 through k and columns $k+1$ through $n+1$ of A_k; moreover, all the entries in B_k are finite.

We will now describe our algorithm. Let $A = \{a[i,j,k]\}$ denote the $n \times (n+1) \times n$ array where

$$a[i,j,k] = \begin{cases} E(i) + c_k(d_{i,j}) + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j}) & \text{if } i \le k < j, \\ +\infty & \text{otherwise.} \end{cases}$$

Furthermore, for $1 \le k \le n$, let $A_k = \{a_k[i,j]\}$ denote the $n \times (n+1)$ two-dimensional plane of A corresponding to those entries whose third coordinate is k, and let $B_k = \{b_k[s,t]\}$ denote the $k \times (n-k+1)$ subarray of A_k consisting of rows 1 through k and columns $k+1$ through $n+1$ of A_k, so that $b_k[s,t] = a_k[s,t+k]$. (One such plane A_k and its subarray B_k are depicted in Figure 7.6.) Finally, for $1 \le k < j \le n+1$, let

$$F(j,k) = \min_{1 \le i \le n} a[i,j,k] .$$

$F(k+1,k), F(k+2,k), \ldots, F(n+1,k)$ are simply the column minima of B_k, and for $2 \le j \le n+1$,

$$E(j) = \min_{1 \le k < j} \{F(j,k)\} .$$

Our algorithm consists of n stages, each requiring $O(n)$ time. In the kth stage, we compute $F(k+1,k), F(k+2,k), \ldots, F(n+1,k)$ and then $E(k+1)$. For computing $E(k+1)$, $O(n)$ time clearly suffices, since $E(k+1)$ depends only on $F(k+1,1), F(k+1,2), \ldots, F(k+1,k)$ and we have already computed these values. Thus, all that remains to be shown is that we can compute the column minima of B_k in $O(n)$ time given $E(1), \ldots, E(k)$. For such an argument, we need the following two lemmas.

Lemma 7.10 B_k is inverse-Monge for all k in the range $1 \leq k \leq n$.

Proof Consider any entry $b_k[s,t] = a_k[s,t+k]$ of B_k. Since $s \leq k$ and $t+k > k$, this entry is finite. In particular,

$$b_k[s,t] \;=\; E(s) + c_k(d_{s,t+k}) + \sum_{m=s}^{k-1} g_m(d_{s,m+1}) + \sum_{m=k+1}^{t+k-1} h_m(d_{m,t+k}) \;.$$

Now observe that the terms $E(s)$ and $\sum_{m=s}^{k-1} g_m(d_{s,m+1})$ in the above depend only on s, and the term $\sum_{m=k+1}^{t+k-1} h_m(d_{m,t+k})$ depends only on t. Furthermore, $c_k(\cdot)$ is a concave function, $d_{s,t+k} = d_{1,t+k} - d_{1,s}$, and $0 = d_{1,1} \leq d_{1,2} \leq \cdots \leq d_{1,n+1}$. Thus, by Properties 1.6, 1.5, and 1.7, B_k is inverse-Monge. ■

Lemma 7.11 Given $O(n^2)$ preprocessing time, we can compute any entry of B_k in constant time for all k in the range $1 \leq k \leq n$.

Proof As we observed in the proof of the previous lemma,

$$b_k[s,t] \;=\; E(s) + c_k(d_{s,t+k}) + \sum_{m=s}^{k-1} g_m(d_{s,m+1}) + \sum_{m=k+1}^{t+k-1} h_m(d_{m,t+k})$$

for all s in the range $1 \leq s \leq k$ and all t in the range $1 \leq t \leq n-k+1$. Now suppose we precompute $d_{1,i}$ for all i in the range $1 \leq i \leq n$, which takes $O(n)$ time. This preprocessing gives us any $d_{i,j}$ in constant time, since $d_{i,j} = d_{1,j} - d_{1,i}$. Suppose we also precompute

$$\sum_{m=i}^{k-1} g_m(d_{i,m+1})$$

for all i and k satisfying $1 \le i \le k \le n$ and

$$\sum_{m=k+1}^{j-1} h_m(d_{m,j})$$

for all k and j satisfying $1 \le k < j \le n+1$. This preprocessing takes an additional $O(n^2)$ time, since

$$\sum_{m=i}^{k-1} g_m(d_{i,m+1}) = g_{k-1}(d_{i,k}) + \sum_{m=i}^{k-2} g_m(d_{i,m+1})$$

and

$$\sum_{m=k+1}^{j-1} h_m(d_{m,j}) = h_{k+1}(d_{k+1,j}) + \sum_{m=k+2}^{j-1} h_m(d_{m,j}) .$$

These precomputations allow us to compute $b_k[s,t]$ in constant time, since by the time we consider B_k, $E(s)$ is known for all $s \le k$. ■

Lemmas 7.10 and 7.11 allow us to apply the off-line SMAWK algorithm of Aggarwal et al. to obtain the column minima of B_k from $E(1), \ldots, E(k)$ in $O(n)$ time. Thus, we have shown that each stage of our algorithm requires only $O(n)$ time, which gives the entire algorithm a running time of $O(n^2)$, including the preprocessing time required for Lemma 7.11.

Theorem 7.12 Given an n-period instance of the backlogging economic lot-size problem such that the $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$ are concave functions, we can find an optimal production schedule in $O(n^2)$ time. ■

7.4.3 Other Cost Structures

In [Zan69], Zangwill considered yet another cost structure for the backlogging economic lot-size problem: he assumed that for $1 \le i \le n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

where c_i^0 and c^1 are constants and $c_i^0 \ge 0$, and that the $h_i(\cdot)$ and $g_i(\cdot)$ are nondecreasing concave functions. (Note that the marginal cost of production c^1 is the same for all time periods; this assumption is again needed to insure that the arrays considered below are Monge.) If we

consider the $n \times (n+1) \times n$ array $A = \{a[i,j,k]\}$ where

$$a[i,j,k] = \begin{cases} E(i) + c_k^0 + c^1 d_{i,j} + \sum_{m=i}^{k-1} g_m(d_{i,m+1}) + \sum_{m=k+1}^{j-1} h_m(d_{m,j}) & \text{if } i \le k < j, \\ +\infty & \text{otherwise.} \end{cases}$$

then A can be decomposed into two two-dimensional Monge arrays S and T as in Subsubsection 7.4.1. These arrays are Monge because

1. $s[i,k]+s[i+1,k+1]-s[i,k+1]-s[i+1,k] = g_k(d_{i+1,k+1}) - g_k(d_{i,k+1})$ for $1 < i+1 < k < n$, and the right-hand side of this equation is nonpositive so long as $g_k(\cdot)$ is a nondecreasing function, and

2. $t[k,j]+t[k+1,j+1]-t[k,j+1]-t[k+1,j] = h_{k+1}(d_{k+1,j}) - h_{k+1}(d_{k+1,j+1})$ for $1 < k+1 < j < n+1$, and the right-hand side of this equation is nonpositive so long as $h_{i+1}(\cdot)$ is a nondecreasing function.

However, it is unclear how to compute $s[i,k]$ and $t[k,j]$ in constant time given $o(n^2)$-time preprocessing; thus, we are unable to improve the running time of Zangwill's $O(n^2)$-time algorithm for the problem.

As a final remark, suppose that for all i and k such that $1 \le i < k \le n$, we knew

$$\sum_{m=i}^{k-1} g_m(d_{i,m+1}) ,$$

and similarly, for all k and j such that $1 \le k < j \le n+1$, we knew

$$\sum_{m=k+1}^{j-1} h_m(d_{m,j}) .$$

In this case, it is easy to see that any entry in row i of S and T could then be computed in constant time, given the minimum entries in columns 1 through i of S and T. Consequently, the column minima of S and T could then be computed in linear additional time using the approach of Subsubsection 7.4.1. We will use this observation in Subsection 7.5.1, as it helps us to obtain an improved algorithm for the periodic variant of the backlogging economic lot-size problem considered by Erickson, Monma and Veinott [EMV87].

7.5 Two Periodic Problems

In this section, we present algorithms for two periodic variants of the backlogging economic lot-size problem. The first was proposed by Erickson, Monma, and Veinott in [EMV87], whereas the second was given by Graves and Orlin in [GO85]. Both problems assume that the planning horizon is infinite (i.e., we are planning for an infinite number of periods) but that demands and costs vary periodically over time with period n, so that

$$\begin{aligned} d_{i+rn} &= d_i\ , \\ c_{i+rn}(\cdot) &= c_i(\cdot)\ , \\ h_{i+rn}(\cdot) &= g_i(\cdot)\ , \text{ and} \\ g_{i+rn}(\cdot) &= h_i(\cdot)\ , \end{aligned}$$

for $1 \leq i \leq n$ and all positive integers r. Erickson et al. consider the problem of finding a production schedule of period n with minimum total cost, whereas Graves and Orlin tackle the more difficult problem of finding a semi-infinite production schedule (starting with period 1, where the initial inventory is assumed to be 0) with minimum average cost per period.

7.5.1 Erickson, Monma, and Veinott's Problem

Given an infinite planning horizon and periodic demands and costs, Erickson, Monma, and Veinott [EMV87] considered the problem of finding an infinite production schedule with minimum average cost per period, subject to the restriction that the production schedule must have period n, i.e., we must have $x_{i+rn} = x_i$ and $y_{i+rn} = y_i$ for $1 \leq i \leq n$ and all positive integers r. This problem is equivalent to finding the minimum-cost n-period production schedule for periods i through $n+i-1$, where i is allowed to vary between 1 and n. In terms of network flows, this new problem is obtained from the backlogging economic lot-size problem by adding two edges to the graph depicted in Figure 7.2, one from the nth sink to the first sink with concave cost function $h_1(\cdot)$ and the second from the first sink to the nth sink with concave cost function $g_n(\cdot)$.

For arbitrary concave costs $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$, Erickson et al. gave an $O(n^3)$-time algo-

rithm for their problem, which they obtained by solving n instances of the n-period backlogging economic lot-size problem. For the special case where

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

and the $g_i(\cdot)$ and $h_i(\cdot)$ are nondecreasing, we can improve this bound to $O(n^2)$ time using the techniques of Subsection 7.4.3: we merely spend $O(n^2)$ time to precompute

$$\sum_{m=i}^{k-1} g_m(d_{i,m+1})$$

for all i and k such that $1 \leq i < k \leq 2n$ and

$$\sum_{m=k+1}^{j-1} h_m(d_{m,j})$$

for all k and j such that $1 \leq k < j \leq 2n$, and then solve n instances of the n-period backlogging economic lot-size problem in $O(n)$ time each.

Theorem 7.13 Given an instance of Erickson, Monma, and Veinott's economic lot-size problem with periodicity n such that

- for $1 \leq i \leq n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c^1 x & \text{if } x > 0, \end{cases}$$

  where c_i^0 and c^1 are constants and $c_i^0 \geq 0$, and

- for $1 \leq i \leq n$, $g_i(\cdot)$ and $h_i(\cdot)$ are nondecreasing concave functions,

we can find an optimal infinite production schedule with period n in $O(n^2)$ time. ■

7.5.2 Graves and Orlin's Problem

In [GO85], Graves and Orlin consider another periodic variant of the backlogging economic lot-size problem. They assume demands and costs are periodic, as do Erickson, Monma, and

Veinott, and seek an infinite production schedule (starting in period 1, where the initial inventory is assumed to be 0) with minimum average cost per period. Unlike Erickson et al., however, they do not restrict the production schedule to have period n. Instead, they assume

$$\lim_{y \to \infty} G(y) = \lim_{y \to \infty} H(y) = \infty ,$$

where

$$G(y) = g_1(y) + g_2(y) + \cdots + g_n(y)$$

and

$$H(y) = h_1(y) + h_2(y) + \cdots + h_n(y) .$$

This assumption allows them to prove the following lemma.

Lemma 7.14 (Graves and Orlin [GO85]) Let

$$C = c_1(d_1) + c_2(d_2) + \cdots + c_n(d_n) ,$$

and let

$$D = d_1 + d_2 + \cdots + d_n .$$

Furthermore, let p denote the minimum integer such that $C < G(pD)$ and $C < H(pD)$. (Such a p exists by our assumption about the unboundedness of $G(\cdot)$ and $H(\cdot)$.) There is an optimal production schedule (i.e., a production schedule of minimum average cost per period) such that every interval of $2(p+1)n$ consecutive periods contains at least one period with nonzero production. ■

This lemma and Theorem 7.1 (which also applies to infinite graphs; see [GO85]) together imply every production schedule must repeat after at most $2(p+1)n^2$ periods, since an optimal production schedule starting from period i must be an optimal production schedule starting from period $i + rn$, for all integers r. Thus, there exists an optimal semi-infinite production schedule consisting of a finite production schedule with length at most $2(p+1)n^2$ followed by an infinite periodic production schedule with period at most $2(p+1)n^2$. In other words, there exist integers n_1 and n_2, both between 1 and $2(p+1)n^2$, such that the optimal production schedule

for periods 1 through n_1 (with no initial or final inventory and no initial or final backlogged demand) and the optimal production schedule for periods $n_1 + 1$ through $n_1 + n_2$ (again with no initial or final inventory or backlogging) together characterize the optimal infinite production schedule.

In [GO85], Graves and Orlin argued that such an optimal semi-infinite production schedule can be computed in $O(p^3n^3)$ time. We reduce this bound to $O(p^2n^3)$ using Monge arrays.

To obtain a faster algorithm, we first compute an optimal production schedule for periods 1 through j, where j is allowed to vary from 1 to $2(p+1)n^2$ and both the initial and final inventory and backlogging are required to be 0. Such a schedule can be computed in $O(p^2n^4)$ time by applying the techniques of Subsection 7.4.2 directly, i.e., by computing the plane minima of an $O(pn^2) \times O(pn^2) \times O(pn^2)$ Monge array A. However, we can reduce this bound to $O(p^2n^3)$ time if we make use of Lemma 7.14. Specifically, since production in period j implies production in some period between $j - 2(p+1)n$ and $j - 1$, we need only consider those entries $a[i, j, k]$ of A such that $j - 2(p+1)n \leq i \leq k < j$. Roughly speaking, we can distribute these entries among $O(n)$ Monge arrays of size $O(pn) \times O(pn) \times O(pn)$ whose plane minima can be computed in $O(p^2n^2)$ time each.

Once we have an optimal production schedule for periods 1 through j for all j between 1 to $2(p+1)n^2$, we can find the optimal infinite schedule as follows. For $1 \leq j \leq 2(p+1)n^2$, we can identify the periodic and nonperiodic portions of the optimal production schedule for periods 1 through j and compute each portion's average cost per period in $O(pn)$ time per value of j, i.e., $O(p^2n^3)$ total time. Then, in $O(pn^2)$ additional time, we can select the value of j giving the best infinite production schedule, which gives us the following theorem.

Theorem 7.15 Given an instance of Graves and Orlin's economic lot-size problem with periodicity n, such that the $c_i(\cdot)$, $g_i(\cdot)$, and $h_i(\cdot)$ are concave functions and p is defined as above, we can find an optimal semi-infinite production schedule in $O(p^2n^3)$ time. ∎

We should remark here that since p depends upon the production, inventory, and backlogging costs and may be exponential in n, both Graves and Orlin's algorithm and our algorithm are weakly polynomial; in fact, obtaining a strongly-polynomial algorithm for this problem remains open.

7.6 Some Final Remarks

In this chapter, we presented efficient dynamic-programming algorithms for several variants of the economic lot-size problem. These algorithms use properties of Monge arrays to improve the running times of previous algorithms, typically by factors of n and $n/\lg n$, where n denotes the number of periods under consideration. Aside from providing faster algorithms for economic lot-size problems, a major contribution of this chapter is the identification of the Monge arrays that arise in connection with economic lot-size problems.

The algorithms given in this chapter can be extended to many other problems related to economic lot-size models. For example, in his paper on Leontif Substitution systems [Vei69], Veinott showed that several other problems (including the product-assortment problem, the batch-queuing problem, the investment-consumption problem, and the reservoir-control problem) can be transformed into economic lot-size problems (with or without backlogging).

Another model related to the economic lot-size model that deserves special mention is the *capacity-expansion* model proposed by Manne and Veinott in [MV67]. This model was developed during Manne's study of four major industries in India between 1950 and 1965 [Man67], and many researchers have studied problems formulated in terms of this model since then. (See, for example, [FR75, Lus79, Lus82, Lus86, LL87].) In [MV67], Manne and Veinott gave an $O(n^3)$-time algorithm for computing an optimal, feasible plan in their capacity-expansion model. Since their dynamic-programming algorithm is identical to Zangwill's $O(n^3)$-time algorithm for solving the backlogging economic lot-size problem with concave costs, the techniques of Subsection 7.4.2 yield an $O(n^2)$-time algorithm for their problem. In a similar vein, several of the algorithms given in Sections 7.3 and 7.4 can be used to speed up various algorithms given by Luss in [Lus82].

On a different note, this chapter considered only production systems involving a single type of item and a single stage of production. However, other researchers (see [Gra82, Lus82, Lus86], for example) have shown that the problem of computing an optimal plan for a multi-item and/or multi-stage production system can usually be decomposed into simpler problems using Lagrangian relaxation methods or simple heuristics that work fairly well in practice. Furthermore, these resulting problems can be expressed as economic lot-size problems. The only difference between the economic lot-size problems considered in this chapter and those

that result when computing optimal production schedules for such complex production systems is that this chapter assumes demands are always nonnegative, whereas in the economic lot-size problems resulting from Lagrangian relaxation methods or heuristics, the demands may be negative in certain situations. In other words, some of the demand nodes may in fact be supply nodes; this supply has no cost, but it must be used up by any feasible production plan. Now when demands are negative, the arrays that occur in Sections 7.3–7.5 are not always Monge. Nevertheless, Aggarwal and Park [AP90] have shown that such arrays are often Monge-like. Consequently, the basic paradigm developed in this chapter can still be applied when the demands are negative, and the time complexities of the resulting algorithms are quite similar to those given in this chapter.

In this chapter's introduction, we mentioned some recent work by Federgruen and Tzur [FT89, FT90] and by Wagelmans, van Hoesel, and Kolen [WvHK89], who have independently obtained several of the results that we present in this chapter. We will now relate their work to our own.

In [WvHK89], Wagelmans, van Hoesel, and Kolen present an $O(n \lg n)$-time algorithm for the basic economic lot-size problem with nearly linear costs. This matches the time bound of our algorithm for this problem, which we describe in Subsubsection 7.3.1. Wagelmans et al. also give an $O(n)$-time algorithm for the special case of the basic economic lot-size problem with nearly linear costs that we consider in Subsubsection 7.3.1. (For this special case, we assume $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$, i.e., the marginal cost of producing in period i is at most the marginal cost of producing in period $i-1$ plus the marginal cost of storing inventory from period $i-1$ to period i; this cost structure subsumes those considered by Manne [Man58] and by Wagner and Whitin [WW58].) This result again matches the time bound of our algorithm for the problem.

These same two results — an $O(n \lg n)$-time algorithm for the basic economic lot-size problem with nearly linear costs and a linear-time algorithm for the special case of Subsubsection 7.3.1 — are independently derived by Federgruen and Tzur in [FT89]. Moreover, Federgruen and Tzur also give an $O(n)$-time algorithm for the basic economic lot-size problem with nearly linear costs when setup costs are nondecreasing, i.e., $c_1^0 \leq c_2^0 \leq \cdots \leq c_n^0$ in the notation of Subsection 7.3.1. Furthermore, in [FT90], Federgruen and Tzur give an $O(n \lg n)$-time al-

gorithm for the backlogging economic lot-size problem with nearly linear costs, matching the result we give in Subsubsection 7.4.1. They also given a linear-time algorithm for the special case of Subsubsection 7.4.1 (again matching our result for this problem), as well as some additional special cases.

Both Federgruen and Tzur and Wagelmans, van Hoesel, and Kolen use essentially the same techniques to obtain their results, and these techniques are substantially different from our own. Roughly speaking, they are computing (in an on-line fashion) the convex hull of n points in an appropriate two-dimensional space, whereas we are searching in Monge arrays. Note that neither Federgruen and Tzur nor Wagelmans, van Hoesel, and Kolen are able to obtain results for the general backlogging economic lot-size problem with arbitrary concave costs comparable to the results that we present in Subsection 7.4.2, which suggests that our techniques are in some sense more general.

We conclude with a list of open problems:

1. In Subsubsection 7.3.1, we gave an $O(n \lg n)$-time algorithm for the basic economic lot-size problem when the production and inventory costs are nearly linear, and in Subsubsection 7.4.1, we gave an $O(n \lg n)$-time algorithm for the backlogging economic lot-size problem when the production, inventory, and backlogging costs are nearly linear. It remains open whether there exists a $o(n \lg n)$-time algorithm for either of these problems.

2. Veinott [Vei63] showed that Wagner and Whitin's algorithm for the basic economic lot-size problem can be used even when the production and inventory cost functions are arbitrary concave functions and that the resulting algorithm still takes $O(n^2)$ time. In this chapter, we were unable to improve upon this bound (see Table 7.1); thus, obtaining better time bounds for the basic problem with concave costs remains a challenging open problem. In fact, as pointed out in Subsection 7.3.2, this problem remains open even for concave inventory costs and nearly linear production costs such that $c_i(0) = 0$ and $c_i(x) = c_i^0 + c^1 x$ for $x > 0$, where $c_i^0 \geq 0$. (This latter cost structure may greatly simplify the problem, since here the resulting array is Monge, as observed in Subsection 7.3.2.)

3. In [EMV87], Erickson, Monma, and Veinott gave an $O(n^3)$-time algorithm for a periodic variant of the backlogging economic lot-size problem. We were unable to obtain a faster

algorithm for this problem when the costs are arbitrary concave functions (see Table 7.3); thus, obtaining a subcubic algorithm for this problem remains open.

4. In [GO85], Graves and Orlin gave an $O(p^3n^3)$-time algorithm for another periodic variant of the backlogging economic lot-size problem, and in this chapter, we improved this bound to $O(p^2n^3)$. However, as mentioned in Subsection 7.5.2, the parameter p in these running times may be exponential in n. Consequently, obtaining a strongly-polynomial algorithm for Graves and Orlin's problem remains open. (See [GO85] for more details.)

Chapter 8

Shortest Paths in Grid DAGs

In this chapter, we give efficient sequential and parallel algorithms for finding shortest paths in certain planar acyclic directed graphs. We then use these algorithms to obtain improved results for string editing and surface reconstruction from planar contours.

The directed graphs that we consider in this chapter are what Apostolico, Atallah, Larmore, and McFaddin [AALM90] refer to as *grid DAGs*. An $m \times n$ grid DAG $G = (V, A)$ is defined as follows:

$$V = \{v_{i,j} : 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$$

and

$$A = A_H \cup A_V \cup A_D \ ,$$

where

$$A_H = \{(v_{i,j}, v_{i,j+1}) : 1 \leq i \leq m \text{ and } 1 \leq j < n\} \ ,$$

$$A_V = \{(v_{i,j}, v_{i+1,j}) : 1 \leq i < m \text{ and } 1 \leq j \leq n\} \ ,$$

and

$$A_D = \{(v_{i,j}, v_{i+1,j+1}) : 1 \leq i < m \text{ and } 1 \leq j < n\} \ .$$

For example, Figure 8.1 depicts an 8×13 grid DAG. (In reference to their orientations in Figure 8.1, the arcs of A_H, A_V, and A_D are known as horizontal, vertical, and diagonal arcs, respectively.) Associated with each arc $a \in A$ is a cost $c(a)$.

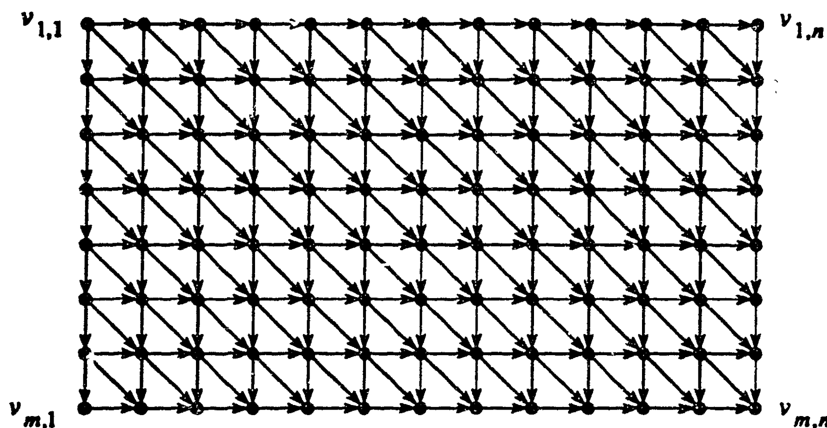

Figure 8.1: An 8×13 grid DAG with vertices $v_{1,1}$, $v_{1,n}$, $v_{m,1}$, and $v_{m,n}$ labeled.

In [FKU77], Fuchs, Kedem, and Uselton reduced the problem of optimal surface reconstruction from planar contours to a shortest paths problem in a grid DAG. In a separate paper, Kedem and Fuchs [KF80] reduced the string editing problem considered in [WF74] to another grid DAG shortest paths problem. They also reduced the circular string-to-string correction problem to the shortest paths problem considered in [FKU77]. More recently, Apostolico, Atallah, Larmore, and McFaddin [AALM90] and Mathies [Mat88] have independently provided parallel algorithms for the string editing and largest common subsequences problems that make use of these reductions.

This chapter is organized as follows. In Section 8.1, we describe a divide-and-conquer algorithm solving the two shortest paths problems mentioned above. Then in Sections 8.2 and 8.3 we apply our shortest paths algorithm to several different problems and compare our results with those given in the literature. The results presented in this chapter represent joint work with Alok Aggarwal from [AP89a].

8.1 A Shortest-Paths Algorithm

Let G be an $m \times n$ grid DAG. We will call vertex $v_{i,j}$ a *source* if either $i = 0$ or $j = 0$ and a *sink* if either $i = m - 1$ or $j = n - 1$. If we orient the arcs of G as in Figure 8.1, then the sources of G are those vertices on its left and bottom boundaries and the sinks are those vertices on its top and right boundaries. Keeping Figure 8.1 in mind, we let $s_0, \ldots, s_{m+n-2}$ denote the sources

of G in counterclockwise order and let $t_0, \ldots, t_{m+n-2}$ denote the sinks of G in clockwise order, so that

$$s_i = \begin{cases} v_{m-1-i,0} & \text{if } 0 \leq i \leq m-1, \\ v_{0,i-m+1} & \text{if } m \leq i \leq m+n-2, \end{cases}$$

and

$$t_j = \begin{cases} v_{m-1,j} & \text{if } 0 \leq j \leq n-1, \\ v_{m+n-2-j,n-1} & \text{if } n \leq j \leq m+n-2. \end{cases}$$

In this section, we consider the problem of computing all $(m+n-1)^2$ source-to-sink shortest paths. In the context of array-searching, this is equivalent to computing all the entries of the $(2n-1) \times (2n-1)$ *distance array* DIST_G, whose row i, column j entry is the length of the shortest path from s_i to t_j.

We begin with the special case of $m = n$, i.e., G is an $n \times n$ grid DAG. We also assume for simplicity that $n = 2^r - 1$ for some positive integer r.

Our algorithm takes the natural divide-and-conquer approach. We divide the $(2^r - 1) \times (2^r - 1)$ grid DAG G into four $(2^{r-1} - 1) \times (2^{r-1} - 1)$ grid DAGs A, B, C, and D, as shown in Figure 8.2. Then, in parallel, we recursively compute all source-to-sink shortest paths for A, B, C, and D, thereby obtaining the four distance arrays DIST_A, DIST_B, DIST_C, and DIST_D. Finally, we perform the following three "merge" steps.

1. We compute $\text{DIST}_{A\cup B}$ from DIST_A and DIST_B ($A \cup B$ is the grid DAG obtained by recombining A and B).

2. We compute $\text{DIST}_{C\cup D}$ from DIST_C and DIST_D.

3. We compute DIST_G from $\text{DIST}_{A\cup B}$ and $\text{DIST}_{C\cup D}$.

The algorithm we have outlined so far is identical to that given by Apostolico et al. in [AALM90]; our algorithms differ only in the implementation of the three "merge" steps listed above. Apostolico et al. solve each of these three subproblems directly, whereas we reduce each of these subproblems to the tube-maxima problem for a particular $\Theta(n) \times \Theta(n) \times \Theta(n)$ three-dimensional totally monotone array. This allows us to use the array-searching algorithms we develop in Section 4.4 and improve upon the results given in [AALM90].

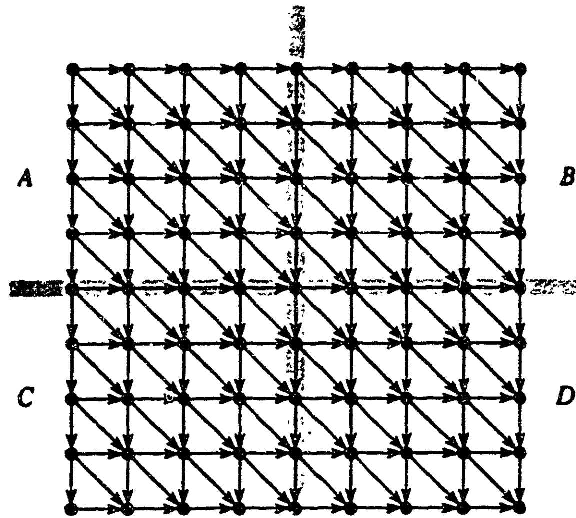

Figure 8.2: The partitioning scheme used by our divide-and-conquer solution to the shortest paths problem for a grid DAG.

In order to show the three-dimensional arrays we consider in connection with the "merge" steps described above are totally monotone, we need the following lemma. (Similar lemmas are given in [FKU77] and [AALM90] — this is essentially the basic observation made by G. Monge [Mon81] in 1781.)

Lemma 8.1 For any grid DAG G, DIST_G satisfies the Monge condition.

Proof Consider any 2×2 minor of DIST_G formed by rows i and k and columns j and l, where $i < k$ and $j < l$. Given our ordering of the sinks and sources of G, every path from s_i to t_l must intersect every path from s_k to t_j. In particular, there must be some vertex v that lies on both the shortest path from s_i to t_l and the shortest path from s_k to t_j, as in Figure 8.3. If we let $\ell(x, y)$ denote the length of the shortest path in G from vertex x to vertex y, then

$$\begin{aligned} \ell(s_i, t_l) + \ell(s_k, t_j) &= \ell(s_i, v) + \ell(v, t_l) + \ell(s_k, v) + \ell(v, t_j) \\ &\geq \ell(s_i, t_j) + \ell(s_k, t_l), \end{aligned}$$

which means the 2×2 minor corresponding to rows i and k and columns j and l of DIST_G satisfies the Monge condition. ∎

Figure 8.3: The shortest path from s_i to t_l and the shortest path from s_k to t_j must intersect at some vertex v. Thus, the length of the shortest path from s_i to t_l plus the length of the shortest path from s_k to t_j is at least the length of the shortest path from s_i to t_j plus the length of the shortest path from s_k to t_l.

We now show how to compute $\text{DIST}_{A\cup B}$ from DIST_A and DIST_B. ($\text{DIST}_{C\cup D}$ and DIST_G can be computed in a similar fashion.) Consider the $(2^r + 2^{r-1} - 1) \times (2^r + 2^{r-1} - 1)$ distance array $\text{DIST}_{A\cup B}$. The sources of $A \cup B$ can be divided into two types: those that are sources of A and those that are sources of B. Similarly, the sinks of $A \cup B$ can be separated into those that are sinks of A and those that are sinks of B. This divides the array $\text{DIST}_{A\cup B}$ into four different regions, as is suggested in Figure 8.4. One region corresponds to shortest paths from sources of A to sinks of A. The entries in this region are known, since they are entries of DIST_A. A second region corresponds to shortest paths from sources of B to sinks of B; the entries in this region are also known, since they are entries of DIST_B. A third region corresponds to shortest paths from sources of B to sinks of A. As there are no paths from any of these sources to any of these sinks, all the entries in this region are ∞. Finally, there is a $(2^{r-1} - 1) \times (2^{r-1} - 1)$ region of $\text{DIST}_{A\cup B}$ corresponding to shortest paths from sources of A to sinks of B, whose entries remain to be computed.

To compute the entries in this last region, we make use of the remaining entries in DIST_A

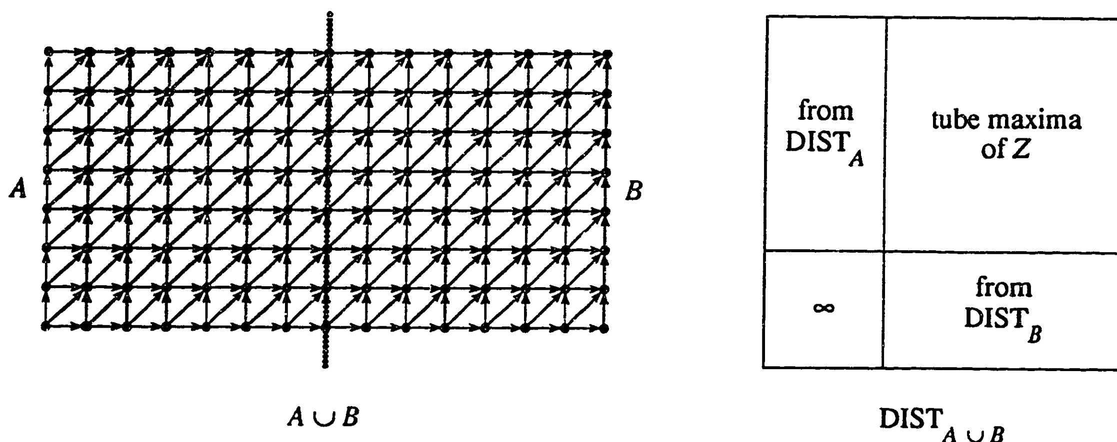

Figure 8.4: The distance array for $A \cup B$ is composed of entries from DIST_A, entries from DIST_B, ∞ entries, and entries corresponding to the tube maxima of Z.

and DIST_B. Specifically, let $X = \{x_{i,j}\}$ denote the $(2^{r-1} - 1) \times (2^{r-2} - 1)$ array obtained by deleting the first $2^{r-2} - 1$ columns of DIST_A. (If the arcs of $A \cup B$ are oriented as in Figure 8.4, then X corresponds to shortest paths from vertices on the left and bottom boundaries of A to vertices on its right boundary.) Similarly, let $Y = \{y_{i,j}\}$ denote the $(2^{r-2} - 1) \times (2^{r-1} - 1)$ array obtained by deleting the last $2^{r-2} - 1$ rows of B. (Again referring to Figure 8.4, the entries of Y are the lengths of shortest paths from vertices on the left boundary of B to vertices on its right and top boundaries.) Now consider the $(2^{r-1} - 1) \times (2^{r-2} - 1) \times (2^{r-1} - 1)$ three-dimensional array $Z = \{z_{i,j,k}\}$, where $z_{i,j,k} = x_{i,j} + y_{j,k}$.

Lemma 8.2 Z satisfies the Monge condition.

Proof Consider any two-dimensional plane of Z. This plane is either the sum of a two-dimensional array satisfying the Monge condition and a one-dimensional vector or the sum of two one-dimensional vectors. Such sums always satisfy the Monge condition. ■

Now the tube maximum of Z are precisely the remaining entries of $\mathrm{DIST}_{A \cup B}$. (The length of the shortest path from a source s of A to a sink t of B is just the minimum over all vertices v on the boundary shared by A and B of the length of the shortest path from s to v plus the length of the shortest path from v to t.) Thus, we can use the tube maxima algorithms of Section 4.4 to complete our computation of $\mathrm{DIST}_{A \cup B}$. Putting it all together, we obtain the following theorem.

Theorem 8.3 All source-to-sink shortest paths in an $n \times n$ grid DAG can be computed

1. in $O(\lg^2 n)$ time using $n^2/\lg n$ processors on a CREW-PRAM, and
2. in $O(\lg\lg n \lg n)$ time using $n^2/\lg\lg n$ processors on a CRCW-PRAM.

Proof Let $T(n)$ and $P(n)$ denote the time and number of processors, respectively, required to compute the lengths of all source-to-sink shortest paths in an $n \times n$ grid DAG. From Lemma 8.2 and the algorithm given above, it follows that

$$T(2^r - 1) \leq T(2^{r-1} - 1) + O(T'(2^r - 1))$$

and

$$P(2^r - 1) \leq \max\{4P(2^{r-1} - 1), O(P'(2^r - 1))\},$$

where $T(2)$ and $P(2)$ are constants and $T'(n)$ and $P'(n)$ denote the time and number of processors, respectively, required to find the tube maxima of an $n \times n \times n$ totally monotone array.

Now from Theorem 4.17, we have $T'(n) = O(\lg n)$ and $P'(n) = O(n^2/\lg n)$ for a CREW-PRAM. Consequently, $T(n) = O(\lg^2 n)$ and $P(n) = O(n^2)$. Furthermore, we can reduce the number of processor required by invoking Brent's theorem [Bre74]. If $W(n)$ denotes the number of operations required to compute the lengths of all source-to-sink shortest paths in an $n \times n$ grid DAG, and if $W'(n)$ denotes the number of operations required to compute the tube maxima of an $n \times n \times n$ totally monotone array, then $W(n) \leq 4W(n) + W'(n)$ and $W(2)$ is a constant. From Theorem 4.17, we have $W'(n) = O(n^2)$ for a CREW-PRAM; this means $W(n) = O(n^2 \lg n)$. Thus, by Brent's theorem, the number of processors required is $O(W(n)/T(n)) = O(n^2/\lg n)$.

In similar fashion, we can obtain the CRCW-PRAM result using Theorems 4.18. ■

Above, we showed how to compute all source-to-sink shortest paths in an $n \times n$ grid DAG; these results extend naturally to general n and m. However, in the applications of Section 8.3, we actually only need to compute a subset of the source-to-sink shortest paths. Specifically, it is sufficient to compute the shortest path from $v_{i,0}$ to $v_{j,n-1}$ for all i and j between 1 and m, i.e., we only have to consider sources on the left boundary of G and sinks on its right boundary. By restricting our attention to these shortest paths, we can obtain better results when m is much smaller than n.

Corollary 8.4 Let G denote an $m \times n$ grid DAG where $m \leq n$. We can compute the length of the shortest path from $v_{i,0}$ to $v_{j,n-1}$, for all i and j between 1 and m,

1. in $O(\lg m \lg n)$ time using $mn/\lg m$ processors on a CREW-PRAM, and
2. in $O(\lg\lg m \lg n)$ time using $mn/\lg\lg m$ processors on a CRCW-PRAM.

Proof We use essentially the same divide-and-conquer approach taken for the $n = m$ case. Specifically, we partition the grid DAG G into two $m \times (\lfloor n/2 \rfloor + 1)$ grid DAGs X and Y that share a common m-vertex boundary. We then recursively compute (in parallel) all shortest paths from the left boundary of X to its right boundary and all shortest paths from the left boundary of Y to its right boundary. Finally, we compute all shortest paths from the left boundary of G to its right boundary by solving the tube-maxima problem for a $\Theta(m) \times \Theta(m) \times \Theta(m)$ three-dimensional totally monotone array whose entries are sums of lengths of shortest paths in X and Y. (This last step is almost identical to the "merge steps" of our all source-to-sink shortest paths algorithm).

Now if $T(m,n)$ and $P(m,n)$ denote the time and number of processors, respectively, required to solve the shortest paths problem in question, and if $T'(m)$ and $P'(m)$ denote the time and number of processors, respectively, required to find the tube maxima of an $n \times n \times n$ totally monotone array, then

$$T(m,n) \leq 2T(m, \lfloor n/2 \rfloor + 1) + O(T'(m))$$

and

$$P(m,n) \leq \max\{2P(m, \lfloor n/2 \rfloor + 1), O(P'(m))\}.$$

Using Theorems 4.17 and 4.18 to bound $T'(m)$ and $P'(m)$ and Theorem 8.3 to bound $T(m,m)$ and $P(m,m)$, we obtain the specified time and processor bounds. ■

The following two questions remain unresolved regarding this last shortest paths problem. First, the best sequential algorithm for computing these shortest paths (given in [FKU77]) requires $O(mn \lg m)$ time, whereas the only known lower bound is $\Omega(mn)$. It seems that the array-searching framework should be useful in improving the upper bound. Second, the processor-time product of the results we give in Corollary 8.4 is a factor of $\lg n/\lg m$ away from the sequential time bound.

8.2 String Editing and Related Problems

The *string editing* problem (also called the *string-to-string correction* problem) for input strings $x = x_1x_2 \dots x_s$ and $y = y_1y_2 \dots y_t$, $s = |x|$ and $t = |y|$, is that of finding a sequence of *edit operations* transforming x to y, such that the sum of the individual edit operations' costs is minimized. Three different types of edit operations are allowed: we can delete the symbol x_i at cost $D(x_i)$, insert the symbol y_j at cost $I(y_j)$, or substitute the symbol x_i for the symbol y_j at cost $S(x_i, y_j)$. In [WF74], Wagner and Fischer gave an $O(st)$ time sequential algorithm for this problem.

The *circular string-to-string correction* problem is that of transforming x to y when an initial cyclic shift of x is allowed (at no cost) before any editing takes place. (In other words, we minimize the cost of transforming $x_ix_{i+1} \dots x_sx_1x_2 \dots x_{i-1}$ to y over all i.) The best sequential algorithm known for this problem is the $O(mn \lg m)$ time algorithm given by Kedem and Fuchs in [KF80], where $m = \min\{|x|, |y|\}$ and $n = \max\{|x|, |y|\}$.

In obtaining their result for the circular string-to-string correction problem, Kedem and Fuchs reduce this problem to a shortest paths problem in a grid DAG. Given strings x and y, they construct a $2m \times (n+1)$ grid DAG G, $m = \min\{|x|, |y|\}$ and $n = \max\{|x|, |y|\}$, such that the minimum cost edit sequences transforming cyclic shifts of x to y are given by the shortest paths in G from $v_{i,0}$ to $v_{i+m,n}$ for all i between 0 and $m-1$. (In particular, the shortest path from $v_{0,0}$ to $v_{m,n}$ corresponds to the minimum cost edit sequence transforming x to y.)

Using this reduction, Mathies [Mat88] and Apostolico, Atallah, Larmore, and McFaddin [AALM90] independently obtained parallel algorithms for the string editing problem. Specifically, Mathies gave an $O(\lg m \lg n)$-time, mn-processor CRCW-PRAM algorithm for the problem, and Apostolico et al. gave an $O(\lg^2 m \lg n)$-time, $(mn/\lg^2 m)$-processor CREW-PRAM algorithm and an $O(\lg m \lg\lg m \lg n)$-time, $(mn/\lg m \lg\lg m)$-processor CRCW-PRAM algorithm.

Since the shortest paths that must be computed in Kedem and Fuchs' reduction are a subset of shortest paths covered by Corollary 8.4, we can apply this corollary and obtain the following theorem.

Theorem 8.5 Both the string editing problem and the more general circular string-to-string

correction problem for strings x and y can be solved

1. in $O(\lg m \lg n)$ time using $mn/\lg m$ processors on a CREW-PRAM, and
2. in $O(\lg\lg m \lg n)$ time using $mn/\lg\lg m$ processors on a CRCW-PRAM.

where $m = \min\{|x|, |y|\}$ and $n = \max\{|x|, |y|\}$. ■

This improves the time bound for CRCW-PRAMs that Mathies gave in [Mat88] by a factor of $\lg m/(\lg\lg m)^2$ and the time bounds for CREW- and CRCW-PRAMs that Apostolico, Atallah, Larmore, and McFaddin gave in [AALM90] by factors of $\lg m$ and $\lg m/\lg\lg m$, respectively. Furthermore, our results have the same processor-time product as those given in [AALM90], and they improve the processor-time product of the result given in [Mat88] by a factor of $\lg m$. Note that this settles the open question posed at the conclusion of [AALM90] as to whether any improvements in the time complexities of their algorithms were achievable at the expense of only a polylogarithmic factor increase in processor complexity.

Very recently, Apostolico, Atallah, Larmore, and McFaddin [AALM90] have independently obtained an $O(\lg m \lg n)$-time, $(mn/\lg m)$-processor CREW-PRAM algorithm for the string editing problem, thus matching our CREW-PRAM result. They also give an $O((\lg\lg m)^2 \lg n)$-time, $(mn/\lg\lg m)$-processor CRCW-PRAM algorithm for the problem. This equals our time bound but has a processor-time product that is factor of $\lg\lg m$ higher.

8.3 Surface Reconstruction from Planar Contours

In biological research, medical diagnosis and therapy, architecture, and manufacturing design, it is often useful to reconstruct a three-dimensional solid from a set of cross-sectional contours; this reconstruction provides insight into the solid's structure and facilitates its automatic manipulation and analysis. For example, the three-dimensional solid might correspond to a human head and the contours might be those obtained from a CAT ("Computerized Axial Tomography") scan of the head.

In [FKU77], Fuchs, Kedem, and Uselton propose a procedure for reconstructing the surface of a three-dimensional solid from a set of planar contours represented by polygons. Between each pair of consecutive contours, they construct a cylindrical surface from triangular tiles, as in

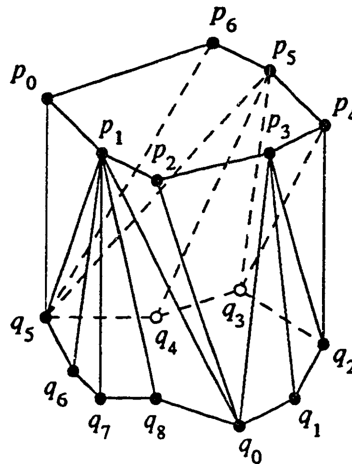

Figure 8.5: The surface between two adjacent planar contours, the first represented by a simple m-gon with vertices $p_0, \ldots, p_{m-1}$ and the second by a simple n-gon with vertices $q_0, \ldots, q_{n-1}$, can be approximated by a sequence of triangular tiles connecting the contours.

Figure 8.5. Given some measure of the appropriateness of a particular triangular tile (its area, for example), Fuchs et al. find the optimal sequence of tiles for a particular pair of contours by solving a shortest paths problem in a grid DAG. In particular, if the two adjacent contours are represented by an m-gon P and an n-gon Q (where we assume without loss of generality that $m \leq n$), Fuchs et al. construct a $(2m + 1) \times (n + 1)$ grid DAG G, such that an optimal sequence of tiles for P and Q is given by the shortest paths in G from $v_{i,0}$ to $v_{m+i,n}$ for all i between 0 and m. They also give an $O(mn \lg m)$ time sequential algorithm for this shortest paths problem.

Since the shortest paths Fuchs et al. compute for the surface reconstruction problem are a subset of shortest paths covered by Corollary 8.4, we can apply the corollary and obtain the following theorem.

Theorem 8.6 The optimal surface reconstruction problem for a pair of contours represented by an m-gon and an n-gon, $m \leq n$, can be solved

1. in $O(\lg m \lg n)$ time using $mn/\lg m$ processors on a CREW-PRAM, and
2. in $O(\lg \lg m \lg n)$ time using $mn/\lg \lg m$ processors on a CRCW-PRAM.

■

Conclusion

This thesis has developed a body of algorithmic techniques and applied them to a wide variety of problems from such diverse areas as computational geometry and operations research. These techniques are centered around a family of highly-structured arrays that we call Monge arrays. We have shown that Monge arrays capture the essential structure of many practical problems, in the sense that algorithms for searching in the abstract world of Monge arrays can be used to obtain efficient algorithms for these practical problems.

We conclude with three open problems:

1. Can the minimum entry in an $n \times n \times n$ three-dimensional Monge array be computed in $O(n)$ time? (The best algorithm currently known for this problem takes $O(n \lg n)$ time.)

2. Can the row minima of an $n \times n$ two-dimensional Monge array be computed in polylogarithmic time with $O(n)$ work? (The best polylogarithmic-time algorithm currently known for this problem takes $O(n \lg n / \lg\lg n)$ work.)

3. Can the row minima of an $n \times n$ two-dimensional partial Monge array of the staircase variety be computed in $O(n)$ time? (The best algorithm currently known for this problem takes $O(n\alpha(n))$ time.)

Appendix A

A Monge-Array Compendium

This appendix gives a comprehensive overview of the Monge-array abstraction and its many applications. This overview takes the form of a list of problems. Each entry in the list consists of a problem name, a problem statement, and a brief summary of known results for the problem. Many of the problems and results contained in the list are not covered in the main body of this thesis; however, if the result is discussed elsewhere in the thesis, a pointer to the appropriate chapter or section is given.

The problems listed in this appendix are divided among six sections. Section A.1 contains basic array-searching problems involving Monge arrays, while Sections A.2 through A.6 cover related problems from other domains, most of which can be transformed into array-searching problems.

A.1 Array-Searching Problems

Row Minimization

Problem Given an $m \times n$ Monge array A, find the minimum entry in each row of A.

Status Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] gave an optimal sequential algorithm for this problem that runs in $O(n)$ time when $m \leq n$ and in $O(n(1 + \lg(m/n)))$ time when $m > n$. This algorithm, which we call the SMAWK algorithm, is presented in Section 2.1 along with an argument for its optimality. The best parallel algorithm for this problem is an $O(\lg m + \lg n)$-time, $(m + n)$-processor EREW-PRAM algorithm due to Atallah and Kosaraju [AK91].

Row Minimization in Partial Monge Arrays

Problem Given an $m \times n$ partial Monge array A, find the minimum entry in each row of A.

Status If A is a staircase array, then its row minima can be computed in $O(n\alpha(m) + m)$ time, where $\alpha(\cdot)$ is a *very* slowly growing inverse of Ackermann's function; the algorithm achieving this result is due to Klawe and Kleitman [KK90]. Though it is not known whether the above time bound is tight, Larmore [Lar90] has used Klawe and Kleitman's algorithm to obtain a potentially improved algorithm for the staircase-array row-minimization problem whose running time is provably optimal

but unknown. Furthermore, Aggarwal, Kravets, J. Park, and Sen [AKPS90] have described an $O(\lg m + \lg n)$-time, $(m+n)$-processor CRCW-PRAM algorithm for computing A's row minima. This algorithm may be transformed into a CREW-PRAM algorithm (with the same time and processor bounds) by incorporating Atallah and Kosaraju's EREW-PRAM algorithm for computing the row minima of a Monge array [AK91] (see the previous entry). As for the other varieties of partial Monge arrays mentioned in Section 1.3, we have the following sequential results. If A is skyline array, then its row minima can again be computed in $O(n\alpha(m)+m)$ time using another extension Klawe and Kleitman's algorithm due to Klawe [Kla89]. However, if A is a v-array or an h-array, then the best result known for computing A's row minima is an $O(n \lg m + m)$-time algorithm due to Aggarwal and Suri [AS90]. Finally, if A is an $n \times n$ v- or h-array whose finite entries are totally monotone but not necessarily Monge, then Klawe [Kla89] has shown an $\Omega(n\alpha(n))$ lower bound on the time needed to find its row minima.

On-Line Row Minimization

Problem Given an $n \times n$ Monge array $A = \{a[i,j]\}$ such that

1. for $i \leq j$, $a[i,j] = +\infty$, and
2. for $i > j$, $a[i,j]$ is available only after the minimum entry in row i has been computed,

find the minimum entry in each row of A.

Status Three groups of researchers independently obtained $O(n)$-time algorithms for this problem: Klawe [Kla89], Galil and K. Park [GP90], and Larmore and Schieber [LS91]. All three algorithms use the SMAWK algorithm, though in different ways. Larmore and Schieber's algorithm is described in Section 2.2. If $a[i,j] + a[k,\ell] \geq a[i,\ell] + a[k,j]$ (rather than $\leq$) for all $i < k < j < \ell$ (i.e., A is a partial inverse-Monge array of the staircase variety), then the row minima of A can be computed in $O(n\alpha(n))$ time using an algorithm due to Klawe and Kleitman [KK90]. Furthermore, Larmore [Lar90] has again proposed a possibly improved algorithm for this problem whose running time is provably optimal but unknown.

Row Selection

Problem Given an $m \times n$ Monge array A and an integer k in the range $1 \leq k \leq n$, find the kth smallest entry in each row of A.

Status The two best results for this problem are an $O(k(m+n))$-time algorithm, due to Kravets and J. Park [KP91], and an $O((\sqrt{m} \lg m)(n \lg n) + m \lg n)$-time algorithm, due to Mansour, J. Park, Schieber, and Sen [MPSS91]. Both these algorithms are described in Section 3.1. Mansour et al. also gave an $O(n \lg m)$-time algorithm for computing an approximate median in each row of A; this approximate median is an entry whose rank in its row lies between $\lfloor n/4 \rfloor$ and $\lceil 3n/4 \rceil - 1$.

Array Selection

Problem Given an $m \times n$ Monge array A and an integer k in the range $1 \leq k \leq mn$, find the kth smallest entry in A.

Status This problem can be solved in $O(m+n+k\lg(mn/k))$ time. The algorithm achieving this time bound, again due to Kravets and J. Park [KP91], is given in Section 3.2.

Row Sorting

Problem Given an $m \times n$ Monge array A, sort the entries in each row of A.

Status Kravets and J. Park [KP91] showed that this problem can be solved in $O(mn)$ time when $m \geq n$ and in $O(mn(1+\lg(n/m)))$ time when $m < n$. Their algorithm is described in Section 3.3.

Array Sorting

Problem Given an $m \times n$ Monge array A, sort the entries of A.

Status Section 3.4 proves an $\Omega(mn\lg(\min\{m,n\}))$ lower bound on the time complexity of this problem when only comparisons between entries are allowed. (This bound is slight generalization of a similar bound for totally monotone arrays given in [KP91].) For $m = \Theta(n)$, the above bound is tight, since the mn entries of A can be sorted in $O(mn\lg mn) = O(mn\lg(\max\{m,n\})$ time without using the Mongeness of A.

Plane Minimization

Problem Given an $n \times n \times \cdots \times n$ d-dimensional Monge array A, find the minimum entry in each plane of A. (A *plane* of A is a $(d-1)$-dimensional subarray corresponding to a fixed value of A's first index.)

Status Aggarwal and J. Park [AP89b] gave an $O(dn\lg^{d-2} n)$-time algorithm for this problem. For the special cases of cycle- and path-decomposable Monge-composite arrays, they also gave $O(dn\lg n)$-time and $O(dn)$-time algorithms, respectively. All three of these algorithms are described in Section 2.3.

Plane Maximization

Problem Given an $n \times n \times \cdots \times n$ d-dimensional Monge array A, find the maximum entry in each plane of A.

Status For three or more dimensions, maximization in Monge arrays is significantly harder than minimization. In particular, Aggarwal and J. Park [AP89b] have proved an $\Omega(n^{d-1}/d)$-time lower bound on the plane maximization problem. They have also given a very simple $O(n^{d-1})$-time algorithm for the problem. Both these results are presented in Section 2.3.

Tube Minimization

Problem Given an $n \times n \times n$ three-dimensional Monge array A, find the minimum entry in each tube of A. (A *tube* of A is a one-dimensional subarray corresponding to fixed values of A's first and third indices.)

Status Apostolico, Atallah, Larmore, and McFaddin [AALM90] and Aggarwal and J. Park [AP89a] independently obtained optimal $O(\lg n)$-time, $(n^2/\lg n)$-processor CREW-PRAM algorithms for this problem. (These algorithms are asymptotically optimal

both in terms of running time and in terms of processor-time product.) Aggarwal and Park's algorithm is described in Section 4.4. The best CRCW-PRAM result for the tube minimization problem is an optimal $O(\lg\lg n)$-time, $(n^2/\lg\lg n)$-processor algorithm discovered by Atallah [Ata90]. Finally, if A is path-decomposable, then its tube minima can be computed in $O(\lg n)$-time on an (n^2)-node hypercube; this last result is due to Aggarwal, Kravets, J. Park, and Sen [AKPS90].

A.2 Geometric Problems

All Farthest/Nearest Neighbors for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, for each vertex v_i, find the vertex v_j farthest from or nearest to v_i.

Status Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] were the first to solve the farthest-neighbor variant of this problem in $O(n)$ time; they achieved this result by reducing the all-farthest-neighbor problem to a Monge-array row-minimization problem and then applying the SMAWK algorithm. This reduction is described in Section 5.1. The all-nearest-neighbor variant of this problem can be solved in linear time using similar techniques; however, the first $O(n)$-time algorithm for computing all nearest neighbors for the vertices of a convex polygon, due to Lee and Preparata [LP78], predates the SMAWK algorithm by many years. Another linear-time algorithm for the all-nearest-neighbors variant of this problem can be obtained using a result due to Aggarwal, Guibas, Saxe, and Shor [AGSS89], who showed that the (nearest-neighbor) Voronoi diagram for the vertices of a convex n-gon can be computed in $O(n)$ time.

All kth-Farthest/kth-Nearest Neighbors for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \le k \le n$, for each vertex v_i, find the vertex v_j whose distance from v_i is the kth largest or kth smallest.

Status Using the reduction mentioned in the previous entry, both variants of this problem can be solved in $O(kn)$ time using the row-selection algorithm of Kravets and J. Park [KP91] and in $O(n^{3/2}\lg^2 n)$ time using the row-selection algorithm of Mansour, J. Park, Schieber, and Sen [MPSS91].

Neighbor Ranking for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, for each vertex v_i, sort the other vertices of P by distance from v_i.

Status In [KP91], Kravets and J. Park showed that their Monge-array row-sorting algorithm solves this problem in $O(n^2)$ time. (This result is briefly mentioned in Section 5.1.) They also showed that if $v_1, \ldots, v_n$ are the vertices of ℓ different convex polygons in the plane, then $O(n^2 \lg \ell)$ time suffices for ranking all the vertices' neighbors.

Farthest/Nearest Pair for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, find the (unordered) pair of vertices (v_i, v_j), separated by the largest or smallest distance.

Status Both variants of this problem are easily solved in $O(n)$ time using the linear-time algorithms mentioned earlier for computing the farthest or nearest neighbor of each vertex of a convex polygon. We remark, however, that the first $O(n)$-time algorithm for the farthest-pair variant for this problem, due to Shamos [Sha78], predates the all-farthest-neighbors algorithm of Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] by many years.

kth-Farthest/kth-Nearest Pair for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order and an integer k in the range $1 \leq k \leq \binom{n}{2}$, find the (unordered) pair of vertices (v_i, v_j) separated by the kth largest or kth smallest distance.

Status Both variants of this problem can be solved in $O(n + k \lg(n^2/k))$ time using the Monge-array array-selection algorithm of Kravets and J. Park [KP91]. (See Section 5.1.)

Pair Ranking for the Vertices of a Convex Polygon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, sort the $\binom{n}{2}$ (unordered) pairs of vertices (v_i, v_j) by separating distance.

Status No $o(n^2 \lg n)$-time algorithm is known for this problem. Reducing this problem to a Monge-array problem is unproductive, since sorting all the entries of an $n \times n$ Monge array requires $\Omega(n^2 \lg n)$ time.

Maximal Inscribed d-gon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order and an integer d in the range $3 \leq d \leq n$, find the maximum-perimeter or maximum-area d-vertex polygon Q contained in P.

Status For the special case of $d = 3$, Dobkin and Snyder [DS79] gave an optimal $O(n)$-time algorithm for the maximum-area variant of this problem. For arbitrary d, the best results are due to Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87]; they showed that both variants of the problem can be solved in $O(dn + n \lg n)$ time. They obtained these results by using the SMAWK algorithm to speed up earlier $O(dn \lg n + n \lg^2 n)$-time algorithms due to Boyce, Dobkin, Drysdale, and Guibas [BDDG85]. Section 5.2 shows how Aggarwal et al.'s algorithm for computing a maximum-perimeter inscribed d-gon can be viewed as computing the plane minima of a d-dimensional cycle-decomposable Monge-composite array.

Minimal Circumscribing d-gon

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order and an integer d in the range $3 \leq d \leq n$, find the minimum-area or minimum-perimeter d-vertex polygon Q containing P.

Status The two best results known for the minimum-area variant of this problem are an optimal $O(n)$-time algorithm for the $d = 3$ special case of the problem, due to O'Rourke, Aggarwal, Maddila, and Baldwin [OAMB86], and an $O(dn + n \lg n)$-time algorithm for the general case of the problem, due to Aggarwal and J. Park [AP89b]. This latter result is described in Section 5.3; it builds on the techniques developed by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] for computing a maximum-perimeter inscribed d-gon (see the previous entry). As for the minimum-perimeter variant of the problem, Aggarwal and J. Park [AP89b] showed that the $d = 3$ special case can be solved in $O(n \lg n)$ time, again using Monge-array techniques.

Minimum-Weight Matching for the Vertices of a Convex Polygon

Problem Given a $2n$-vertex polygon P in the plane with vertices $v_1, \ldots, v_{2n}$ in clockwise order, find a minimum-weight matching of the vertices of P, where the weight associated with the edge between vertices v_i and v_j is the Euclidean distance $\mathrm{d}(v_i, v_j)$ between them.

Status In [MS91], Marcotte and Suri gave an $O(n \lg n)$-time algorithm for this problem. The SMAWK algorithm is an important subroutine of their matching algorithm; it is used to compute nearest neighbors relative to an unusual distance metric. Whether Marcotte and Suri's matching algorithm is optimal remains open, as does the question of whether their techniques are relevant to the matching problem for arbitrary points in the plane. (Currently, the best algorithm known for this more general problem is due to Vaidya [Vai88] and runs in $O(n^{5/2} \lg^4 n)$ time.) Furthermore, He [He91] has parallelized Marcotte and Suri's result; using Atallah and Kosaraju's Monge-array row-minimization algorithm, he (He) obtained an $O(\lg^2 n)$-time, n-processor CREW-PRAM for matching the vertices of a convex polygon.

Optimal Convex-Polygon Triangulation

Problem Given a convex polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, find a triangulation of P minimizing the sum of the lengths of the triangulation's diagonals.

Status Gilbert [Gil79] and Klinseck [Kli80] independently observed that this problem can be solved in $O(n^3)$ time using dynamic programming. Furthermore, if the polygon P satisfies the semicircle property (see Section 5.1 for this property's definition), then this problem can be formulated as an instance of Yao's problem (see entry below) whose interval function $w(\cdot, \cdot)$ both satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals. Thus, this special case of the optimal-convex-polygon-triangulation problem can be solved in $O(n^2)$ time.

Longest Diagonal of a Simple Polygon

Problem Given a simple polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, find the longest line segment connecting a pair of vertices that does not intersect the exterior of P.

Status In [AS90], Aggarwal and Suri gave an $O(n \lg^3 n)$-time algorithm for this problem that uses the SMAWK algorithm to find maximal entries in yet another variety of partial Monge arrays. For the variant of this problem (known as the "biggest stick" problem) that does not constrain the line segment's endpoints to be vertices, the best result is an $O(n^{1.98})$-time algorithm due to Chazelle and Sharir [CS88] that does not use Monge-array techniques.

All Internal Farthest Neighbors of a Simple Polygon's Vertices

Problem Given a simple polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, for each vertex v_i, find the vertex v_j maximizing the length of the shortest path between v_i and v_j that does not intersect the exterior of P.

Status Two $O(n \lg n)$-time algorithms for this problem are known, one due to Guibas and Hershberger [GH87] and the other due to Suri [Sur87]. Both algorithms use the Mongeness of shortest internal paths, but only [GH87] uses the SMAWK algorithm directly.

All External Farthest Neighbors of a Simple Polygon's Vertices

Problem Given a simple polygon P in the plane with vertices $v_1, \ldots, v_n$ in clockwise order, for each vertex v_i, find the point p on P maximizing the length of the shortest path between v_i and p that does not intersect the interior of P.

Status Agarwal, Aggarwal, Aronov, Kosaraju, Schieber, and Suri [AAA+91] gave an $O(n \lg n)$-time algorithm for this problem. More precisely, they showed that this problem can be solved in $O(\tau(n) + \delta(n) + n)$ time, where $\tau(N)$ is the time required to find an *internal* farthest neighbor for each vertex of a simple N-gon (see the previous entry) and $\delta(N)$ is the time required to triangulate a simple N-gon.

Largest Empty Rectangle

Problem Given a rectangle R in the plane containing n points, find the maximum-area or maximum-perimeter subrectangle R' such that

1. R' lies in the interior of R,
2. the sides of R' are parallel to those of R, and
3. the interior of R' contains no points.

Status Aggarwal and Suri [AS87] gave an $O(n \lg^2 n)$-time algorithm for the maximum-area variant of this problem and an $O(n \lg n)$-time algorithm for the maximum-perimeter variant. The former algorithm is based on an array-searching algorithm for computing row minima in certain partial Monge arrays that uses the SMAWK algorithm as a subroutine.

A.3 VLSI Problems

Minimum Separation

Problem Given two VLSI modules P and Q, each modeled as a sequence of n terminals lying on horizontal line, a fixed horizontal offset of P relative to Q, and a set of design rules governing the routing of wires, find the minimum vertical separation of P and Q that permits the routing of n wires satisfying the design rules such that for $1 \leq i \leq n$, the ith wire connects the ith terminal of P to the ith terminal of Q.

Status Provided the design rules governing the routing of wires satisfy certain very general concavity conditions, this problem can be reduced to the Monge-array row-minimization problem. This reduction is due to Siegel and Dolev [SD88]. Thus, so long as the design rules satisfy Siegel and Dolev's concavity conditions, the SMAWK algorithm solves this problem in $O(n)$ sequential time and Atallah and Kosaraju's EREW-PRAM algorithm [AK91] solves this problem in $O(\lg n)$ time using n processors.

Offset Range

Problem Given two VLSI modules P and Q, each modeled as a sequence of n terminals lying on horizontal line, a fixed vertical separation of P and Q, and a set of design rules governing the routing of wires, find all horizontal offsets of P relative to Q that permit the routing of n wires satisfying the design rules such that for $1 \leq i \leq n$, the ith wire connects the ith terminal of P to the ith terminal of Q.

Status In [SD81], Siegel and Dolev argued that this problem can also be reduced to the Monge-array row-minimization problem, provided the design rules satisfy certain concavity conditions. Thus, the time bounds mentioned in the previous entry apply to this problem as well.

Optimal Offset

Problem Given two VLSI modules P and Q, each modeled as a sequence of n terminals lying on horizontal line, and a set of design rules governing the routing of wires, find a horizontal offset of P relative to Q that minimizes the minimum vertical separation of P and Q permitting the routing of n wires satisfying the design rules such that for $1 \leq i \leq n$, the ith wire connects the ith terminal of P to the ith terminal of Q.

Status Siegel and Dolev [SD81] showed that if the separation problem can be solved in $\Phi(n)$ sequential time, then the optimal offset problem can be solved in $\Phi(n) \lg n$ sequential time. Furthermore, Aggarwal and J. Park [AP89a] observed that this reduction is easily parallelized. Thus, so long as the design rules governing the routing of wires satisfy Siegel and Dolev's concavity conditions, the optimal offset problem can be solved in $O(n \lg n)$ time on a sequential RAM and in $O(\lg^2 n)$ time using n processors on a EREW-PRAM.

A.4 Dynamic-Programming Problems

Least-Weight Subsequence

Problem Given an interval function $w(\cdot,\cdot)$, find an integer t in the range $1 \le t \le n$ and a subsequence $i_0, i_1, \ldots, i_t$ of $0, 1, \ldots, n$ minimizing

$$\sum_{s=1}^{t} w(i_{s-1}, i_s) .$$

Status This problem can be solved in $O(n^2)$ time using a simple dynamic-programming approach. Moreover, this time bound is optimal for an arbitrary interval function $w(\cdot,\cdot)$, since we must examine $w(i,j)$ for all i and j satisfying $0 \le i < j \le n$. However, for many applications, $w(\cdot,\cdot)$ satisfies the quadrangle inequality. (Two such applications are the airplane-refueling problem and the optimal-paragraph-formation problem that Hirschberg and Larmore study in [HL87]; this latter problem is briefly described in the introduction.) For this special case of the least-weight-subsequence problem (often called the concave least-weight-subsequence problem), Wilber [Wil88] gave an $O(n)$-time algorithm. His algorithm uses on-line array-searching techniques such as those described in Section 2.2. Finally, if $w(\cdot,\cdot)$ instead satisfies the inverse quadrangle inequality, then Klawe and Kleitman's on-line array-searching algorithm [KK90] (mentioned in Section 2.4) solves the problem in $O(n\alpha(n))$ time. (This last variant is often called the convex least-weight-subsequence problem.)

String Editing

Problem Given functions $D(\cdot)$, $I(\cdot)$, and $S(\cdot,\cdot)$ with domains $\{1,\ldots,m\}$, $\{1,\ldots,n\}$, and $\{1,\ldots,m\} \times \{1,\ldots,n\}$, respectively, find the minimum-cost sequence of deletions, insertions, and substitutions transforming one string $a = a_1 a_2 \cdots a_m$ to another string $b = b_1 b_2 \cdots b_n$, where the cost of deleting a_i is $D(i)$, the cost of inserting b_j is $I(j)$, and the cost of substituting b_j for a_i is $S(i,j)$. (We assume without loss of generality that $m \le n$.)

Status The best sequential result known for this generalization of the longest-common-subsequence problem is an $O(mn)$-time algorithm due to Wagner and Fischer [WF74]. Kedem and Fuchs [KF80] considered a variant of the string-editing problem that allows the string a to be rotated at no cost prior to any editing. (In other words, they sought a rotation i from the range $1 \le i \le m$ minimizing the cost of transforming $a_i a_{i+1} \cdots a_n a_1 \cdots a_{i-1}$ to b.) For this problem, Kedem and Fuchs gave an $O(mn \lg m)$-time algorithm. Eppstein [Epp90] considered yet another variant of the string-editing problem that allows any substring $a_i a_{i+1} \cdots a_{k-1}$ of a to be deleted at cost $D(i,k)$ and any substring $b_j b_{j+1} \cdots b_{\ell-1}$ of b to be inserted at cost $I(j,\ell)$. For certain functions $D(\cdot,\cdot)$ and $I(\cdot,\cdot)$, Eppstein showed that the naive $O(mn^2)$-time dynamic-programming algorithm for this variant of the string-editing problem can be sped up using an on-line Monge-array row-minimization algorithm. As for parallel results, both the basic string-editing

problem and Kedem and Fuch's variant can be reduced to the grid-DAG-shortest-paths problem described below. (We discuss this reduction in Section 8.2.) Combining this reduction with parallel algorithms for the grid-DAG-shortest-paths problem gives three results: an $O(\lg m \lg n)$-time, $(mn/\lg m)$-processor CRCW-PRAM algorithm, an $O(\lg\lg m \lg n)$-time, $(mn/\lg\lg m)$-processor CREW-PRAM algorithm, and an $O(\lg m \lg n)$-time algorithm for an mn-node hypercube.

Surface Reconstruction from Planar Contours

Problem Given an m-vertex convex polygon P and an n-vertex convex polygon Q such that P and Q lie in parallel planes in three-dimensional space, use triangular tiles to construct a minimum-cost cylindrical surface joining P and Q, where the cost of a surface is the sum of the costs some triangle-weighting function assigns to the triangular tiles forming the surface. (We assume without loss of generality that $m \leq n$.)

Status In [FKU77], Fuchs, Kedem, and Uselton reduced this problem to an instance of the grid-DAG-shortest-paths problem described in the next entry. We discuss this reduction in Section 8.3.

Shortest Paths in Grid DAGs

Problem Given an $2m \times (n+1)$ grid DAG (as defined in Chapter 8) where $m \leq n$, find the shortest path from $v_{i,1}$ to $v_{m+i,n+1}$ for all i in the range $1 \leq i \leq m$.

Status Fuchs, Kedem, and Uselton [FKU77] gave an $O(mn \lg m)$-time sequential algorithm for this problem. As for parallel results, Apostolico, Atallah, Larmore, and McFaddin [AALM90] reduced this problem to several instances of the Monge-array tube-minimization problem. (This reduction is described in Section 8.1.) Combining this reduction with the best known parallel tube-minimization results gives an $O(\lg n \lg m)$-time, $(mn/\lg m)$-processor CREW-PRAM algorithm, an $O(\lg n \lg\lg m)$-time, $(mn/\lg\lg m)$-processor CRCW-PRAM algorithm, and an $O(\lg m \lg n)$-time algorithm for an mn-node hypercube.

Waterman's Problem

Problem Given an interval function $w(\cdot,\cdot)$ and values $E(0,0)$, $E(1,0), \ldots, E(n,0)$, and $E(0,1), \ldots, E(0,n)$, compute $E(i,j)$ for all i and j satisfying $1 \leq i \leq n$ and $1 \leq j \leq n$ from the recurrence

$$E(i,j) = \min_{\substack{0 \leq i' < i \\ 0 \leq j' < j}} \{F(i',j') + w(i'+j', i+j)\} ,$$

where $F(i,j)$ is some function that can be computed in constant time from $E(i,j)$.

Status This problem arises in the prediction of RNA secondary structure [Wat78, WS86]. If $w(\cdot,\cdot)$ satisfies the quadrangle inequality, then Waterman's problem can be solved in $O(n^2)$ time using the on-line LIEBER algorithm of Section 2.2. This result is due to Larmore and Schieber [LS91]. Larmore and Schieber also gave an $O(n^2\alpha(n))$-time algorithm for the case where $w(\cdot,\cdot)$ satisfies the inverse quadrangle inequality;

this latter algorithm is based on Klawe and Kleitman's on-line array-searching algorithm [KK90], which is briefly mentioned in Section 2.4.

Huffman Coding

Problem Given probabilities $p_1, \ldots, p_n$ such that $p_1 + \cdots + p_n = 1$, construct a prefix code for characters $a_1, \ldots, a_n$ minimizing the expected length of the code for the character b, where b is selected at random so that

$$\Pr\{b = a_i\} = p_i .$$

Status Huffman [Huf52] gave a simple greedy $O(n \lg n)$-time algorithm for this problem. (This running time can be reduced to $O(n)$ if $p_1, \ldots, p_n$ are given in sorted order.) Furthermore, Atallah, Kosaraju, Larmore, Miller, and Teng [AKL+89] reduced this problem to $O(\lg n)$ instances of the Monge-array tube-minimization problem, thereby obtaining an $O(\lg^2 n)$-time, $(n^2/\lg n)$-processor CREW-PRAM algorithm and an $O(\lg n \lg\lg n)$-time, $(n^2/\lg\lg n)$-processor CRCW-PRAM algorithm for Huffman coding.

Optimal Binary Search Trees

Problem Given probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ such that $p_1 + \cdots + p_n + q_0 + \cdots + q_n = 1$, construct a binary search tree for keys $a_1 < a_2 < \cdots < a_n$ minimizing the expected number of comparisons performed by a find operation whose argument b is selected at random so that

$$\Pr\{b = a_i\} = p_i$$

and

$$\Pr\{a_i < b < a_{i+1}\} = q_i .$$

(By convention, $a_0 = -\infty$ and $a_{n+1} = +\infty$.)

Status The asymptotically fastest algorithms known for this problem run in $O(n^2)$ time. Knuth gave such an algorithm in [Knu71]. Yao [Yao80] showed that an instance of this problem can be reduced to an instance of Yao's problem (see the following entry) whose interval function $w(\cdot,\cdot)$ both satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals. (This simple reduction is given in Section 6.2.) Thus, the algorithms of Yao [Yao80] and Aggarwal and J. Park [AP89b] mentioned below give alternate $O(n^2)$-time algorithms for the optimal-binary-search-tree problem. Furthermore, Larmore [Lar87] gave a subquadratic-time algorithm for approximating an optimal binary search tree. For any constant $r > 1/(1 + \lg \phi) \approx 0.59023$, where ϕ is the "golden ratio" $(1 + \sqrt{5})/2$, Larmore's algorithm finds, in $O(n^{1+r})$ time, a binary search tree for $a_1, \ldots, a_n$ whose expected number of comparisons per find operation differs from the optimal expected number of comparisons per find operation by $o(1)$. An important subproblem solved by Larmore's algorithm involves computing the row minima of a two-dimensional Monge array. Finally, Atallah, Kosaraju, Larmore, Miller, and Teng [AKL+89] reduced a variant of Larmore's approximately-optimal-binary-search-tree problem to $O(\lg n)$ instances of the Monge-array tube-minimization problem.

Yao's Problem

Problem Given an interval function $w(\cdot,\cdot)$, compute $E(1, n+1)$ from the recurrence

$$E(i,j) = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) + \min_{i \le k < j}\{E(i,k) + E(k+1,j)\} & \text{if } i < j. \end{cases}$$

Status For an arbitrary interval function $w(\cdot,\cdot)$, the best result known for this problem is the $O(n^3)$-time algorithm obtained by evaluating the $E(i,j)$ in the straightforward manner. However, if $w(\cdot,\cdot)$ both satisfies the quadrangle inequality and is monotonically increasing on the lattice of intervals, then Yao [Yao80] showed that this problem can be solved in $O(n^2)$ time. Furthermore, Aggarwal and J. Park [AP89b] gave an alternate $O(n^2)$ algorithm for this special case that uses the SMAWK algorithm as a subroutine. Finally, Aggarwal and Park also showed that if $w(\cdot,\cdot)$ both satisfies the *inverse* quadrangle inequality and is monotonically *decreasing* on the lattice of intervals, then Yao's problem can be solved in $O(n^2\alpha(n))$ time using Klawe and Kleitman's on-line algorithm for computing the row minima of a partial inverse-Monge array of the staircase variety. These last two results are described in Section 6.2.

Optimal Matrix-Chain Multiplication

Problem Given a sequence of $n+1$ dimensions $p_0, p_1, \ldots, p_n$, compute an optimal parenthesization for the matrix product $A_1 A_2 \cdots A_n$, where for $1 \le i \le n$, A_i is a $p_{i-1} \times p_i$ matrix. (We assume that the $p \times r$ matrix product of a $p \times q$ matrix and a $q \times r$ matrix requires pqr time to compute.)

Status No way of using Monge arrays to solve this problem efficiently is known, despite the problem's superficial similarity to Yao's problem (see the previous entry). The best result for this problem is an $O(n \lg n)$-time algorithm due to Hu and Shing [HS82, HS84].

A.5 Problems from Operations Research

Economic Lot Sizing

Problem Given an n-period production system characterized by demands $d_1, \ldots, d_n$, production cost functions $c_1(\cdot), \ldots, c_n(\cdot)$, and inventory cost functions $h_2(\cdot), \ldots, h_n(\cdot)$, find a minimum-cost production schedule $x_1, \ldots, x_n$ such that

1. $x_1 + \ldots + x_n = d_1 + \ldots + d_n$, and
2. for $2 \le i \le n$, the excess inventory

$$y_i = \sum_{\ell=1}^{i-1} x_\ell - \sum_{\ell=1}^{i-1} d_\ell$$

carried from period $i-1$ to period i is nonnegative,

where the cost of a production schedule (and the inventory quantities it induces) is given by

$$\sum_{i=1}^{n} c_i(x_i) + \sum_{i=2}^{n} h_i(y_i) .$$

Status For arbitrary production and inventory cost functions, this problem is NP-hard, as Florian, Lenstra, and Rinnooy Kan showed in [FLR80]. However, if the production and inventory cost functions are all concave on $[0, +\infty)$, then Veinott [Vei63] has shown that a minimum-cost production schedule can be found in $O(n^2)$ time using dynamic programming. Furthermore, if for all i in the range $1 \leq i \leq n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

and $h_i(y) = h_i^1 y$, where c_i^0, c_i^1, and h_i^1 are nonnegative constants, then $O(n \lg n)$ time suffices to find an optimal production schedule. Finally, if we assume that the constants c_i^1 and h_i^1 given above also satisfy $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$, then the problem can be solved in $O(n)$ time. These last two results were obtained independently by Aggarwal and J. Park [AP91], Federgruen and Tzur [FT89], and Wagelmans, van Hoesel, and Kolen [WvHK89]. Only Aggarwal and Park use Monge-array techniques; their approach is described in Chapter 7.

Economic Lot Sizing with Backlogging

Problem This problem is identical to basic economic-lot-sizing problem given in the previous entry, except that here we allow the inventory y_i carried from period $i-1$ to period i to be negative for $2 \leq i \leq n$. (Such negative inventory is called backlogged demand.)

Status If the production cost functions are all concave on $[0, +\infty)$ and the backlogging/inventory cost functions are all concave on $(-\infty, 0]$ and on $[0, +\infty)$, then Aggarwal and J. Park [AP91] have shown that a minimum-cost production schedule can be found in $O(n^2)$ time using Monge-array techniques. Furthermore, if for all i in the range $1 \leq i \leq n$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

$g_i(y) = g_i^1 y$, and $h_i(y) = h_i^1 y$, where c_i^0, c_i^1, g_i^1, and h_i^1 are nonnegative constants, then $O(n \lg n)$ time suffices to find an optimal production schedule. Finally, if we assume that the constants c_i^1, g_i^1, and h_i^1 given above also satisfy $c_i^1 \leq c_{i+1}^1 + g_i^1$ for $1 \leq i < n$ and $c_i^1 \leq c_{i-1}^1 + h_i^1$ for $1 < i \leq n$, then the problem can be solved in $O(n)$ time. These last two results were obtained independently by Aggarwal and J. Park [AP91] and Federgruen and Tzur [FT90]. Again only Aggarwal and Park use Monge-array techniques; their approach is described in Chapter 7.

Economic Lot Sizing with Capacities

Problem This problem is identical to economic-lot-sizing problem with backlogging described in the previous entry, except that here we are also given bounds $x_1^L, \ldots, x_n^L$, $x_1^U, \ldots, x_n^U$, $y_2^L, \ldots, y_n^L$, and $y_2^U, \ldots, y_n^U$ on production and backlogging/inventory, such that the production schedule we seek (and the inventory quantities it induces) must satisfy $x_i^L \leq x_i \leq x_i^U$ for $1 \leq i \leq n$ and $y_i^L \leq y_i \leq y_i^U$ for $2 \leq i \leq n$.

Status For arbitrary upper bounds x_i^U on production, Florian, Lenstra, and Rinnooy Kan [FLR80] showed that this problem is NP-hard, even if we assume that

1. $c_i(\cdot)$ is concave on $[x_i^L, x_i^U]$ for $1 \leq i \leq n$,
2. $h_i(\cdot)$ is concave on $[y_i^L, 0]$ and on $[0, y_i^U]$ for $2 \leq i \leq n$, and
3. $x_i^L = 0$, $y_i^L = -\infty$, and $y_i^U = +\infty$ for $1 \leq i \leq n$.

However, if we also assume that all the upper bounds on production are the same (i.e., $x_i^U = x^U$ for $1 \leq i \leq n$), then Florian and Klein [FK71] proved that an optimal production schedule can be computed in $O(n^4)$ time. Furthermore, Aggarwal and J. Park [AP90] showed that if

1. $c_i(\cdot)$ is concave on $[x_i^L, x_i^U]$ for $1 \leq i \leq n$,
2. $h_i(\cdot)$ is concave on $[y_i^L, 0]$ and on $[0, y_i^U]$ for $2 \leq i \leq n$, and
3. there are no bounds on production, i.e., $x_i^L = 0$ and $x_i^U = +\infty$ for $1 \leq i \leq n$,

then $O(n^2\alpha(n))$ time suffices for computing an optimal production schedule. Finally, if for $1 \leq i \leq n$, we also assume $y_i^L = 0$,

$$c_i(x) = \begin{cases} 0 & \text{if } x = 0, \\ c_i^0 + c_i^1 x & \text{if } x > 0, \end{cases}$$

and $h_i(y) = h_i^1 y$, where c_i^0, c_i^1, and h_i^1 are nonnegative constants, then Aggarwal and Park showed that this problem can be solved in $O(n \lg n \alpha(n))$ time. These last two results build on the work of Love [Lov73] and use Monge-array techniques similar to those employed in Chapter 7.

A.6 Graph-Theoretic Problems

Weighted Bipartite Matching

Problem Given an $m \times n$ cost array $C = \{c[i,j]\}$ such that $m \geq n$, find a minimum-cost n-edge matching for the complete undirected bipartite graph $G = (U \cup V, E)$ given by $U = \{u_1, u_2, \ldots, u_m\}$, $V = \{v_1, v_2, \ldots, v_n\}$, and $E = U \times V$, where the edge (u_i, v_j) has cost $c[i,j]$.

Status For an arbitrary array C, the best known algorithm, due to Kuhn [Kuh55], takes $O(n^3)$ time. However, if C is a Monge array, then the problem is much simpler. In particular, if $m = n$, then $\{(u_i, v_i) : 1 \leq i \leq m\}$ is a minimum-cost matching, and if $m > n$, a simple dynamic-programming algorithm finds an optimal matching in $O(m(n-m))$ time.

Traveling-Salesman Tour

Problem Given an $n \times n$ cost array $C = \{c[i,j]\}$, find a minimum-cost tour through the n-vertex complete directed graph $G = (V, A)$ given by $V = \{v_1, v_2, \ldots, v_n\}$ and $A = V \times V$, where the cost of traversing arc (v_i, v_j) is $c[i,j]$. (A *tour* through G is a cycle that passes though each of G's n vertices exactly once.)

Status In general, this (very famous) problem is NP-hard. However, if the cost array C is Monge, then J. Park [Par91] has shown that the on-line Monge-array row-minimization algorithm of Section 2.2 can be used to find a minimum-cost tour in $O(n)$ time. This result is described in Section 6.1. Furthermore, Gilmore and Gomory [GG64] have described an $O(n \lg n)$-time algorithm for another special case of the traveling salesman problem corresponding to a set Ξ of cost arrays with the property that, for any $C \in \Xi$, some permutation of C's columns yields a Monge array. (Note that this result does not hold for every C such that some permutation of C's columns yields a Monge array; in fact, as Gilmore, Lawler, and Shmoys proved in [GLS85], the traveling salesman remains NP-hard for some such C.)

Uncapacitated Transportation

Problem Given an $m \times n$ cost array $C = \{c[i,j]\}$, a vector $A = \{a[i]\}$ of m supplies, and a vector $B = \{b[j]\}$ of n demands, where $a[1]+a[2]+\cdots+a[m] = b[1]+b[2]+\cdots+b[n]$, find a minimum-cost flow in the complete uncapacitated directed bipartite graph $D = (U \cup V, A)$ given by $U = \{u_1, u_2, \ldots, u_m\}$, $V = \{v_1, v_2, \ldots, v_n\}$, and $A = U \times V$, where each vertex $u_i \in U$ is a source supplying $a[i]$ units of flow, each vertex $v_j \in U$ is a sink demanding $b[j]$ units of flow, and the cost of sending one unit of flow across the arc (u_i, v_j) is $c[i,j]$. (We assume without loss of generality that $m \leq n$.)

Status The best result known for the general case of this problem is an $O(mn^2 \lg n + n^2 \lg^2 n)$-time algorithm due to Orlin [Orl88]. (Orlin's algorithm is the fastest strongly-polynomial algorithm known for the minimum-cost flow problem, to which this problem is easily reduced.) However, if the cost array C is Monge, then, as Hoffman observed in [Hof63], the uncapacitated transportation problem can be solved in $O(m+n)$ time with a simple greedy algorithm. More generally, Hoffman defined a *Monge sequence* for a cost array C to be an ordering of the arcs of D such that for any two sources u_i and u_k and any two sinks v_j and v_ℓ, whenever (i,j) precedes both (i,ℓ) and (j,k), C satisfies $c[i,j] + c[k,\ell] \leq c[i,\ell] + c[k,j]$. He then proposed an $O(mn)$-time greedy algorithm that selects the lexicographically first feasible flow according to some ordering σ of D's arcs and proved that such a greedy algorithm produces a minimum-cost flow for all supply and demand vectors A and B if and only if σ is a Monge sequence for C. More recently, Alon, Cosares, Hochbaum, and Shamir [ACHS89] discovered an $O(m^2 n \lg n)$-time algorithm that constructs a Monge sequence for a particular cost array C, provided such a sequence exists. Furthermore, Shamir and Dietrich [SD90] generalized these ideas to bipartite graphs D that are not complete, i.e., $A \subset U \times V$. Finally, Bein, Brucker, and Pathak [BBP90] considered a d-dimensional generalization of the uncapacitated transportation problem with d-dimensional cost arrays and identified a class of Monge-like cost arrays for which this problem can be solved efficiently.

Bibliography

[AAA+91] P. K. Agarwal, A. Aggarwal, B. Aronov, S. R. Kosaraju, B. Schieber, and S. Suri. Computing external-farthest neighbors for a simple polygon. *Discrete Applied Mathematics*, 1991. To appear. An earlier version of this paper appears as Research Report RC 15119, IBM T. J. Watson Research Center, October 1989.

[AALM90] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[ACG87] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–160, 1987.

[ACHS89] N. Alon, S. Cosares, D. S. Hochbaum, and R. Shamir. An algorithm for the detection and construction of Monge sequences. *Linear Algebra and Its Applications*, 114/115:669–680, 1989.

[ACY85] A. Aggarwal, J. S. Chang, and C. K. Yap. Minimum area circumscribing polygons. *The Visual Computer*, 1(2):112–117, 1985.

[AGSS89] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4(6):591–604, 1989.

[AK91] M. J. Atallah and S. R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 394–403, 1991. Submitted to *Journal of Algorithms*.

[AKL+89] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. Miller, and S. Teng. Constructing trees in parallel. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, 1989.

[AKM+87] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(2):195–208, 1987.

[AKPS90] A. Aggarwal, D. Kravets, J. K. Park, and S. Sen. Parallel searching in generalized Monge arrays with applications. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 259–268, 1990.

[AP88] A. Aggarwal and J. K. Park. Notes on searching in multidimensional monotone arrays. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[AP89a] A. Aggarwal and J. K. Park. Parallel searching in multidimensional monotone arrays. Research Report RC 14826, IBM T. J. Watson Research Center, August 1989. Submitted to *Journal of Algorithms.* Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[AP89b] A. Aggarwal and J. K. Park. Sequential searching in multidimensional monotone arrays. Research Report RC 15128, IBM T. J. Watson Research Center, November 1989. Submitted to *Journal of Algorithms.* Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[AP90] A. Aggarwal and J. K. Park. Algorithms for economic lot-size problems with bounds on inventory and backlogged demand. Unpublished manuscript, September 1990.

[AP91] A. Aggarwal and J. K. Park. Improved algorithms for economic lot-size problems. *Operations Research*, 1991. To appear. An earlier version of this paper appears as Research Report RC 15626, IBM T. J. Watson Research Center, March 1990.

[AS87] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry*, pages 278–290, 1987. Submitted to *SIAM Journal on Computing.*

[AS90] A. Aggarwal and S. Suri. Computing the longest diagonal of a simple polygon. *Information Processing Letters*, 35(1):13–18, 1990. Also published as IBM Research Report 14775, July 1989.

[Ata90] M. J. Atallah. A faster parallel algorithm for a matrix searching problem. In G. Goos and J. Hartmanis, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, pages 192–200, New York, NY, 1990. Springer-Verlag. Submitted to *Algorithmica.*

[BBP90] W. W. Bein, P. Brucker, and P. K. Pathak. Monge properties in higher dimensions. Technical Report CS90-11, University of New Mexico, October 1990.

[BDDG85] J. E. Boyce, D. P. Dobkin, R. L. Drysdale, and L. J. Guibas. Finding extremal polygons. *SIAM Journal on Computing*, 14(1):134–147, 1985.

[Bel57] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[BK74] J. D. Blackburn and H. Kunreuther. Planning horizons for the dynamic lot size model with backlogging. *Management Science*, 21(3):251–255, 1974.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[BRG87] H. C. Bahl, L. R. Ritzman, and J. N. D. Gupta. Determining lot sizes and resource requirements: A review. *Operations Research*, 35(3):329–345, 1987.

[BY82] G. R. Bitran and H. H. Yanasse. Computational complexity of the capacitated lot size problem. *Management Science*, 28(10):1174–1186, 1982.

[CL88] C.-S. Chung and C.-H. M. Lin. An $O(T^2)$ algorithm for NI/G/NI/ND capacitated lot size problem. *Management Science*, 34(3):420–426, 1988.

[CS88] B. M. Chazelle and M. Sharir. An algorithm for generalized point location and its applications. Technical Report ?, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1988.

[CY84] J. S. Chang and C. K. Yap. A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 408–416, 1984.

[Den82] E. V. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[DeP87] N. A. A. DePano. *Polygon Approximation with Optimized Polygonal Enclosures: Applications and Algorithms.* PhD thesis, Department of Computer Science, The Johns Hopkins University, Baltimore, MD, 1987.

[DS79] D. P. Dobkin and L. Snyder. On a general method for maximizing and minimizing among certain geometric problems. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 9–17, 1979.

[EGG88] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 488–496, 1988.

[EGP69] G. D. Eppen, F. J. Gould, and B. P. Pashigian. Extensions of the planning horizon theorem in the dynamic lot size model. *Management Science*, 15(5):268–277, 1969.

[EMV87] R. E. Erickson, C. L. Monma, and A. F. Veinott, Jr. Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research*, 12(4):634–664, 1987.

[Epp90] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11(1):85–101, 1990.

[FJ82] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X + Y and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(4):197–208, 1982.

[FK71] M. Florian and M. Klein. Deterministic production planning with concave costs and capacity constraints. *Management Science*, 18(1):12–20, 1971.

[FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, 1977.

[FLR80] M. Florian, J. K. Lenstra, and A. H. G. Rinnooy Kan. Deterministic production planning: Algorithms and complexity. *Management Science*, 26(7):669–679, 1980.

[FR75] C. O. Fong and M. R. Rao. Capacity expansion with two producing regions and concave costs. *Management Science*, 22(3):331–339, 1975.

[FT89] A. Federgruen and M. Tzur. A simple forward algorithm to solve general dynamic lot sizing models with n periods in $O(n \log n)$ or $O(n)$ time. Working paper. Graduate School of Business, Columbia University, New York, NY, 1989.

[FT90] A. Federgruen and M. Tzur. The dynamic lot sizing model with backlogging: A simple $O(n \log n)$ algorithm. Working paper. Graduate School of Business, Columbia University, New York, NY, 1990.

[GG64] P. C. Gilmore and R. E. Gomory. Sequencing a one state-variable machine: A solvable case of the traveling salesman problem. *Operations Research*, 12(5):655–679, 1964.

[GH87] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry*, pages 50–63, 1987.

[Gil79] P. D. Gilbert. New results on planar triangulations. Technical report, Coordinated Science Laboratory, University of Illinois, 1979.

[GLS85] P. C. Gilmore, E. L. Lawler, and D. B. Shmoys. Well-solved special cases. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem*, pages 87–143. John Wiley & Sons, Inc., New York, NY, 1985.

[GO85] S. C. Graves and J. B. Orlin. A minimum concave-cost dynamic network flow problem with an application to lot-sizing. *Networks*, 15(1):59–71, 1985.

[GP90] Z. Galil and K. Park. A linear-time algorithm for concave one-dimensional dynamic programming. *Information Processing Letters*, 33(6):309–311, 1990.

[Gra82] S. C. Graves. Using Lagrangian techniques to solve hierarchical production planning problems. *Management Science*, 28(3):260–275, 1982.

[Har15] F. W. Harris. What quantity to make at once. In *Operation and Costs: Planning and Filling Orders, Cost-Keeping Methods, Controlling Your Operations, Standardizing Material and Labor Costs*, volume 5 of *The Library of Factory Management*, pages 47–52. A. W. Shaw Co., Chicago, IL, 1915.

[HC84] A. C. Hax and D. Candea. *Production and Inventory Management*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[He91] X. He. An efficient parallel algorithm for finding minimum weight matching for points on a convex polygon. *Information Processing Letters*, 37(2):111–116, 1991.

[HL87] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

[Hof63] A. J. Hoffman. On simple linear programming problems. In V. Klee, editor, *Convexity: Proceedings of the Seventh Symposium in Pure Mathematics of the AMS*, volume 7 of *Proceedings of Symposia in Pure Mathematics*, pages 317–327. American Mathematical Society, Providence, RI, 1963.

[HS82] T. C. Hu and M. T. Shing. Computation of matrix chain products, part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[HS84] T. C. Hu and M. T. Shing. Computation of matrix chain products, part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[JM74] L. A. Johnson and D. C. Montgomery. *Operations Research in Production Planning, Scheduling, and Inventory Control.* John Wiley & Sons, Inc., New York, NY, 1974.

[KF80] Z. M. Kedem and H. Fuchs. On finding several shortest paths in certain graphs. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 677–686, 1980.

[KK90] M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.

[KL85] V. Klee and M. C. Laskowski. Finding the smallest triangles containing a given convex polygon. *Journal of Algorithms*, 6(3):359–375, 1985.

[Kla89] M. M. Klawe. A simple linear time algorithm for concave one-dimensional dynamic programming. Technical Report 89-16, University of British Columbia, Vancouver, 1989.

[Kli80] G. T. Klinseck. Minimal triangulations of polygonal domains. *Annals of Discrete Mathematics*, 9:121–123, 1980.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publishing Co., Reading, MA, 1973.

[KP91] D. Kravets and J. K. Park. Selection and sorting in totally monotone arrays. *Mathematical Systems Theory*, 1991. To appear. An earlier version of this paper appears in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 494–502, 1990.

[Kru83] C. P. Kruskal. Searching, merging, and sorting. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[Kuh55] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[Lar87] L. L. Larmore. A subquadratic algorithm for constructing approximately optimal binary search trees. *Journal of Algorithms*, 8(4):579–591, 1987.

[Lar90] L. L. Larmore. An optimal algorithm with unknown time complexity for convex matrix searching. *Information Processing Letters*, 36(3):147–151, 1990.

[LD86] C.-Y. Lee and E. V. Denardo. Rolling planning horizons: Error bounds for the dynamic lot size model. *Mathematics of Operations Research*, 11(3):423–432, 1986.

[LL87] S.-B. Lee and H. Luss. Multifacility-type capacity expansion planning: Algorithms and complexities. *Operations Research*, 35(2):249–253, 1987.

[LM75] R. A. Lundin and T. E. Morton. Planning horizons for the dynamic lot size model. *Operations Research*, 23(4):711–734, 1975.

[Lov73] S. F. Love. Bounded production and inventory models with piecewise concave costs. *Management Science*, 20(3):313–318, 1973.

[Lov83] L. Lovász. Submodular functions and convexity. In A. Bachem, M. Grötschel, and B. Korte, editors, *Mathematical Programming: The State of the Art, Bonn 1982*, pages 235–257. Springer-Verlag, New York, NY, 1983.

[LP78] D.-T. Lee and F. P. Preparata. The all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 7(4):189–192, 1978.

[LS91] L. L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure. *Journal of Algorithms*, 1991. To appear. An earlier version of this paper appears in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 503–512, 1990.

[Lus79] H. Luss. A capacity expansion model for two facility types. *Naval Research Logistics Quarterly*, 26(2):291–303, 1979.

[Lus82] H. Luss. Operations research and capacity expansion problems: A survey. *Operations Research*, 30(5):907–947, 1982.

[Lus86] H. Luss. A heuristic for capacity expansion planning with multiple facility types. *Naval Research Logistics Quarterly*, 33(4):685–701, 1986.

[Man58] A. S. Manne. Programming of economic lot sizes. *Management Science*, 4(2):115–135, 1958.

[Man67] A. S. Manne, editor. *Investments for Capacity Expansion; Size, Location, and Time-Phasing*. MIT Press, Cambridge, MA, 1967.

[Mat88] T. R. Mathies. A fast parallel algorithm to determine edit distance. Technical Report CMU-CS-88-130, Carnegie-Mellon University, April 1988.

[Mon81] G. Monge. Déblai et remblai. Mémoires de l'Académie des Sciences, 1781.

[Mor78] T. E. Morton. An improved algorithm for the stationary cost dynamic lot size model with backlogging. *Management Science*, 24(8):869–873, 1978.

[MPSS91] Y. Mansour, J. K. Park, B. Schieber, and S. Sen. Improved selection in totally monotone arrays. Submitted to the Eleventh Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, May 1991.

[MS91] O. Marcotte and S. Suri. Fast matching algorithms for points on a polygon. *SIAM Journal on Computing*, 1991. To appear. An earlier version of this paper appears in *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 60–65, 1989.

[MV67] A. S. Manne and A. F. Veinott, Jr. Optimal plant size with arbitrary increasing time paths of demand. In A. S. Manne, editor, *Investments for Capacity Expansion; Size, Location, and Time-Phasing*, pages 178–190. MIT Press, Cambridge, MA, 1967.

[OAMB86] J. O'Rourke, A. Aggarwal, S. Maddila, and M. Baldwin. An optimal algorithm for finding minimal enclosing triangles. *Journal of Algorithms*, 7(2):258–268, 1986.

[Orl88] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 377–387, 1988.

[Par91] J. K. Park. A special case of the n-vertex traveling-salesman problem that can be solved in $O(n)$ time. Submitted to *Information Processing Letters*, April 1991.

[SD81] A. Siegel and D. Dolev. The separation for general single-layer wiring barriers. In H. T. Kung, B. Sproull, and G. Steele, editors, *Proceedings of the CMU Conference on VLSI Systems and Computations*, pages 143–152, Rockville, MD, 1981. Computer Science Press, Inc.

[SD88] A. Siegel and D. Dolev. Some geometry for general river routing. *SIAM Journal on Computing*, 17(3):583–605, 1988.

[SD90] R. Shamir and B. Dietrich. Characterization and algorithms for greedily solvable transportation problems. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 358–366, 1990.

[Sha78] M. I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.

[Sur87] S. Suri. The all-geodesic-furthest neighbors problem for simple polygons. In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry*, pages 64–75, 1987.

[SV81] Y. Shiloach and V. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[Vai88] P. M. Vaidya. Geometry helps in matching. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 422–425, 1988.

[Vei63] A. F. Veinott, Jr. Unpublished class notes. Program in Operations Research, Stanford University, Stanford, CA, 1963.

[Vei69] A. F. Veinott, Jr. Minimum concave-cost solution of Leontief substitution models of multi-facility inventory systems. *Operations Research*, 17(2):262–291, 1969.

[Wag60] H. M. Wagner. A postscript to "Dynamic problems in the theory of the firm". *Naval Research Logistics Quarterly*, 7(1):7–12, 1960.

[Wag75] H. M. Wagner. *Principles of Operations Research, with Applications to Managerial Decisions*. Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1975.

[Wat78] M. S. Waterman. Secondary structure of single-stranded nucleic acids. In G.-C. Rota, editor, *Studies in Foundations and Combinatorics*, volume 1 of *Advances in Mathematics: Supplementary Studies*, pages 167–212. Academic Press, Inc., New York, NY, 1978.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[Wil88] R. Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, 1988.

[WS86] M. S. Waterman and T. F. Smith. Rapid dynamic programming algorithms for RNA secondary structure. *Advances in Applied Mathematics*, 7:455–464, 1986.

[WvHK89] A. Wagelmans, S. van Hoesel, and A. Kolen. Economic lot-sizing: An $O(n \log n)$-algorithm that runs in linear time in the Wagner-Whitin case. Report 8952/A, Econometric Institute, Erasmus University Rotterdam, Rotterdam, The Netherlands, 1989.

[WW58] H. M. Wagner and T. M. Whitin. Dynamic version of the economic lot size model. *Management Science*, 5(1):89–96, 1958.

[Yao80] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 429–435, 1980.

[Yao82] F. F. Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 3(4):532–540, 1982.

[YL79] C.-C. Yang and D.-T. Lee. A note on the all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 8(4):193–194, 1979.

[Zab64] E. Zabel. Some generalizations of an inventory planning horizon theorem. *Management Science*, 10(3):465–471, 1964.

[Zan66] W. I. Zangwill. A deterministic multi-period production scheduling model with backlogging. *Management Science*, 13(1):105–119, 1966.

[Zan68] W. I. Zangwill. Minimum concave cost flows in certain networks. *Management Science*, 14(7):429–450, 1968.

[Zan69] W. I. Zangwill. A backlogging model and a multiechelon model of a dynamic economic lot size production system — a network approach. *Management Science*, 15(9):506–527, 1969.