Network Optimization: Continuous and Discrete Models

Dimitri P. Bertsekas

Massachusetts Institute of Technology

WWW site for book information and orders

http://www.athenasc.com

Athena Scientific, Belmont, Massachusetts

Athena Scientific
Post Office Box 391
Belmont, Mass. 02178-9998
U.S.A.

Email: info@athenasc.com
WWW: http://www.athenasc.com

Cover Design: *Ann Gallager*

Publisher's Cataloging-in-Publication Data

Bertsekas, Dimitri P.
Network Optimization: Continuous and Discrete Models
Includes bibliographical references and index
1. Network analysis (Planning). 2. Mathematical Optimization. I. Title.
T57.85.B44 1998   658.4'032-dc20   98-70298

ISBN 1-886529-02-7

ABOUT THE AUTHOR

Dimitri Bertsekas studied Mechanical and Electrical Engineering at the National Technical University of Athens, Greece, and obtained his Ph.D. in system science from the Massachusetts Institute of Technology.

He has held faculty positions at Stanford University and the University of Illinois. Since 1979 he has been teaching at the Massachusetts Institute of Technology (M.I.T.), where he is currently McAfee Professor of Engineering. He consults regularly with private industry and has held editorial positions in several journals. His research spans several fields, including optimization, control, large-scale computation, and data communication networks. He has written many research papers and he is the author or coauthor of thirteen textbooks and research monographs.

Professor Bertsekas was awarded the INFORMS 1997 Prize for Research Excellence in the Interface Between Operations Research and Computer Science for his book ”Neuro-Dynamic Programming” (co-authored with John Tsitsiklis), the 2000 Greek National Award for Operations Research, and the 2001 ACC John R. Ragazzini Education Award. In 2001, he was elected to the United States National Academy of Engineering.

Contents

1. Introduction . . . . p. 1

1.1. Graphs and Flows . . . . p. 3
1.1.1. Paths and Cycles . . . . p. 4
1.1.2. Flow and Divergence . . . . p. 6
1.1.3. Path Flows and Conformal Decomposition . . . . p. 7
1.2. Network Flow Models – Examples . . . . p. 8
1.2.1. The Minimum Cost Flow Problem . . . . p. 9
1.2.2. Network Flow Problems with Convex Cost . . . . p. 16
1.2.3. Multicommodity Flow Problems . . . . p. 17
1.2.4. Discrete Network Optimization Problems . . . . p. 19
1.3. Network Flow Algorithms – An Overview . . . . p. 20
1.3.1. Primal Cost Improvement . . . . p. 21
1.3.2. Dual Cost Improvement . . . . p. 24
1.3.3. Auction . . . . p. 27
1.3.4. Good, Bad, and Polynomial Algorithms . . . . p. 35
1.4. Notes, Sources, and Exercises . . . . p. 37

2. Shortest Path Problems . . . . p. 51

2.1. Problem Formulation and Applications . . . . p. 52
2.2. A Generic Shortest Path Algorithm . . . . p. 57
2.3. Label Setting (Dijkstra) Methods . . . . p. 65
2.3.1. Performance of Label Setting Methods . . . . p. 68
2.3.2. The Binary Heap Method . . . . p. 69
2.3.3. Dial's Algorithm . . . . p. 70
2.4. Label Correcting Methods . . . . p. 73
2.4.1. The Bellman-Ford Method . . . . p. 73
2.4.2. The D'Esopo-Pape Algorithm . . . . p. 75
2.4.3. The SLF and LLL Algorithms . . . . p. 76
2.4.4. The Threshold Algorithm . . . . p. 78
2.4.5. Comparison of Label Setting and Label Correcting . . . p. 80
2.5. Single Origin/Single Destination Methods . . . . p. 81
2.5.1. Label Setting . . . . p. 81

2.5.2. Label Correcting . . . . p. 81
2.6. Auction Algorithms . . . . p. 86
2.7. Multiple Origin/Multiple Destination Methods . . . . p. 96
2.8. Notes, Sources, and Exercises . . . . p. 98

3. The Max-Flow Problem . . . . p. 115

3.1. The Max-Flow and Min-Cut Problems . . . . p. 116
3.1.1. Cuts in a Graph . . . . p. 119
3.1.2. The Max-Flow/Min-Cut Theorem . . . . p. 121
3.1.3. The Maximal and Minimal Saturated Cuts . . . . . . p. 123
3.1.4. Decomposition of Infeasible Network Problems . . . . p. 124
3.2. The Ford-Fulkerson Algorithm . . . . p. 125
3.3. Price-Based Augmenting Path Algorithms . . . . p. 132
3.3.1. A Price-Based Path Construction Algorithm . . . . . p. 135
3.3.2. A Price-Based Max-Flow Algorithm . . . . p. 139
3.4. Notes, Sources, and Exercises . . . . p. 139

4. The Min-Cost Flow Problem . . . . p. 151

4.1. Transformations and Equivalences . . . . p. 152
4.1.1. Setting the Lower Flow Bounds to Zero . . . . p. 152
4.1.2. Eliminating the Upper Flow Bounds . . . . p. 153
4.1.3. Reduction to a Circulation Format . . . . p. 154
4.1.4. Reduction to an Assignment Problem . . . . p. 154
4.2. Duality . . . . p. 155
4.2.1. Interpretation of CS and the Dual Problem . . . . . . p. 162
4.2.2. Duality and CS for Nonnegativity Constraints . . . . p. 163
4.3. Notes, Sources, and Exercises . . . . p. 164

5. Simplex Methods for Min-Cost Flow . . . . p. 169

5.1. Main Ideas in Simplex Methods . . . . p. 170
5.1.1. Using Prices to Obtain the In-Arc . . . . p. 176
5.1.2. Obtaining the Out-Arc . . . . p. 179
5.1.3. Dealing with Degeneracy . . . . p. 183
5.2. The Basic Simplex Algorithm . . . . p. 186
5.2.1. Termination Properties of the Simplex Method . . . . p. 187
5.2.2. Initialization of the Simplex Method . . . . p. 188
5.3. Extension to Problems with Upper and Lower Bounds . . . p. 195
5.4. Implementation Issues . . . . p. 199
5.5. Notes, Sources, and Exercises . . . . p. 203

6. Dual Ascent Methods for Min-Cost Flow . . . . p. 213

6.1. Dual Ascent . . . . p. 214

6.2. The Primal-Dual (Sequential Shortest Path) Method . . . p. 221
6.3. The Relaxation Method . . . . p. 234
6.4. Solving Variants of an Already Solved Problem . . . . . . p. 243
6.5. Implementation Issues . . . . p. 243
6.6. Notes, Sources, and Exercises . . . . p. 244

7. Auction Algorithms for Min-Cost Flow . . . . p. 251

7.1. The Auction Algorithm for the Assignment Problem . . . p. 252
7.1.1. The Main Auction Algorithm . . . . p. 253
7.1.2. Approximate Coordinate Descent Interpretation . . . p. 257
7.1.3. Variants of the Auction Algorithm . . . . p. 257
7.1.4. Computational Complexity – ϵ-Scaling . . . . p. 259
7.1.5. Dealing with Infeasibility . . . . p. 265
7.2. Extensions of the Auction Algorithm . . . . p. 268
7.2.1. Reverse Auction . . . . p. 268
7.2.2. Auction Algorithms for Asymmetric Assignment . . . p. 272
7.2.3. Auction Algorithms with Similar Persons . . . . . . p. 279
7.3. The Preflow-Push Algorithm for Max-Flow . . . . p. 282
7.3.1. Analysis and Complexity . . . . p. 285
7.3.2. Implementation Issues . . . . p. 293
7.3.3. Relation to the Auction Algorithm . . . . p. 294
7.4. The ϵ-Relaxation Method . . . . p. 304
7.4.1. Computational Complexity – ϵ-Scaling . . . . p. 310
7.4.2. Implementation Issues . . . . p. 318
7.5. The Auction/Sequential Shortest Path Algorithm . . . . . p. 320
7.6. Notes, Sources, and Exercises . . . . p. 326

8. Nonlinear Network Optimization . . . . p. 337

8.1. Convex and Separable Problems . . . . p. 339
8.2. Problems with Side Constraints . . . . p. 346
8.3. Multicommodity Flow Problems . . . . p. 349
8.4. Integer Constraints . . . . p. 355
8.5. Networks with Gains . . . . p. 360
8.6. Optimality Conditions . . . . p. 365
8.7. Duality . . . . p. 370
8.8. Algorithms and Approximations . . . . p. 375
8.8.1. Feasible Direction Methods . . . . p. 375
8.8.2. Piecewise Linear Approximation . . . . p. 380
8.8.3. Interior Point Methods . . . . p. 382
8.8.4. Penalty and Augmented Lagrangian Methods . . . . . p. 384
8.8.5. Proximal Minimization . . . . p. 386
8.8.6. Smoothing . . . . p. 387
8.8.7. Transformations . . . . p. 389

8.9. Notes, Sources, and Exercises . . . . p. 398

9. Convex Separable Network Problems . . . . p. 407

9.1. Convex Functions of a Single Variable . . . . p. 408
9.2. Optimality Conditions . . . . p. 412
9.3. Duality . . . . p. 414
9.4. Dual Function Differentiability . . . . p. 426
9.5. Algorithms for Differentiable Dual Problems . . . . p. 430
9.6. Auction Algorithms . . . . p. 433
9.6.1. The ϵ-Relaxation Method . . . . p. 441
9.6.2. Auction/Sequential Shortest Path Algorithm . . . . . p. 446
9.7. Monotropic Programming . . . . p. 449
9.8. Notes, Sources, and Exercises . . . . p. 463

10. Network Problems with Integer Constraints . . . . . p. 467

10.1. Formulation of Integer-Constrained Problems . . . . . . p. 469
10.2. Branch-and-Bound . . . . p. 483
10.3. Lagrangian Relaxation . . . . p. 492
10.3.1. Subgradients of the Dual Function . . . . p. 497
10.3.2. Subgradient Methods . . . . p. 499
10.3.3. Cutting Plane Methods . . . . p. 503
10.3.4. Decomposition and Multicommodity Flows . . . . . p. 507
10.4. Local Search Methods . . . . p. 512
10.4.1. Genetic Algorithms . . . . p. 514
10.4.2. Tabu Search . . . . p. 515
10.4.3. Simulated Annealing . . . . p. 516
10.5. Rollout Algorithms . . . . p. 517
10.6. Notes, Sources, and Exercises . . . . p. 525

Appendix A: Mathematical Review . . . . p. 545

A.1. Sets . . . . p. 546
A.2. Euclidean Space . . . . p. 547
A.3. Matrices . . . . p. 547
A.4. Analysis . . . . p. 548
A.5. Convex Sets and Functions . . . . p. 551
A.6. Subgradients . . . . p. 553

References . . . . p. 555

Index . . . . p. 587

Preface

Network optimization lies in the middle of the great divide that separates the two major types of optimization problems, continuous and discrete. The ties between linear programming and combinatorial optimization can be traced to the representation of the constraint polyhedron as the convex hull of its extreme points. When a network is involved, however, these ties become much stronger because the extreme points of the polyhedron are integer and represent solutions of combinatorial problems that are seemingly unrelated to linear programming. Because of this structure and also because of their intuitive character, network models provide ideal vehicles for explaining many of the fundamental ideas in both continuous and discrete optimization.

Aside from their interesting methodological characteristics, network models are also used extensively in practice, in an ever expanding spectrum of applications. Indeed collectively, network problems such as shortest path, assignment, max-flow, transportation, transhipment, spanning tree, matching, traveling salesman, generalized assignment, vehicle routing, and multicommodity flow constitute the most common class of practical optimization problems. There has been steady progress in the solution methodology of network problems, and in fact the progress has accelerated in the last fifteen years thanks to algorithmic and technological advances.

The purpose of this book is to provide a fairly comprehensive and up-to-date development of linear, nonlinear, and discrete network optimization problems. The interplay between continuous and discrete structures has been highlighted, the associated analytical and algorithmic issues have been treated quite extensively, and a guide to important network models and applications has been provided.

Regarding continuous network optimization, we focus on two ideas, which are also fundamental in general mathematical programming: *duality* and *iterative cost improvement*. We provide an extensive treatment of iterative algorithms for the most common linear cost problem, the minimum cost flow or transhipment problem, and for its convex cost extensions. The discussion of duality is comprehensive: it starts with linear network programming duality, and culminates with Rockafellar's development of monotropic programming duality.

Regarding discrete network optimization, we illustrate problem formulation through major paradigms such as traveling salesman, generalized assignment, spanning tree, matching, and routing. This is essential because the structure of discrete optimization problems is far less streamlined than the structure of their continuous counterparts, and familiarity with important types of problems is important for modeling, analysis, and algorithmic solution. We also develop the main algorithmic approaches, including branch-and-bound, Lagrangian relaxation, Dantzig-Wolfe decomposition, heuristics, and local search methods.

This is meant to be an introductory book that covers a very broad variety of topics. It is thus inevitable that some topics have been treated in less detail than others. The choices made reflect in part personal taste and expertise, and in part a preference for simple models that can help most effectively the reader develop insight. At the same time, our analysis and presentation aims to enhance the reader's mathematical modeling ability in two ways: by delineating the range of problems for which various algorithms are applicable and efficient, and by providing many examples of problem formulation.

The chapter-by-chapter description of the book follows:

Chapter 1: This is an introductory chapter that establishes terminology and basic notions about graphs, discusses some examples of network models, and provides some orientation regarding linear network optimization algorithms.

Chapter 2: This chapter provides an extensive treatment of shortest path problems. It covers the major methods, and discusses their theoretical and practical performance.

Chapter 3: This chapter focuses on the max-flow problem and develops the class of augmenting path algorithms for its solution. In addition to the classical variants of the Ford-Fulkerson method, a recent algorithm based on auction ideas is discussed.

Chapter 4: The minimum cost flow problem (linear cost, single commodity, no side constraints) and its equivalent variants are introduced here. Subsequently, the basic duality theory for the problem is developed and interpreted.

Chapter 5: This chapter focuses on simplex methods for the minimum cost flow problem. The basic results regarding the integrality of solutions are developed here constructively, using the simplex method. Furthermore, the duality theory of Chapter 4 is significantly strengthened.

Chapter 6: This chapter develops dual ascent methods, including primal-dual, sequential shortest path, and relaxation methods.

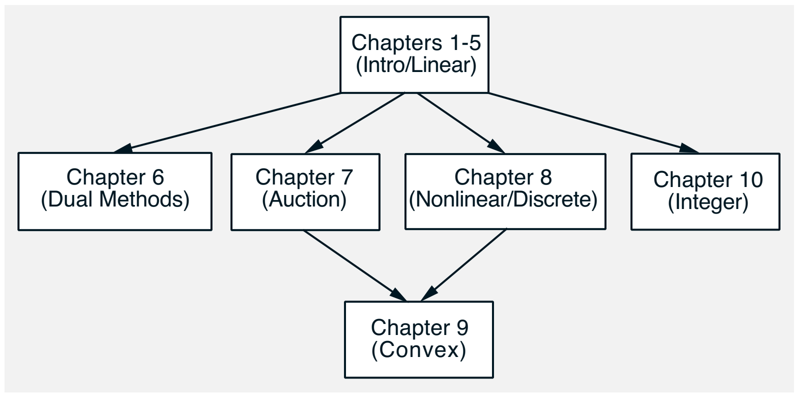
Chapter 7: This chapter starts with the auction algorithm for the assignment problem, and proceeds to show how this algorithm can be extended to more complex problems. In this way, preflow-push methods for the max-flow problem and the ϵ-relaxation method for the minimum cost flow problem are obtained. Several additional variants of auction algorithms are developed.

Chapter 8: This is an important chapter that marks the transition from linear to nonlinear network optimization. The primary focus is on continuous (convex) problems, and their associated broad variety of structures and methodology. In particular, there is an overview of the types of algorithms from nonlinear programming that are useful in connection with various convex network problems. There is also some discussion of discrete (integer) problems with an emphasis on their ties with continuous problems.

Chapter 9: This is a fairly sophisticated chapter that is directed primarily towards the advanced and/or research-oriented reader. It deals with separable convex problems, discusses their connection with classical network equilibrium problems, and develops their rich theoretical structure. The salient features of this structure are a particularly sharp duality theory, and a combinatorial connection of descent directions with the finite set of elementary vectors of the subspace defined by the conservation of flow constraints. Besides treating convex separable network problems, this chapter provides an introduction to monotropic programming, which is the largest class of nonlinear programming problems that possess the strong duality and combinatorial properties of linear programs. This chapter also develops auction algorithms for convex separable problems and provides an analysis of their running time.

Chapter 10: This chapter deals with the basic methodological approaches for integer-constrained problems. There is a treatment of exact methods such as branch-and-bound, and the associated methods of Lagrangian relaxation, subgradient optimization, and cutting plane. There is also a description of approximate methods based on local search, such as genetic algorithms, tabu search, and simulated annealing. Finally, there is a discussion of rollout algorithms, a relatively new and broadly applicable class of approximate methods, which can be used in place of, or in conjunction with local search.

The book can be used for a course on network optimization or for part of a course on introductory optimization at the first-year graduate level. With the exception of some of the material in Chapter 9, the prerequisites are fairly elementary. The main one is a certain degree of mathematical maturity, as provided for example by a rigorous mathematics course beyond the calculus level. One may cover most of the book in a course on linear and nonlinear network optimization. A shorter version of this course may consist of Chapters 1-5, and 8. Alternatively, one may teach a course that focuses on linear and discrete network optimization, using Chapters 1-5, a small part of Chapter 8, and Chapter 10. Actually, in these chapter sequences, it is not essential to cover Chapter 5, if one is content with weaker versions of duality results (given in Chapter 4) and one establishes the integrality properties of optimal solutions with a line of argument such as the one given in Exercise 1.34. The following figure illustrates the chapter dependencies.



The book contains a large number of examples and exercises, which should enhance its suitability for classroom instruction. Some of the exercises are theoretical in nature and supplement substantially the main text. Solutions to a subset of these (as well as errata and additional material) will be posted and periodically updated on the book's web page:

http://www.athenasc.com/netsbook.html

Also, the author's web page

http://web.mit.edu/dimitrib/www/home.html

contains listings of FORTRAN codes implementing many of the algorithms discussed in the book.

There is a very extensive literature on continuous and discrete network optimization, and to give a complete bibliography and a historical account of the research that led to the present form of the subject would have been impossible. Thus I have not attempted to compile a comprehensive list of original contributions to the field. I have cited sources that I have used extensively, that provide important extensions to the material of the book, that survey important topics, or that are particularly well suited for further reading. I have also cited selectively a few sources that are historically significant, but the reference list is far from exhaustive in this respect. Generally, to aid researchers in the field, I have preferred to cite surveys and textbooks for subjects that are relatively mature, and to give a larger number of references for relatively recent developments.

A substantial portion of this book is based on the author's research on network optimization over the last twenty years. I was fortunate to have several outstanding collaborators in this research, and I would like to mention those with whom I have worked extensively. Eli Gafni assisted with the computational experimentation using the auction algorithm and the relaxation method for assignment problems in 1979. The idea of ϵ-scaling arose during my interactions with Eli at that time. Furthermore, Eli collaborated extensively with me on various routing methods for data networks, including projection methods for convex multicommodity flow problems. Paul Tseng worked with me on network optimization starting in 1982. Together we developed the RELAX codes, we developed several extensions to the basic relaxation method and we collaborated closely on a broad variety of other subjects, including the recent auction algorithms for convex network problems and network problems with gains. David Castanon has worked extensively with me on a broad variety of algorithms for assignment, transportation, and minimum cost flow problems, for both serial and parallel computers, since 1987. John Tsitsiklis has been my coauthor and close collaborator for many years on a variety of optimization and large scale computation topics, including some that deal with networks. In addition to Eli, Paul, David, and John, I have had substantial research collaborations with several colleagues, the results of which have been reflected in this book. In this regard, I would like to mention Jon Eckstein, Bob Gallager, Francesca Guerriero, Roberto Musmanno, Stefano Pallottino, and Maria-Grazia Scutellà. Several colleagues proofread portions of the book, and contributed greatly with their suggestions. David Castanon, Stefano Pallottino, Steve Patek, Serap Savari, Paul Tseng, and John Tsitsiklis were particularly helpful in this regard. The research support of NSF under grants from the DDM and the CCI divisions are very much appreciated. My family has been a source of stability and loving support, without which the book would not have been written.

Dimitri P. Bertsekas
Cambridge, Mass.
Spring 1998

1

Introduction

Contents

1.1. Graphs and Flows
  1.1.1. Paths and Cycles
  1.1.2. Flow and Divergence
  1.1.3. Path Flows and Conformal Decomposition
1.2. Network Flow Models – Examples
  1.2.1. The Minimum Cost Flow Problem
  1.2.2. Network Flow Problems with Convex Cost
  1.2.3. Multicommodity Flow Problems
  1.2.4. Discrete Network Optimization Problems
1.3. Network Flow Algorithms – An Overview
  1.3.1. Primal Cost Improvement
  1.3.2. Dual Cost Improvement
  1.3.3. Auction
  1.3.4. Good, Bad, and Polynomial Algorithms
1.4. Notes, Sources, and Exercises

Network flow problems are one of the most important and most frequently encountered class of optimization problems. They arise naturally in the analysis and design of large systems, such as communication, transportation, and manufacturing networks. They can also be used to model important classes of combinatorial problems, such as assignment, shortest path, and traveling salesman problems.

Loosely speaking, network flow problems consist of supply and demand points, together with several routes that connect these points and are used to transfer the supply to the demand. These routes may contain intermediate transhipment points. Often, the supply, demand, and transhipment points can be modeled by the nodes of a graph, and the routes can be modeled by the paths of the graph. Furthermore, there may be multiple "types" of supply/demand (or "commodities") sharing the routes. There may also be some constraints on the characteristics of the routes, such as their carrying capacities, and some costs associated with using particular routes. Such situations are naturally modeled as network optimization problems whereby, roughly speaking, we try to select routes that minimize the cost of transfer of the supply to the demand.

This book deals with a broad spectrum of network optimization problems, involving linear and nonlinear cost functions. We pay special attention to four major classes of problems:

(a) The *transhipment* or *minimum cost flow problem*, which involves a single commodity and a linear cost function. This problem has several important special cases, such as the shortest path, the max-flow, the assignment, and the transportation problems.

(b) The *single commodity network flow problem with convex cost*. This problem is identical to the preceding transhipment problem, except that the cost function is convex rather than linear.

(c) The *multicommodity network flow problem with linear or convex cost*. This problem generalizes the preceding two classes of problems to the case of multiple commodities.

(d) *Discrete network optimization problems*. These are problems where the quantities transferred along the routes of the network are restricted to take one of a finite number of values. Many combinatorial optimization problems can be modeled in this way, including some problems where the network structure is not immediately apparent. Some discrete optimization problems are computationally very difficult, and in practice can only be solved approximately. Their algorithmic solution often involves the solution of "continuous" subproblems that belong to the preceding three classes.

All of the network flow problems above can be mathematically modeled in terms of graph-related notions. In Section 1.1, we introduce the associated notation and terminology. In Section 1.2, we provide mathematical formulations and practical examples of network optimization models. Finally, in Section 1.3, we give an overview of some of the types of computational algorithms that we develop in subsequent chapters.

1.1 GRAPHS AND FLOWS

In this section, we introduce some of the basic definitions relating to graphs, paths, flows, and other related notions. Graph concepts are fairly intuitive, and can be understood in terms of suggestive figures, but often involve hidden subtleties. Thus the reader may wish to revisit the present section and pay close attention to some of the fine points of the definitions.

A *directed graph*, $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, consists of a set $\mathcal{N}$ of *nodes* and a set $\mathcal{A}$ of pairs of distinct nodes from $\mathcal{N}$ called *arcs*. The numbers of nodes and arcs are denoted by N and A, respectively, and it is assumed throughout that $1 \leq N < \infty$ and $0 \leq A < \infty$. An arc (i, j) is viewed as an ordered pair, and is to be distinguished from the pair (j, i). If (i, j) is an arc, we say that (i, j) is *outgoing* from node i and *incoming* to node j; we also say that j is an *outward neighbor* of i and that i is an *inward neighbor* of j. We say that arc (i, j) is *incident* to i and to j, and that i is the *start* node and j is the *end* node of the arc. We also say that i and j are the *end nodes* of arc (i, j). The *degree* of a node i is the number of arcs that are incident to i. A graph is said to be *complete* if it contains all possible arcs; that is, if there exists an arc for each ordered pair of nodes.

We do not exclude the possibility that there is a separate arc connecting a pair of nodes in each of the two directions. However, we do not allow more than one arc between a pair of nodes in the same direction, so that we can refer unambiguously to the arc with start i and end j as arc (i, j). This is done for notational convenience.† Our analysis can be simply extended to handle multiple arcs with start i and end j; the extension is based on modifying the graph by introducing for each such arc, an additional node, call it n, together with the two arcs (i, n) and (n, j). On occasion, we will pause to provide examples of this type of extension.

We note that much of the literature of graph theory distinguishes between *directed* graphs where an arc (i, j) is an ordered pair to be distinguished from arc (j, i), and *undirected* graphs where an arc is associated with a pair of nodes regardless of order. One may use directed graphs, even in contexts where the use of undirected graphs would be appropriate and conceptually simpler. For this, one may need to replace an undirected arc (i, j) with two directed arcs (i, j) and (j, i) having identical characteristics.

† Some authors use a single symbol, such as a, to denote an arc, and use something like $s(a)$ and $e(a)$ to denote the start and end nodes of a, respectively. This notational method allows the existence of multiple arcs with the same start and end nodes, but is also more cumbersome and less suggestive.

We have chosen to deal exclusively with directed graphs because in our development there are only a few occasions where undirected graphs are convenient. Thus, *all our references to a graph implicitly assume that the graph is directed.* In fact we often omit the qualifier "directed" and refer to a directed graph simply as a *graph.*

1.1.1 Paths and Cycles

A *path* P in a directed graph is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ with $k \geq 2$ and a corresponding sequence of $k-1$ arcs such that the ith arc in the sequence is either (n_i, n_{i+1}) (in which case it is called a *forward* arc of the path) or (n_{i+1}, n_i) (in which case it is called a *backward* arc of the path). Nodes n_1 and n_k are called the *start node* (or *origin*) and the *end node* (or *destination*) of P, respectively. A path is said to be *forward* (or *backward*) if all of its arcs are forward (respectively, backward) arcs. We denote by P^+ and P^- the sets of forward and backward arcs of P, respectively.

A *cycle* is a path for which the start and end nodes are the same. A path is said to be *simple* if it contains no repeated arcs and no repeated nodes, except that the start and end nodes could be the same (in which case the path is called a *simple cycle*). A *Hamiltonian cycle* is a simple forward cycle that contains all the nodes of the graph. These definitions are illustrated in Fig. 1.1. We mention that some authors use a slightly different terminology: they use the term "walk" to refer to a path and they use the term "path" to refer to a simple path.

Note that the sequence of nodes $(n_1, n_2, \ldots, n_k)$ is not sufficient to specify a path; the sequence of arcs may also be important, as Fig. 1.1(c) shows. The difficulty arises when for two successive nodes n_i and n_{i+1} of the path, both (n_i, n_{i+1}) and (n_{i+1}, n_i) are arcs, so there is ambiguity as to which of the two is the corresponding arc of the path. If a path is known to be forward or is known to be backward, it is uniquely specified by the sequence of its nodes. Otherwise, however, the intended sequence of arcs must be explicitly defined.

A graph that contains no simple cycles is said to be *acyclic*. A graph is said to be *connected* if for each pair of nodes i and j, there is a path starting at i and ending at j; it is said to be *strongly connected* if for each pair of nodes i and j, there is a forward path starting at i and ending at j. Thus, for example, the graph of Fig. 1.1(b) is connected but not strongly connected. It can be shown that if a graph is connected and each of its nodes has even degree, there is a cycle (not necessarily forward) that contains all the arcs of the graph exactly once (see Exercise 1.5). Such a cycle is called an *Euler cycle*, honoring the historically important work of Euler; see the discussion in Section 10.1 about the Königsberg bridge problem. Figure 1.2 gives an example of an Euler cycle.

We say that a graph $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$ is a *subgraph* of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ if $\mathcal{N}' \subset \mathcal{N}$ and $\mathcal{A}' \subset \mathcal{A}$. A *tree* is a connected acyclic graph. A *spanning*

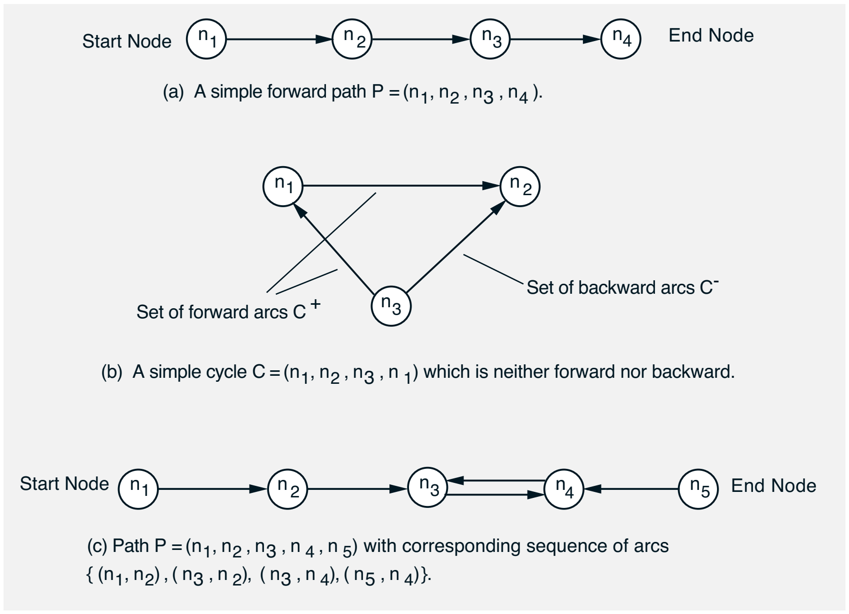

Start Node
End Node
(a) A simple forward path P = (n1, n2, n3, n4).
Set of forward arcs C+
Set of backward arcs C-
(b) A simple cycle C = (n1, n2, n3, n1) which is neither forward nor backward.
Start Node
End Node
(c) Path P = (n1, n2, n3, n4, n5) with corresponding sequence of arcs {(n1, n2), (n3, n2), (n3, n4), (n5, n4)}.

Figure 1.1: Illustration of various types of paths and cycles. The cycle in (b) is not a Hamiltonian cycle; it is simple and contains all the nodes of the graph, but it is not forward. Note that for the path (c), in order to resolve ambiguities, it is necessary to specify the sequence of arcs of the path (rather than just the sequence of nodes) because both (n_3, n_4) and (n_4, n_3) are arcs.

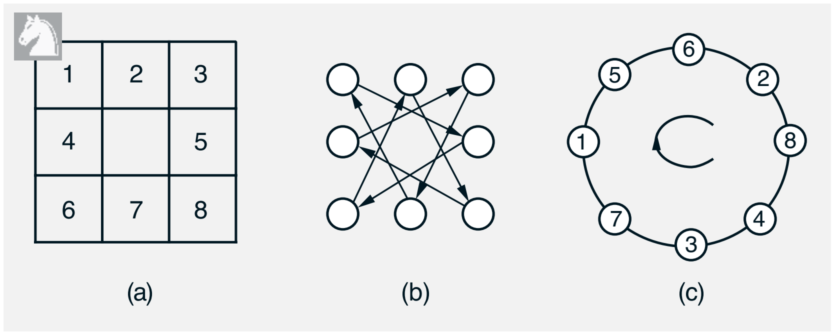


Figure 1.2: Example of an Euler cycle. Consider a 3×3 chessboard, where the middle square has been deleted. A knight starting at one of the squares of the board can visit every other square exactly once and return to the starting square as shown in the graph (b), or equivalently in (c). In the process, the knight will make all the possible moves (in one direction only), or equivalently, it will cross every arc of the graph in (b) exactly once. The knight's tour is an Euler cycle for the graph of (b).

tree of a graph $\mathcal{G}$ is a subgraph of $\mathcal{G}$, which is a tree and includes all the nodes of $\mathcal{G}$. It can be shown [Exercise 1.14(c)] that a subgraph is a spanning tree if and only if it is connected and it contains $N - 1$ arcs.

1.1.2 Flow and Divergence

In many applications involving graphs, it is useful to introduce a variable that measures the quantity flowing through each arc, like for example, electric current in an electric circuit, or water flow in a hydraulic network. We refer to such a variable as the *flow of an arc*. Mathematically, the flow of an arc (i, j) is simply a scalar (real number), which we usually denote by x_{ij}. It is convenient to allow negative as well as positive values for flow. In applications, a negative arc flow indicates that whatever is represented by the flow (material, electric current, etc.), moves in a direction opposite to the direction of the arc. We can always change the sign of a negative arc flow to positive as long as we change the arc direction, so in many situations we can assume without loss of generality that all arc flows are nonnegative. For the development of a general methodology, however, this device is often cumbersome, which is why we prefer to simply accept the possibility of negative arc flows.

Given a graph $(\mathcal{N}, \mathcal{A})$, a set of flows $\{x_{ij} \mid (i, j) \in \mathcal{A}\}$ is referred to as a *flow vector*. The *divergence vector* y associated with a flow vector x is the N-dimensional vector with coordinates

$$y_i = \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji}, \qquad \forall\ i \in \mathcal{N}. \tag{1.1}$$

Thus, y_i is the total flow departing from node i less the total flow arriving at i; it is referred to as the *divergence of* i.

We say that node i is a *source* (respectively, *sink*) for the flow vector x if $y_i > 0$ (respectively, $y_i < 0$). If $y_i = 0$ for all $i \in \mathcal{N}$, then x is called a *circulation*. These definitions are illustrated in Fig. 1.3. Note that by adding Eq. (1.1) over all $i \in \mathcal{N}$, we obtain

$$\sum_{i \in \mathcal{N}} y_i = 0.$$

Every divergence vector y must satisfy this equation.

The flow vectors x that we will consider will often be constrained to lie between given lower and upper bounds of the form

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\ (i, j) \in \mathcal{A}.$$

Given a flow vector x that satisfies these bounds, we say that a path P is *unblocked with respect to* x if, roughly speaking, we can send some positive flow along P without violating the bound constraints; that is, if flow can

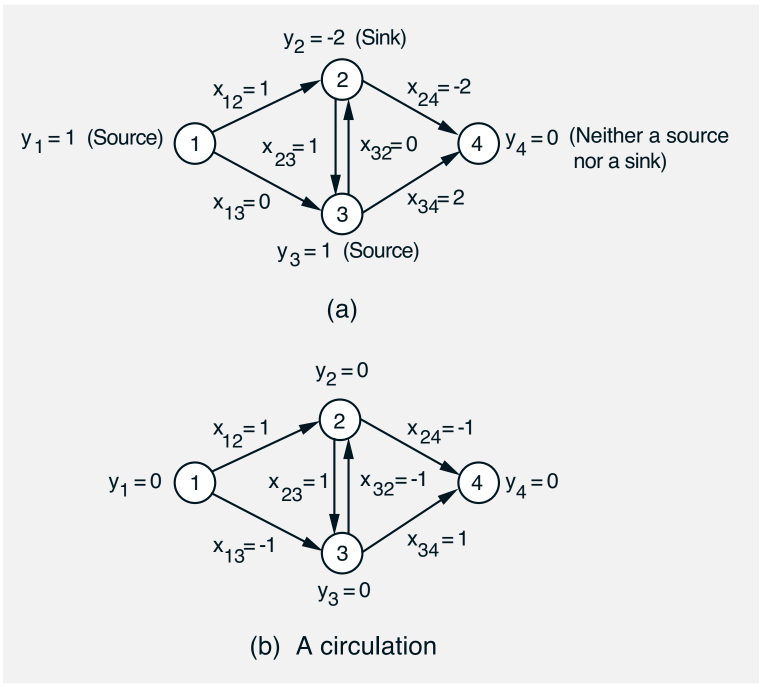


Figure 1.3: Illustration of flows x_{ij} and the corresponding divergences y_i. The flow in (b) is a circulation because $y_i = 0$ for all i.

be increased on the set P^+ of the forward arcs of P, and can be decreased on the set P^- of the backward arcs of P:

$$x_{ij} < c_{ij}, \quad \forall\ (i,j) \in P^+, \qquad b_{ij} < x_{ij}, \quad \forall\ (i,j) \in P^-.$$

For example, in Fig. 1.3(a), suppose that all arcs (i,j) have flow bounds $b_{ij} = -2$ and $c_{ij} = 2$. Then the path consisting of the sequence of nodes $(1,2,4)$ is unblocked, while the reverse path $(4,2,1)$ is not unblocked.

1.1.3 Path Flows and Conformal Decomposition

A *simple path flow* is a flow vector that corresponds to sending a positive amount of flow along a simple path; more precisely, it is a flow vector x with components of the form

$$x_{ij} = \begin{cases} a & \text{if } (i,j) \in P^+, \\ -a & \text{if } (i,j) \in P^-, \\ 0 & \text{otherwise,} \end{cases} \tag{1.2}$$

where a is a positive scalar, and P^+ and P^- are the sets of forward and backward arcs, respectively, of some simple path P. Note that the path P may be a cycle, in which case x is also called a *simple cycle flow*.

It is often convenient to break down a flow vector into the sum of simple path flows. This leads to the notion of a conformal realization, which we proceed to discuss.

We say that a path P *conforms* to a flow vector x if $x_{ij} > 0$ for all forward arcs (i, j) of P and $x_{ij} < 0$ for all backward arcs (i, j) of P, and furthermore either P is a cycle or else the start and end nodes of P are a source and a sink of x, respectively. Roughly, a path conforms to a flow vector if it "carries flow in the forward direction," i.e., in the direction from the start node to the end node. In particular, for a forward cycle to conform to a flow vector, all its arcs must have positive flow. For a forward path which is not a cycle to conform to a flow vector, its arcs must have positive flow, and in addition the start and end nodes must be a source and a sink, respectively; for example, in Fig. 1.3(a), the path consisting of the sequence of arcs (1,2), (2,3), (3,4) does not conform to the flow vector shown, because node 4, the end node of the path, is not a sink.

We say that a simple path flow x^s *conforms* to a flow vector x if the path P corresponding to x^s via Eq. (1.2) conforms to x. This is equivalent to requiring that

$$\begin{aligned} 0 < x_{ij} \qquad & \text{for all arcs } (i, j) \text{ with } 0 < x^s_{ij}, \\ x_{ij} < 0 \qquad & \text{for all arcs } (i, j) \text{ with } x^s_{ij} < 0, \end{aligned}$$

and that either P is a cycle or else the start and end nodes of P are a source and a sink of x, respectively.

An important fact is that any flow vector can be decomposed into a set of conforming simple path flows, as illustrated in Fig. 1.4. We state this as a proposition. The proof is based on an algorithm that can be used to construct the conforming components one by one (see Exercise 1.2).

Proposition 1.1: (Conformal Realization Theorem) A nonzero flow vector x can be decomposed into the sum of t simple path flow vectors $x^1, x^2, \ldots, x^t$ that conform to x, with t being at most equal to the sum of the numbers of arcs and nodes $A + N$. If x is integer, then $x^1, x^2, \ldots, x^t$ can also be chosen to be integer. If x is a circulation, then $x^1, x^2, \ldots, x^t$ can be chosen to be simple cycle flows, and $t \leq A$.

1.2 NETWORK FLOW MODELS – EXAMPLES

In this section we introduce some of the major classes of problems that will be discussed in this book. We begin with the *minimum cost flow problem*, which, together with its special cases, will be the subject of the following six chapters.

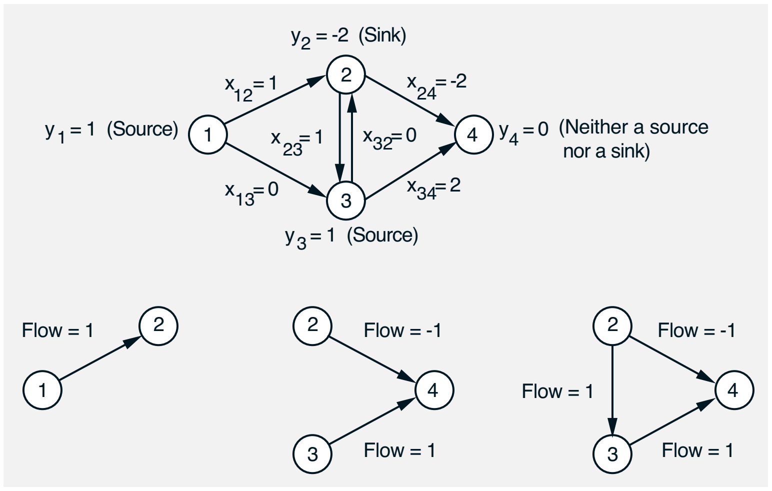


Figure 1.4: Decomposition of a flow vector x into three simple path flows conforming to x. Consistent with the definition of conformance of a path flow, each arc (i,j) of the three component paths carries positive (or negative) flow only if $x_{ij} > 0$ (or $x_{ij} < 0$, respectively). The first two paths $[(1,2)$ and $(3,4,2)]$ are not cycles, but they start at a source and end at a sink, as required. Arcs $(1,3)$ and $(3,2)$ do not belong to any of these paths because they carry zero flow. In this example, the decomposition is unique, but in general this need not be the case.

1.2.1 The Minimum Cost Flow Problem

This problem is to find a set of arc flows that minimize a linear cost function, subject to the constraints that they produce a given divergence vector and they lie within some given bounds; that is,

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \tag{1.3}$$

subject to the constraints

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N}, \tag{1.4}$$

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{1.5}$$

where a_{ij}, b_{ij}, c_{ij}, and s_i are given scalars. We use the following terminology:

a_{ij}: the *cost coefficient* (or simply *cost*) of (i,j),

b_{ij} and c_{ij}: the *flow bounds* of (i,j),

$[b_{ij}, c_{ij}]$: the *feasible flow range* of (i,j),

s_i: the *supply* of node i (when s_i is negative, the scalar $-s_i$ is called the *demand* of i).

We also refer to the constraints (1.4) and (1.5) as the *conservation of flow constraints*, and the *capacity constraints*, respectively. A flow vector satisfying both of these constraints is called *feasible*, and if it satisfies just the capacity constraints, it is called *capacity-feasible*. If there exists at least one feasible flow vector, the minimum cost flow problem is called *feasible*; otherwise it is called *infeasible*. On occasion, we will consider the variation of the minimum cost flow problem where the lower or the upper flow bound of some of the arcs is either $-\infty$ or ∞, respectively. In these cases, we will explicitly state so.

For a typical application of the minimum cost flow problem, think of the nodes as locations (cities, warehouses, or factories) where a certain product is produced or consumed. Think of the arcs as transportation links between the locations, each with transportation cost a_{ij} per unit transported. The problem then is to move the product from the production points to the consumption points at minimum cost while observing the capacity constraints of the transportation links.

However, the minimum cost flow problem has many applications that are well beyond the transportation context just described, as will be seen from the following examples. These examples illustrate how some important discrete/combinatorial problems can be modeled as minimum cost flow problems, and highlight the important connection between continuous and discrete network optimization.

Example 1.1. The Shortest Path Problem

Suppose that each arc (i,j) of a graph is assigned a scalar cost a_{ij}, and suppose that we define the cost of a forward path to be the sum of the costs of its arcs. Given a pair of nodes, the shortest path problem is to find a forward path that connects these nodes and has minimum cost. An analogy here is made between arcs and their costs, and roads in a transportation network and their lengths, respectively. Within this transportation context, the problem becomes one of finding the shortest route between two geographical points. Based on this analogy, the problem is referred to as the *shortest path problem*, and the arc costs and path costs are commonly referred to as the *arc lengths* and *path lengths*, respectively.

The shortest path problem arises in a surprisingly large number of contexts. For example in a data communication network, a_{ij} may denote the average delay of a packet to cross the communication link (i,j), in which case a shortest path is a minimum average delay path that can be used for routing the packet from its origin to its destination. As another example, if p_{ij} is the probability that a given arc (i,j) in a communication network is usable, and each arc is usable independently of all other arcs, then the product of the probabilities of the arcs of a path provides a measure of reliability of the path. With this in mind, it is seen that finding the most reliable path connecting two nodes is equivalent to finding the shortest path between the two nodes with arc lengths $(-\ln p_{ij})$.

The shortest path problem also arises often as a subroutine in algorithms that solve other more complicated problems. Examples are the primal-dual algorithm for solving the minimum cost flow problem (see Chapter 6), and the conditional gradient and projection algorithms for solving multicommodity flow problems (see Chapter 8).

It is possible to cast the problem of finding a shortest path from node s to node t as the following minimum cost flow problem:

$$\begin{aligned} &\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} \quad \sum_{\{j\,|\,(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\,|\,(j,i)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1 & \text{if } i=s, \\ -1 & \text{if } i=t, \\ 0 & \text{otherwise,} \end{cases} \\ &\qquad\qquad 0 \le x_{ij}, \qquad \forall\ (i,j)\in\mathcal{A}. \end{aligned} \tag{1.6}$$

To see this, let us associate with any forward path P from s to t the flow vector x with components given by

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ belongs to } P, \\ 0 & \text{otherwise.} \end{cases} \tag{1.7}$$

Then x is feasible for problem (1.6) and the cost of x is equal to the length of P. Thus, if a vector x of the form (1.7) is an optimal solution of problem (1.6), the corresponding path P is shortest.

Conversely, it can be shown that if problem (1.6) has at least one optimal solution, then it has an optimal solution of the form (1.7), with a corresponding path P that is shortest. This is not immediately apparent, but its proof can be traced to a remarkable fact that we will show in Chapter 5 about minimum cost flow problems with node supplies and arc flow bounds that are integer: such problems, if they have an optimal solution, they have an *integer* optimal solution, that is, a set of optimal arc flows that are integer (an alternative proof of this fact is sketched in Exercise 1.34). From this it follows that if problem (1.6) has an optimal solution, it has one with arc flows that are 0 or 1, and which is of the form (1.7) for some path P. This path is shortest because its length is equal to the optimal cost of problem (1.6), so it must be less or equal to the cost of any other flow vector of the form (1.7), and therefore also less or equal to the length of any other path from s to t. Thus the shortest path problem is essentially equivalent with the minimum cost flow problem (1.6).

Example 1.2. The Assignment Problem

Suppose that there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit or value a_{ij} for matching person i with object j, and we want to assign persons to objects so as to maximize the total

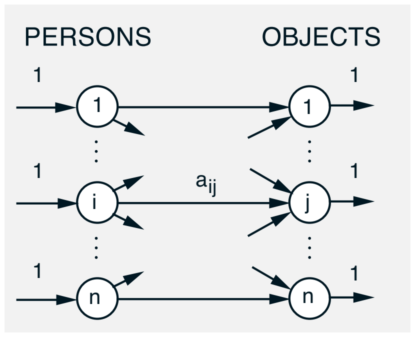


Figure 1.5: The graph representation of an assignment problem.

benefit. There is also a restriction that person i can be assigned to object j only if (i, j) belongs to a given set of pairs $\mathcal{A}$. Mathematically, we want to find a set of person-object pairs $(1, j_1), \ldots, (n, j_n)$ from $\mathcal{A}$ such that the objects $j_1, \ldots, j_n$ are all distinct, and the total benefit $\sum_{i=1}^{n} a_{ij_i}$ is maximized.

The assignment problem is important in many practical contexts. The most obvious ones are resource allocation problems, such as assigning employees to jobs, machines to tasks, etc. There are also situations where the assignment problem appears as a subproblem in methods for solving various complex combinatorial problems (see Chapter 10).

We may associate any assignment with the set of variables $\{x_{ij} \mid (i, j) \in \mathcal{A}\}$, where $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise. The value of this assignment is $\sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$. The restriction of one object per person can be stated as $\sum_j x_{ij} = 1$ for all i and $\sum_i x_{ij} = 1$ for all j. We may then formulate the assignment problem as the linear program

$$\begin{aligned}
&\text{maximize} && \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\
&\text{subject to} && \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\ i = 1, \ldots, n, \\
& && \sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\ j = 1, \ldots, n, \\
& && 0 \le x_{ij} \le 1, \qquad \forall\ (i, j) \in \mathcal{A}.
\end{aligned} \tag{1.8}$$

Actually we should further restrict x_{ij} to be either 0 or 1. However, as we will show in Chapter 5, the above linear program has the property that if it has a feasible solution at all, then it has an optimal solution where all x_{ij} are either 0 or 1 (compare also with the discussion in the preceding example and Exercise 1.34). In fact, the set of its optimal solutions includes all the optimal assignments.

We now argue that the assignment/linear program (1.8) is a minimum cost flow problem involving the graph shown in Fig. 1.5. Here, there are $2n$ nodes divided into two groups: n corresponding to persons and n corresponding to objects. Also, for every possible pair $(i, j) \in \mathcal{A}$, there is an arc connecting person i with object j. The variable x_{ij} is the flow of arc (i, j).

The constraint

$$\sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} = 1$$

indicates that the divergence of person/node i should be equal to 1, while the constraint

$$\sum_{\{i \mid (i,j) \in \mathcal{A}\}} x_{ij} = 1$$

indicates that the divergence of object/node j should be equal to -1. Finally, we may view $(-a_{ij})$ as the cost coefficient of the arc (i,j) (by reversing the sign of a_{ij}, we convert the problem from a maximization to a minimization problem).

Example 1.3. The Max-Flow Problem

In the max-flow problem, we have a graph with two special nodes: the *source*, denoted by s, and the *sink*, denoted by t. Roughly, the objective is to move as much flow as possible from s into t while observing the capacity constraints. More precisely, we want to find a flow vector that makes the divergence of all nodes other than s and t equal to 0 while maximizing the divergence of s.

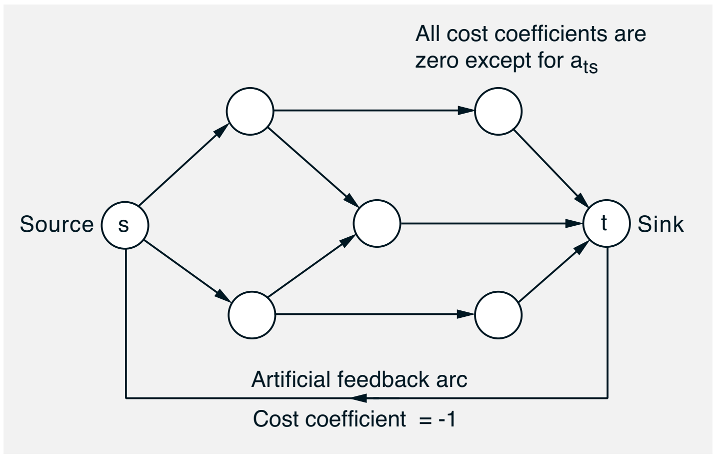


Figure 1.6: The minimum cost flow representation of a max-flow problem. At the optimum, the flow x_{ts} equals the maximum flow that can be sent from s to t through the subgraph obtained by deleting the artificial arc (t,s).

The max-flow problem arises in many practical contexts, such as calculating the throughput of a highway system or a communication network. It also arises often as a subproblem in more complicated problems or algorithms; in particular, it bears a fundamental connection to the question of existence of a feasible solution of a general minimum cost flow problem (see our discussion in Chapter 3). Finally, several discrete/combinatorial optimization problems can be formulated as max-flow problems (see the Exercises in Chapter 3).

We formulate the problem as a special case of the minimum cost flow problem by assigning cost 0 to all arcs and by introducing an artificial arc (t, s) with cost -1, as shown in Fig. 1.6. Mathematically, the problem is:

$$\begin{aligned}
&\text{maximize} \quad x_{ts} \\
&\text{subject to} \\
&\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} = 0, \qquad \forall\, i \in \mathcal{N} \text{ with } i \neq s \text{ and } i \neq t, \\
&\sum_{\{j \mid (s,j)\in\mathcal{A}\}} x_{sj} = \sum_{\{i \mid (i,t)\in\mathcal{A}\}} x_{it} = x_{ts}, \\
&b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\, (i,j) \in \mathcal{A} \text{ with } (i,j) \neq (t,s).
\end{aligned}$$

Viewing the problem as a maximization is consistent with its intuitive interpretation. Alternatively, we could write the problem as a minimization of $-x_{ts}$ subject to the same constraints. Also, we could introduce upper and lower bounds on x_{ts},

$$\sum_{\{i \mid (i,t)\in\mathcal{A}\}} b_{it} \le x_{ts} \le \sum_{\{i \mid (i,t)\in\mathcal{A}\}} c_{it},$$

but these bounds are actually redundant since they are implied by the other upper and lower arc flow bounds.

Example 1.4. The Transportation Problem

This problem is the same as the assignment problem except that the node supplies need not be 1 or -1, and the numbers of sources and sinks need not be equal. It has the form

$$\begin{aligned}
&\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} \\
&\text{subject to} \quad \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = \alpha_i, \qquad \forall\, i = 1, \ldots, m, \\
&\qquad\qquad \sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} = \beta_j, \qquad \forall\, j = 1, \ldots, n, \\
&\qquad\qquad 0 \le x_{ij} \le \min\{\alpha_i, \beta_j\}, \qquad \forall\, (i,j) \in \mathcal{A}.
\end{aligned} \tag{1.9}$$

Here α_i and β_j are positive scalars, which for feasibility must satisfy

$$\sum_{i=1}^{m} \alpha_i = \sum_{j=1}^{n} \beta_j,$$

(add the conservation of flow constraints). In an alternative formulation, the upper bound constraint $x_{ij} \leq \min\{\alpha_i, \beta_j\}$ could be discarded, since it is implied by the conservation of flow and the nonnegativity constraints.

As a practical example of a transportation problem that has a combinatorial flavor, suppose that we have m communication terminals, each to be connected to one of n traffic concentrators. We introduce variables x_{ij}, which take the value 1 if terminal i is connected to concentrator j. Assuming that concentrator j can be connected to no more than b_j terminals, we obtain the constraints

$$\sum_{i=1}^{m} x_{ij} \leq b_j, \qquad \forall\ j = 1, \ldots, n.$$

Also, since each terminal must be connected to exactly one concentrator, we have the constraints

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \forall\ i = 1, \ldots, m.$$

Assuming that there is a cost a_{ij} for connecting terminal i to concentrator j, the problem is to find the connection of minimum cost, that is, to minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_{ij}$$

subject to the preceding constraints. This problem is not yet a transportation problem of the form (1.9) for two reasons:

(a) The arc flows x_{ij} are constrained to be 0 or 1.

(b) The constraints $\sum_{i=1}^{m} x_{ij} \leq b_j$ are not equality constraints, as required in problem (1.9).

It turns out, however, that we can ignore the 0-1 constraint on x_{ij}. As discussed in connection with the shortest path and assignment problems, even if we relax this constraint and replace it with the capacity constraint $0 \leq x_{ij} \leq 1$, there is an optimal solution such that each x_{ij} is either 0 or 1. Furthermore, to convert the inequality constraints to equalities, we can introduce a total of $\sum_{j=1}^{n} b_j - m$ "dummy" terminals that can be connected at zero cost to all of the concentrators. In particular, we introduce a special supply node 0 together with the constraint

$$\sum_{j=1}^{n} x_{0j} = \sum_{j=1}^{n} b_j - m,$$

and we change the inequality constraints $\sum_{j=1}^{n} x_{ij} \leq b_j$ to

$$x_{0j} + \sum_{i=1}^{m} x_{ij} = b_j.$$

The resulting problem has the transportation structure of problem (1.9), and is equivalent to the original problem.

1.2.2 Network Flow Problems with Convex Cost

A more general version of the minimum cost flow problem arises when the cost function is convex rather than linear. An important special case is the problem

$$\begin{aligned} &\text{minimize} && \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) \\ &\text{subject to} && \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N}, \\ &&& x_{ij} \in X_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \end{aligned}$$

where f_{ij} is a convex function of the flow x_{ij} of arc (i,j), s_i are given scalars, and X_{ij} are convex intervals of real numbers, such as for example

$$X_{ij} = [b_{ij}, c_{ij}],$$

where b_{ij} and c_{ij} are given scalars. We refer to this as the *separable convex cost network flow problem*, because the cost function separates into the sum of cost functions, one per arc. This problem will be discussed in detail in Chapters 8 and 9.

Example 1.5. The Matrix Balancing Problem

Here the problem is to find an $m \times n$ matrix X that has given row sums and column sums, and approximates a given $m \times n$ matrix M in some optimal manner. We can formulate such a problem in terms of a graph consisting of m sources and n sinks. In this graph, the set of arcs consists of the pairs (i,j) for which the corresponding entry x_{ij} of the matrix X is allowed to be nonzero. The given row sums r_i and the given column sums c_j are expressed as the constraints

$$\sum_{\{j \mid (i,j)\in A\}} x_{ij} = r_i, \qquad i = 1, \ldots, m,$$

$$\sum_{\{i \mid (i,j)\in A\}} x_{ij} = c_j, \qquad j = 1, \ldots, n.$$

There may be also bounds for the entries x_{ij} of X. Thus, the structure of this problem is similar to the structure of a transportation problem. The cost function to be optimized has the form

$$\sum_{(i,j)\in A} f_{ij}(x_{ij}),$$

and expresses the objective of making the entries of X close to the corresponding entries of the given matrix M. A commonly used example is the quadratic function

$$f_{ij}(x_{ij}) = \sum_{(i,j)\in A} w_{ij}(x_{ij} - m_{ij})^2,$$

where w_{ij} are given positive scalars.

Another interesting cost function is the logarithmic

$$f_{ij}(x_{ij}) = x_{ij}\left[\ln\left(\frac{x_{ij}}{m_{ij}}\right) - 1\right],$$

where we assume that $m_{ij} > 0$ for all $(i,j) \in A$. Note that this function is not defined for $x_{ij} \le 0$, so to obtain a problem that fits our framework, we must use a constraint interval of the form $X_{ij} = (0,\infty)$ or $X_{ij} = (0, c_{ij}]$, where c_{ij} is a positive scalar.

An example of a practical problem that can be addressed using the preceding optimization model is to predict the distribution matrix X of telephone traffic between m origins and n destinations. Here we are given the total supplies r_i of the origins and the total demands c_j of the destinations, and we are also given some matrix M that defines a nominal traffic pattern obtained from historical data.

There are other types of network flow problems with convex cost that often arise in practice. We generically represent such problems in the form

$$\begin{aligned} &\text{minimize} \;\; f(x) \\ &\text{subject to} \;\; x \in F \end{aligned}$$

where F is a convex subset of flow vectors in a graph and f is a convex function over the set F. We will discuss in some detail various classes of problems of this type in Chapter 8, and we will see that they arise in several different ways; for example, the cost function may be *nonseparable* because of coupling of the costs of several arc flows, and/or there may be *side constraints*, whereby the flows of several arcs are jointly restricted by the availability of resource. An important example is multicommodity flow problems, which we discuss next.

1.2.3 Multicommodity Flow Problems

Multicommodity network flow problems involve several flow "types" or *commodities*, which simultaneously use the network and are coupled through either the arc flow bounds, or through the cost function. Important examples of such problems arise in communication, transportation, and manufacturing networks. For example, in communication networks the commodities are the streams of different classes of traffic (telephone, data, video, etc.) that involve different origin-destination pairs. Thus there is a separate commodity per class of traffic and origin-destination pair. The following example introduces this context. In Chapter 8, we will discuss similar and/or more general multicommodity network flow problems that arise in other practical contexts.

Example 1.6. Routing in Data Networks

We are given a directed graph, which is viewed as a model of a data communication network. We are also given a set of ordered node pairs (i_m, j_m), $m = 1, \dots, M$, referred to as *origin-destination (OD) pairs*. The nodes i_m and j_m are referred to as the *origin* and the *destination* of the OD pair. For each OD pair (i_m, j_m), we are given a scalar r_m that represents its input traffic. In the context of routing of data in a communication network, r_m (measured for example in bits/second) is the arrival rate of traffic entering the network at node i_m and exiting at node j_m. The routing objective is to divide each r_m among the many paths from the origin i_m to the destination j_m in a way that the resulting total arc flow pattern minimizes a suitable cost function (see Fig. 1.7).

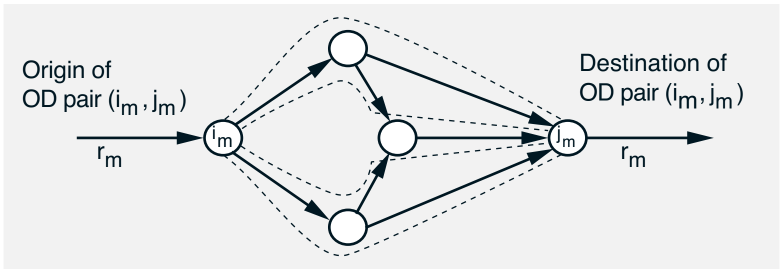


Figure 1.7: Illustration of how the input r_m of the OD pair (i_m, j_m) is divided into nonnegative path flows that start at i_m and end at j_m. The flows of the different OD pairs interact by sharing the arcs of the network.

If we denote by $x_{ij}(m)$ the flow on arc (i, j) of OD pair (i_m, j_m), we have the conservation of flow constraints

$$\sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij}(m) - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji}(m) = \begin{cases} r_m & \text{if } i = i_m, \\ -r_m & \text{if } i = j_m, \\ 0 & \text{otherwise}, \end{cases} \qquad \forall\, i \in \mathcal{N},$$

for each $m = 1, \dots, M$. Furthermore, the flows $x_{ij}(m)$ are required to be nonnegative, and possibly to satisfy additional constraints, such as upper bounds. The cost function often has the form

$$f(x) = \sum_{(i,j) \in \mathcal{A}} f_{ij}(y_{ij}),$$

where f_{ij} is a function of the *total flow* of arc (i,j)

$$y_{ij} = \sum_{m=1}^{M} x_{ij}(m).$$

Such a cost function is often based on a queueing model of average delay (see for example the data network textbook by Bertsekas and Gallager [1992]).

1.2.4 Discrete Network Optimization Problems

Many linear or convex network flow problems, in addition to the conservation of flow constraints and arc flow bounds, involve some additional constraints. In particular, there may be constraints that couple the flows of different arcs, and there may also be *integer constraints* on the arc flows, such as for example that each arc flow be either 0 or 1. Several famous combinatorial optimization problems, such as the following one, are of this type.

Example 1.7. The Traveling Salesman Problem

This problem refers to a salesman who wants to find a minimum mileage/cost tour that visits each of N given cities exactly once and returns to the city he started from. To convert this to a network flow problem, we associate a node with each city $i = 1, \ldots, N$, and we introduce an arc (i,j) with traversal cost a_{ij} for each ordered pair of nodes i and j. A *tour* is synonymous to a Hamiltonian cycle, which was earlier defined to be a simple forward cycle that contains all the nodes of the graph. Equivalently, a tour is a connected subgraph that consists of N arcs, such that there is exactly one incoming and one outgoing arc for each node $i = 1, \ldots, N$. The problem is to find a tour with minimum sum of arc costs.

To formulate this problem as a network flow problem, we denote by x_{ij} the flow of arc (i,j) and we require that this flow is either 1 or 0, indicating that the arc is or is not part of the tour, respectively. The cost of a tour T is then

$$\sum_{(i,j)\in T} a_{ij} x_{ij}.$$

The constraint that each node has a single incoming and a single outgoing arc on the tour is expressed by the following two conservation of flow equations:

$$\sum_{\substack{j=1,\ldots,N \\ j\neq i}} x_{ij} = 1, \qquad i = 1, \ldots, N,$$

$$\sum_{\substack{i=1,\ldots,N \\ i\neq j}} x_{ij} = 1, \qquad j = 1, \ldots, N.$$

There is one additional connectivity constraint:

the subgraph with node set $\mathcal{N}$ and arc set $\{(i,j) \mid x_{ij} = 1\}$ is connected.

If this constraint was not present, the problem would be an ordinary assignment problem. Unfortunately, this constraint is essential, since without it, there would be feasible solutions involving multiple disconnected cycles.

Despite the similarity, the traveling salesman problem is far more difficult than the assignment problem. Solving problems having a mere few hundreds of nodes can be very challenging. By contrast, assignment problems with hundreds of thousands of nodes can be solved in reasonable time with the presently available methodology.

Actually, we have already described some discrete/combinatorial problems that fall within the framework of the minimum cost flow problem, such as shortest path and assignment (cf. Examples 1.1 and 1.2). These problems require that the arc flows be 0 or 1, but, as mentioned earlier, we can neglect these 0-1 constraints because it turns out that even if we relax them and replace them with flow bound intervals $[0,1]$, we can obtain optimal flows that are 0 or 1 (for a proof, see Section 5.2 or Exercise 1.34).

On the other hand, once we deviate from the minimum cost flow structure and we impose additional constraints or use a nonlinear cost function, the integer character of optimal solutions is lost, and all integer constraints must be explicitly imposed. This often complicates dramatically the solution process, and in fact it may be practically impossible to obtain an exactly optimal solution. As we will discuss in Chapter 10, there are several approximate solution approaches that are based on simplified versions of the problem, such as relaxing the integer constraints. These simplified problems can often be addressed with the efficient minimum cost flow algorithms that we will develop in Chapters 2-7.

1.3 NETWORK FLOW ALGORITHMS – AN OVERVIEW

This section, which may be skipped without loss of continuity, provides a broad classification of the various classes of algorithms for linear and convex network optimization problems. It turns out that these algorithms rely on just a few basic ideas, so they can be easily grouped in a few major categories. By contrast, there is a much larger variety of algorithmic ideas for discrete optimization problems. For this reason, we postpone the corresponding discussion for Chapter 10.

Network optimization problems typically cannot be solved analytically. Usually they must be addressed computationally with one of several available algorithms. One possibility, for linear and convex problems, is to use a general purpose linear or nonlinear programming algorithm. However, the network structure can be exploited to speed up the solution by using either an adaptation of a general purpose algorithm such as the simplex method, or by using a specialized network optimization algorithm. In practice, network optimization problems can often be solved hundreds and even thousands of times faster than general linear or convex programs of comparable dimension.

The algorithms for linear and convex network problems that we will discuss in this book can be grouped in three main categories:

(a) *Primal cost improvement.* Here we try to iteratively improve the cost to its optimal value by constructing a corresponding sequence of feasible flows.

(b) *Dual cost improvement.* Here we define a problem related to the original network flow problem, called the *dual problem*, whose variables are called *prices*. We then try to iteratively improve the dual cost to its optimal value by constructing a corresponding sequence of prices. Dual cost improvement algorithms also iterate on flows, which are related to the prices through a property called *complementary slackness*.

(c) *Auction.* Here we generate a sequence of prices in a way that is reminiscent of real-life auctions. Strictly speaking, there is no primal or dual cost improvement here, although we will show that auction can be viewed as an approximate dual cost improvement process. In addition to prices, auction algorithms also iterate on flows, which are related to prices through a property called *ϵ-complementary slackness*; this is an approximate form of the complementary slackness property mentioned above.

All of the preceding types of algorithms can be used to solve both linear and convex network problems (although the structure of the given problem may favor significantly the use of some types of methods over others). For simplicity, in this chapter we will explain these ideas primarily through the assignment problem, deferring a more detailed development to subsequent chapters. Our illustrations, however, are relevant to the general minimum cost flow problem and to its convex cost extensions. Some of our explanations are informal. Precise statements of algorithms and results will be given in subsequent chapters.

1.3.1 Primal Cost Improvement

Primal cost improvement algorithms for the minimum cost flow problem start from an initial feasible flow vector and then generate a sequence of feasible flow vectors, each having a better cost than the preceding one. Let us derive an important characterization of the differences between successive vectors, which is the basis for algorithms as well as for optimality conditions.

Let x and $\overline{x}$ be two feasible flow vectors, and consider their difference $z = \overline{x} - x$. This difference must be a circulation with components

$$z_{ij} = \overline{x}_{ij} - x_{ij},$$

since both x and $\overline{x}$ are feasible. Furthermore, if the cost of $\overline{x}$ is smaller than the cost of x, the circulation z must have negative cost, i.e.,

$$\sum_{(i,j)\in\mathcal{A}} a_{ij} z_{ij} < 0.$$

We can decompose z into the sum of simple cycle flows by using the conformal realization theorem (Prop. 1.1). In particular, for some positive integer K, we have

$$z = \sum_{k=1}^{K} w^k \xi^k,$$

where w^k are positive scalars, and ξ^k are simple cycle flows whose nonzero components ξ^k_{ij} are 1 or -1, depending on whether $z_{ij} > 0$ or $z_{ij} < 0$, respectively. It is seen that the cost of z is

$$\sum_{(i,j)\in\mathcal{A}} a_{ij} z_{ij} = \sum_{k=1}^{K} w^k c^k,$$

where c^k is the cost of the simple cycle flow ξ^k. Thus, since the scalars w^k are positive, if the cost of z is negative, at least one c^k must be negative. Note that if C_k is the cycle corresponding to ξ^k, we have

$$c^k = \sum_{(i,j)\in\mathcal{A}} a_{ij} \xi^k_{ij} = \sum_{(i,j)\in C_k^+} a_{ij} - \sum_{(i,j)\in C_k^-} a_{ij},$$

where C_k^+ and C_k^- are the sets of forward and backward arcs of the cycle C_k, respectively. We refer to the expression in the right-hand side above as the *cost of the cycle* C_k.

The preceding argument has shown that *if x is feasible but not optimal, and $\overline{x}$ is feasible and has smaller cost than x, then at least one of the cycles corresponding to a conformal decomposition of the circulation $\overline{x} - x$ as above has negative cost*. This is used to prove the following important optimality condition.

Proposition 1.2: Consider the minimum cost flow problem. A flow vector x^* is optimal if and only if x^* is feasible and every simple cycle C that is unblocked with respect to x^* has nonnegative cost; that is,

$$\sum_{(i,j)\in C^+} a_{ij} - \sum_{(i,j)\in C^-} a_{ij} \geq 0.$$

Proof: Let x^* be an optimal flow vector and let C be a simple cycle that is unblocked with respect to x^*. Then there exists an $\epsilon > 0$ such that increasing (decreasing) the flow of arcs of C^+ (of C^-, respectively) by ϵ results in a feasible flow that has cost equal to the cost of x^* plus ϵ times the cost of C. Thus, since x^* is optimal, the cost of C must be nonnegative.

Conversely, suppose, to arrive at a contradiction, that x^* is feasible and has the nonnegative cycle property stated in the proposition, but is not optimal. Let $\overline{x}$ be a feasible flow vector with cost smaller that the one of x^*, and consider a conformal decomposition of the circulation $z = \overline{x} - x^*$. From the discussion preceding the proposition, we see that there is a simple cycle C with negative cost, such that $x^*_{ij} < \overline{x}_{ij}$ for all $(i,j) \in C^+$, and such that $x^*_{ij} > \overline{x}_{ij}$ for all $(i,j) \in C^-$. Since $\overline{x}$ is feasible, we have $b_{ij} \leq \overline{x}_{ij} \leq c_{ij}$ for all (i,j). It follows that $x^*_{ij} < c_{ij}$ for all $(i,j) \in C^+$, and $x^*_{ij} > b_{ij}$ for all $(i,j) \in C^-$, so that C is unblocked with respect to x^*. This contradicts the hypothesis that every simple cycle that is unblocked with respect to x^* has nonnegative cost. **Q.E.D.**

Most primal cost improvement algorithms (including for example the simplex method, to be discussed in Chapter 5) are based on the preceding proposition. They employ various mechanisms to construct negative cost cycles along which flow is pushed without violating the bound constraints. The idea of improving the cost by pushing flow along a suitable cycle often has an intuitive meaning as we illustrate in the context of the assignment problem.

Example 1.7. Multi-Person Exchanges in Assignment

Consider the $n \times n$ assignment problem (cf. Example 1.2) and suppose that we have a feasible assignment, that is, a set of n pairs (i,j) involving each person i exactly once and each object j exactly once. In order to improve this assignment, we could consider a *two-person exchange*, that is, replacing two pairs (i_1,j_1) and (i_2,j_2) from the assignment with the pairs (i_1,j_2) and (i_2,j_1). The resulting assignment will still be feasible, and it will have a higher value if and only if

$$a_{i_1j_2} + a_{i_2j_1} > a_{i_1j_1} + a_{i_2j_2}.$$

We note here that, in the context of the minimum cost flow representation of the assignment problem, a two-person exchange can be identified with a cycle involving the four arcs (i_1,j_1), (i_2,j_2), (i_1,j_2), and (i_2,j_1). Furthermore, this cycle is the difference between the assignment before and the assignment after the exchange, while the preceding inequality is equivalent to the cycle having a positive value.

Unfortunately, it may be impossible to improve the current assignment by a two-person exchange, even if the assignment is not optimal; see Fig. 1.8. An improvement, however, is possible by means of a *k-person exchange*, for some $k \geq 2$, where a set of pairs $(i_1,j_1),\ldots,(i_k,j_k)$ from the current assignment is replaced by the pairs $(i_1,j_2),\ldots,(i_{k-1},j_k),(i_k,j_1)$. To see this,

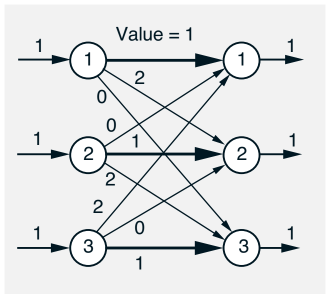


Figure 1.8: An example of a nonoptimal feasible assignment that cannot be improved by a two-person exchange. The value of each pair is shown next to the corresponding arc. Here, the value of the assignment $\{(1,1),(2,2),(3,3)\}$ is left unchanged at 3 by any two-person exchange. Through a three-person exchange, however, we obtain the optimal assignment, $\{(1,2),(2,3),(3,1)\}$, which has value 6.

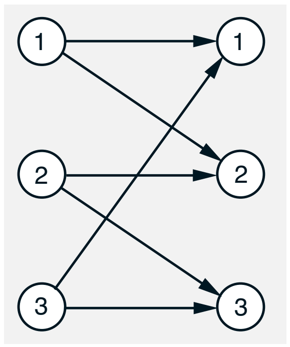


Figure 1.9: Illustration of the correspondence of a k-person exchange to a simple cycle. This is the same example as in the preceding figure. The backward arcs of the cycle are $(1,1)$, $(2,2)$, and $(3,3)$, and correspond to the current assignment pairs. The forward arcs of the cycle are $(1,2)$, $(2,3)$, and $(3,1)$, and correspond to the new assignment pairs. This three-person exchange is value-improving because the sum of the values of the forward arcs $(2+2+2)$ is greater than the sum of the values of the backward arcs $(1+1+1)$.

note that in the context of the minimum cost flow representation of the assignment problem, a k-person exchange corresponds to a simple cycle with k forward arcs (corresponding to the new assignment pairs) and k backward arcs (corresponding to the current assignment pairs that are being replaced); see Fig. 1.9. Thus, performing a k-person exchange is equivalent to pushing one unit of flow along the corresponding simple cycle. The k-person exchange improves the assignment if and only if

$$a_{i_k j_1} + \sum_{m=1}^{k-1} a_{i_m j_{m+1}} - \sum_{m=1}^{k} a_{i_m j_m},$$

which is equivalent to the corresponding cycle having positive value. Furthermore, by Prop. 1.2, a cost improving cycle exists if the flow corresponding to the current assignment is not optimal.

1.3.2 Dual Cost Improvement

Duality theory deals with the relation between the original network optimization problem and another optimization problem called the *dual*. To develop an intuitive understanding of duality, we will focus on an $n \times n$ assignment problem (cf. Example 1.2) and consider a closely related economic equilibrium problem. In particular, let us consider matching the n objects with the n persons through a market mechanism, viewing each person as an economic agent acting in his/her own best interest. Suppose that object j has a price p_j and that the person who receives the object must pay the price p_j. Then the net value of object j for person i is $a_{ij} - p_j$, and each person i will logically want to be assigned to an object j_i with maximal value, that is, with

$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{1.10}$$

where

$$A(i) = \big\{j \mid (i,j) \in \mathcal{A}\big\}$$

is the set of objects that can be assigned to person i. When this condition holds for all persons i, we say that the assignment and the price vector $p = (p_1, \ldots, p_n)$ satisfy *complementary slackness* (CS for short); this name is standard in linear programming. The economic system is then at equilibrium, in the sense that no person would have an incentive to unilaterally seek another object. Such equilibrium conditions are naturally of great interest to economists, but there is also a fundamental relation with the assignment problem. We have the following proposition.

Proposition 1.3: If a feasible assignment and a set of prices satisfy the complementary slackness condition (1.10) for all persons i, then the assignment is optimal and the prices are an optimal solution of a dual problem, which is to minimize over $p = (p_1, \ldots, p_n)$ the cost function

$$\sum_{i=1}^{n} q_i(p) + \sum_{j=1}^{n} p_j,$$

where the functions q_i are given by

$$q_i(p) = \max_{j \in A(i)} \big\{a_{ij} - p_j\big\}, \qquad i = 1, \ldots, n.$$

Furthermore, the value of the optimal assignment and the optimal cost of the dual problem are equal.

Proof: The total value of any feasible assignment $\{(i, k_i) \mid i = 1, \ldots, n\}$ satisfies

$$\sum_{i=1}^{n} a_{ik_i} \le \sum_{i=1}^{n} \max_{j \in A(i)} \big\{a_{ij} - p_j\big\} + \sum_{j=1}^{n} p_j, \tag{1.11}$$

for any set of prices $\{p_j \mid j = 1, \ldots, n\}$, since the first term of the right-hand side is no less than

$$\sum_{i=1}^{n} (a_{ik_i} - p_{k_i}),$$

while the second term is equal to $\sum_{i=1}^{n} p_{k_i}$. On the other hand, the given assignment and set of prices, denoted by $\{(i, j_i) \mid i = 1, \ldots, n\}$ and $\{\overline{p}_j \mid j = 1, \ldots, n\}$, respectively, satisfy the CS conditions, so we have

$$a_{ij_i} - \overline{p}_{j_i} = \max_{j \in A(i)} \{a_{ij} - \overline{p}_j\}, \qquad i = 1, \ldots, n.$$

By adding this relation over all i, we have

$$\sum_{i=1}^{n} \left(\max_{j \in A(i)} \{a_{ij} - \overline{p}_j\} + \overline{p}_{j_i} \right) = \sum_{i=1}^{n} a_{ij_i}$$

and by using Eq. (1.11), we obtain

$$\begin{aligned} \sum_{i=1}^{n} a_{ik_i} &\leq \sum_{i=1}^{n} \left(\max_{j \in A(i)} \{a_{ij} - \overline{p}_j\} + \overline{p}_{j_i} \right) \\ &= \sum_{i=1}^{n} a_{ij_i} \\ &\leq \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} + \sum_{j=1}^{n} p_j, \end{aligned}$$

for every feasible assignment $\{(i, k_i) \mid i = 1, \ldots, n\}$ and every set of prices $\{p_j \mid j = 1, \ldots, n\}$. Therefore, the assignment $\{(i, j_i) \mid i = 1, \ldots, n\}$ is optimal for the primal problem, and the set of prices $\{\overline{p}_j \mid j = 1, \ldots, n\}$ is optimal for the dual problem. Furthermore, the two optimal values are equal. **Q.E.D.**

In analogy with primal cost improvement algorithms, one may start with a price vector and try to successively obtain new price vectors with improved dual cost. The major algorithms of this type involve price changes of the form

$$p_i := \begin{cases} p_i + \gamma & \text{if } i \in \mathcal{S}, \\ p_i & \text{if } i \notin \mathcal{S}, \end{cases} \tag{1.12}$$

where $\mathcal{S}$ is a connected subset of nodes, and γ is some positive scalar that is small enough to ensure that the new price vector has an improved dual cost.

The existence of a node subset $\mathcal{S}$ that results in cost improvement at a nonoptimal price vector, as described above, will be shown in Chapter 6.

This is an important and remarkable result, which may be viewed as a dual version of the result of Prop. 1.2 (at a nonoptimal flow vector, there exists at least one unblocked simple cycle with negative cost). In fact both results are special cases of a more general theorem concerning elementary vectors of subspaces, which is central in the theory of *monotropic programming* (see Chapter 9).

Most dual cost improvement methods, simultaneously with changing p along a direction of dual cost improvement, also iterate on a flow vector x satisfying CS together with p. They terminate when x becomes feasible, at which time, by Prop. 1.3, the pair (x, p) must consist of a primal and a dual optimal solution.

In Chapter 6 we will discuss two main methods that select subsets $\mathcal{S}$ and corresponding directions of dual cost improvement in different ways:

(a) In the *primal-dual method*, the direction has a *steepest ascent property*, that is, it provides the maximal rate of improvement of the dual cost per unit change in the price vector.

(b) In the *relaxation (or coordinate ascent) method*, the direction is computed so that it has a small number of nonzero elements (i.e., the set $\mathcal{S}$ has few nodes). Such a direction may not be optimal in terms of rate of dual cost improvement, but can typically be computed much faster than the steepest ascent direction. Often the direction has only one nonzero element, in which case only one node price coordinate is changed; this motivates the name "coordinate ascent." Note, however, that coordinate ascent directions cannot be used exclusively to improve the dual cost, as is shown in Fig. 1.10.

1.3.3 Auction

Our third type of algorithm represents a significant departure from the cost improvement idea; at any one iteration, it may deteriorate both the primal and the dual cost, although in the end it does find an optimal primal solution. It is based on an approximate version of complementary slackness, called *ϵ-complementary slackness*, and while it implicitly tries to solve a dual problem, it actually attains a dual solution that is not quite optimal. This subsection introduces the main ideas underlying auction algorithms. Chapters 7 and 9 provide a detailed discussion for the minimum cost flow problem and for the separable convex cost problem, respectively.

Naive Auction

Let us return to the assignment problem, and consider a natural process for finding an equilibrium assignment and price vector. We will call this process the *naive auction algorithm*, because it has a serious flaw, as will be

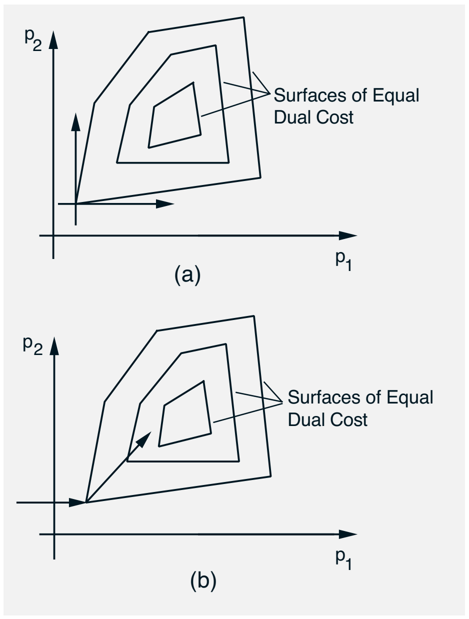


Figure 1.10: (a) The difficulty with using exclusively coordinate ascent iterations to solve the dual problem. Because the dual cost is piecewise linear, it may be impossible to improve it at some corner points by changing any *single* price coordinate. (b) As will be discussed in Chapter 6, a dual cost improvement is possible by changing several price coordinates by equal amounts, as in Eq. (1.12).

seen shortly. Nonetheless, this flaw will help motivate a more sophisticated and correct algorithm.

The naive auction algorithm proceeds in iterations and generates a sequence of price vectors and partial assignments. By a *partial assignment* we mean an assignment where only a subset of the persons have been matched with objects. A partial assignment should be contrasted with a *feasible* or *complete* assignment where all the persons have been matched with objects on a one-to-one basis. At the beginning of each iteration, the CS condition [cf. Eq. (1.10)]

$$a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\}$$

is satisfied for all pairs (i, j_i) of the partial assignment. If all persons are assigned, the algorithm terminates. Otherwise some person who is unassigned, say i, is selected. This person finds an object j_i which offers maximal value, that is,

$$j_i = \arg \max_{j \in A(i)} \{a_{ij} - p_j\},$$

and then:

(a) Gets assigned to the best object j_i; the person who was assigned to j_i at the beginning of the iteration (if any) becomes unassigned.

(b) Sets the price of j_i to the level at which he/she is indifferent between j_i and the second best object; that is, he/she sets p_{j_i} to

$$p_{j_i} + \gamma_i,$$

where

$$\gamma_i = v_i - w_i, \tag{1.13}$$

v_i is the best object value,

$$v_i = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{1.14}$$

and w_i is the second best object value,

$$w_i = \max_{j \in A(i),\, j \neq j_i} \{a_{ij} - p_j\}. \tag{1.15}$$

(Note that as p_{j_i} is increased, the value $a_{ij_i} - p_{j_i}$ offered by object j_i to person i is decreased. γ_i is the largest increment by which p_{j_i} can be increased, while maintaining the property that j_i offers maximal value to i.)

This process is repeated in a sequence of iterations until each person has been assigned to an object.

We may view this process as an auction where at each iteration the bidder i raises the price of a preferred object by the *bidding increment* γ_i. Note that γ_i cannot be negative, since $v_i \geq w_i$ [compare Eqs. (1.14)and (1.15)], so the object prices tend to increase. The choice γ_i is illustrated in Fig. 1.11. Just as in a real auction, bidding increments and price increases spur competition by making the bidder's own preferred object less attractive to other potential bidders.

ϵ-Complementary Slackness

Unfortunately, the naive auction algorithm does not always work (although it is an excellent initialization procedure for other methods, such as primal-dual or relaxation, and it is useful in other specialized contexts). The difficulty is that the bidding increment γ_i is 0 when two or more objects are tied in offering maximum value for the bidder i. As a result, a situation may be created where several persons contest a smaller number of equally desirable objects without raising their prices, thereby creating a never ending cycle; see Fig. 1.12.

To break such cycles, we introduce a perturbation mechanism, motivated by real auctions where each bid for an object must raise its price by a minimum positive increment, and bidders must on occasion take risks to win their preferred objects. In particular, let us fix a positive scalar ϵ, and

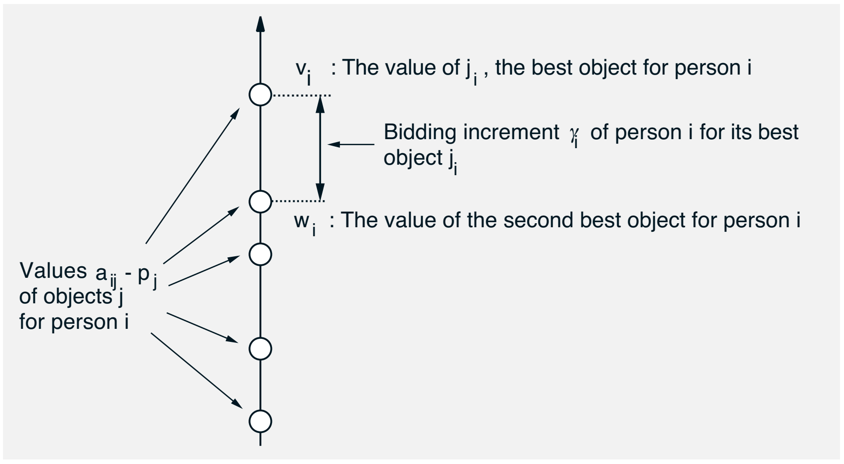

v_i : The value of j_i , the best object for person i
Bidding increment γ_i of person i for its best object j_i
w_i : The value of the second best object for person i
Values a_{ij} - p_j of objects j for person i

Figure 1.11: In the naive auction algorithm, even after the price of the best object j_i is increased by the bidding increment γ_i, j_i continues to be the best object for the bidder i, so CS is satisfied at the end of the iteration. However, we have $\gamma_i = 0$ if there is a tie between two or more objects that are most preferred by i.

say that a partial assignment and a price vector p satisfy *ϵ-complementary slackness (ϵ-CS for short)* if

$$a_{ij} - p_j \geq \max_{k \in A(i)} \{a_{ik} - p_k\} - \epsilon$$

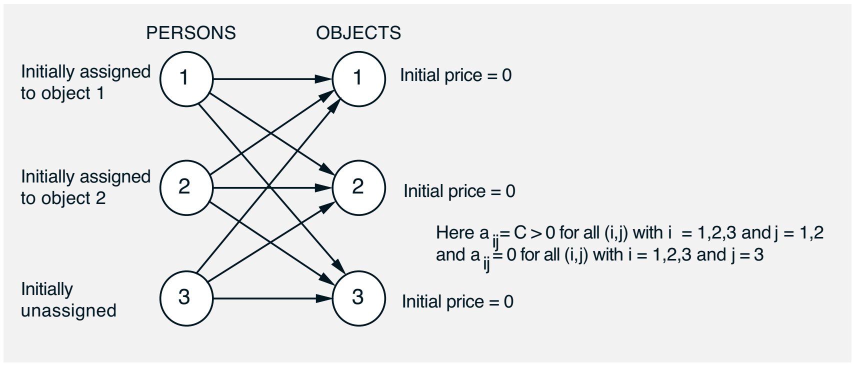
for all assigned pairs (i, j). In words, to satisfy ϵ-CS, all assigned persons of the partial assignment must be assigned to objects that are within ϵ of being best.

The Auction Algorithm

We now reformulate the previous auction process so that the bidding increment is always at least equal to ϵ. The resulting method, the *auction algorithm*, is the same as the naive auction algorithm, except that the bidding increment γ_i is

$$\gamma_i = v_i - w_i + \epsilon \tag{1.16}$$

rather than $\gamma_i = v_i - w_i$ as in Eq. (1.13). With this choice, the ϵ-CS condition is satisfied, as illustrated in Fig. 1.13. The particular increment $\gamma_i = v_i - w_i + \epsilon$ used in the auction algorithm is the maximum amount with this property. Smaller increments γ_i would also work as long as $\gamma_i \geq \epsilon$, but using the largest possible increment accelerates the algorithm. This is consistent with experience from real auctions, which tend to terminate faster when the bidding is aggressive.

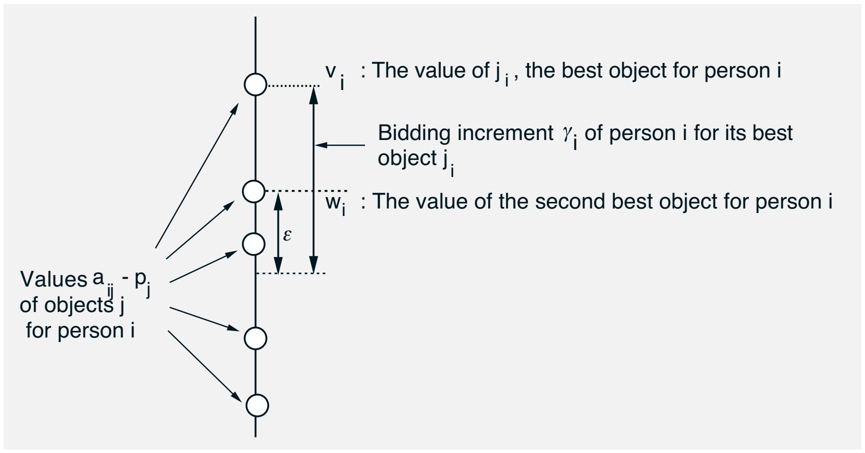

PERSONS
OBJECTS
Initially assigned to object 1
Initially assigned to object 2
Initially unassigned
1
2
3
1
2
3
Initial price = 0
Initial price = 0
Initial price = 0
Here a_{ij} = C > 0 for all (i,j) with i = 1,2,3 and j = 1,2
and a_{ij} = 0 for all (i,j) with i = 1,2,3 and j = 3

At Start of Iteration #	**Object Prices**	**Assigned Pairs**	**Bidder**	**Preferred Object**	**Bidding Increment**
1	0,0,0	(1,1), (2,2)	3	2	0
2	0,0,0	(1,1), (3,2)	2	2	0
3	0,0,0	(1,1), (2,2)	3	2	0

Figure 1.12: Illustration of how the naive auction algorithm may never terminate for a problem involving three persons and three objects. Here objects 1 and 2 offer benefit $C > 0$ to all persons, and object 3 offers benefit 0 to all persons. The algorithm cycles as persons 2 and 3 alternately bid for object 2 without changing its price because they prefer equally object 1 and object 2.

It can be shown that this reformulated auction process terminates, necessarily with a feasible assignment and a set of prices that satisfy ϵ-CS. To get a sense of this, note that if an object receives a bid during m iterations, its price must exceed its initial price by at least $m\epsilon$. Thus, for sufficiently large m, the object will become "expensive" enough to be judged "inferior" to some object that has not received a bid so far. It follows that only for a limited number of iterations can an object receive a bid while some other object still has not yet received any bid. On the other hand, once every object has received at least one bid, the auction terminates. (This argument assumes that any person can bid for any object, but it can be generalized to the case where the set of feasible person-object pairs is limited, as long as at least one feasible assignment exists; see Prop. 7.2 in Chapter 7.) Figure 1.14 shows how the auction algorithm, based on the bidding increment $\gamma_i = v_i - w_i + \epsilon$ [see Eq. (1.16)], overcomes the cycling difficulty in the example of Fig. 1.12.

When the auction algorithm terminates, we have an assignment satisfying ϵ-CS, but is this assignment optimal? The answer depends strongly



v_i : The value of j_i, the best object for person i
Bidding increment γ_i of person i for its best object j_i
w_i : The value of the second best object for person i
ε
Values a_{ij} - p_j of objects j for person i

Figure 1.13: In the auction algorithm, even after the price of the preferred object j_i is increased by the bidding increment γ_i, j_i will be within ϵ of being most preferred, so the ϵ-CS condition holds at the end of the iteration.

on the size of ϵ. In a real auction, a prudent bidder would not place an excessively high bid for fear of winning the object at an unnecessarily high price. Consistent with this intuition, we can show that if ϵ is small, then the final assignment will be "almost optimal." In particular, we will show that *the total benefit of the final assignment is within $n\epsilon$ of being optimal*. The idea is that a feasible assignment and a set of prices satisfying ϵ-CS may be viewed as satisfying CS for a *slightly different* problem, where all benefits a_{ij} are the same as before except the benefits of the n assigned pairs, which are modified by no more than ϵ.

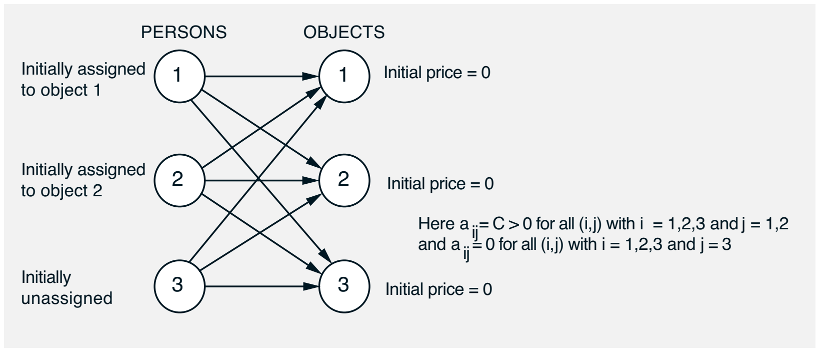
Proposition 1.4: A feasible assignment satisfying ϵ-complementary slackness, together with some price vector, attains within $n\epsilon$ the optimal primal value. Furthermore, the price vector attains within $n\epsilon$ the optimal dual cost.

Proof: Let A^* be the optimal total assignment benefit

$$A^* = \max_{\substack{k_i,\, i=1,\ldots,n \\ k_i \neq k_m \text{ if } i \neq m}} \sum_{i=1}^{n} a_{ik_i}$$

and let D^* be the optimal dual cost (cf. Prop. 1.3):

$$D^* = \min_{\substack{p_j \\ j=1,\ldots,n}} \left\{ \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} + \sum_{j=1}^{n} p_j \right\}.$$


PERSONS
OBJECTS
Initially assigned to object 1
Initially assigned to object 2
Initially unassigned
Initial price = 0
Initial price = 0
Initial price = 0
Here a_{ij} = C > 0 for all (i,j) with i = 1,2,3 and j = 1,2
and a_{ij} = 0 for all (i,j) with i = 1,2,3 and j = 3

At Start of Iteration #	**Object Prices**	**Assigned Pairs**	**Bidder**	**Preferred Object**	**Bidding Increment**
1	0,0,0	(1,1), (2,2)	3	2	ϵ
2	0,ϵ,0	(1,1), (3,2)	2	1	2ϵ
3	2ϵ,ϵ,0	(2,1), (3,2)	1	2	2ϵ
4	2ϵ,3ϵ,0	(1,2), (2,1)	3	1	2ϵ
5	4ϵ,3ϵ,0	(1,2), (3,1)	2	2	2ϵ
6	...	...	...	...	...

Figure 1.14: Illustration of how the auction algorithm, by making the bidding increment at least ϵ, overcomes the cycling difficulty for the example of Fig. 1.12. The table shows one possible sequence of bids and assignments generated by the auction algorithm, starting with all prices equal to 0 and with the partial assignment $\{(1,1),(2,2)\}$. At each iteration except the last, the person assigned to object 3 bids for either object 1 or 2, increasing its price by ϵ in the first iteration and by 2ϵ in each subsequent iteration. In the last iteration, after the prices of 1 and 2 reach or exceed C, object 3 receives a bid and the auction terminates.

If $\{(i,j_i) \mid i = 1,\ldots,n\}$ is the given assignment satisfying the ϵ-CS condition together with a price vector $\overline{p}$, we have

$$\max_{j\in A(i)} \{a_{ij} - \overline{p}_j\} - \epsilon \le a_{ij_i} - \overline{p}_{j_i}.$$

By adding this relation over all i, we see that

$$D^* \le \sum_{i=1}^{n} \left(\max_{j\in A(i)} \{a_{ij} - \overline{p}_j\} + \overline{p}_{j_i} \right) \le \sum_{i=1}^{n} a_{ij_i} + n\epsilon \le A^* + n\epsilon.$$

Since we showed in Prop. 1.3 that $A^* = D^*$, it follows that the total assignment benefit $\sum_{i=1}^n a_{ij_i}$ is within $n\epsilon$ of the optimal value A^*, while the dual cost of $\overline{p}$ is within $n\epsilon$ of the optimal dual cost. **Q.E.D.**

Suppose now that the benefits a_{ij} are all integer, which is the typical practical case. (If a_{ij} are rational numbers, they can be scaled up to integer by multiplication with a suitable common number.) Then the total benefit of any assignment is integer, so if $n\epsilon < 1$, any complete assignment that is within $n\epsilon$ of being optimal must be optimal. It follows that *if*

$$\epsilon < \frac{1}{n}$$

and the benefits a_{ij} are all integer, then the assignment obtained upon termination of the auction algorithm is optimal.

Figure 1.15 shows the sequence of generated object prices for the example of Fig. 1.12 in relation to the contours of the dual cost function. It can be seen from this figure that each bid has the effect of setting the price of the object receiving the bid nearly equal (within ϵ) to the price that minimizes the dual cost with respect to that price, with all other prices held fixed (this will be shown rigorously in Section 7.1). Successive minimization of a cost function along single coordinates is a central feature of coordinate descent and relaxation methods, which are popular for unconstrained minimization of smooth functions and for solving systems of smooth equations. Thus, the auction algorithm can be interpreted as an approximate coordinate descent method; as such, it is related to the relaxation method discussed in the previous subsection.

Scaling

Figure 1.15 also illustrates a generic feature of auction algorithms. The amount of work needed to solve the problem can depend strongly on the value of ϵ and on the maximum absolute object benefit

$$C = \max_{(i,j)\in\mathcal{A}} |a_{ij}|.$$

Basically, for many types of problems, the number of iterations up to termination tends to be proportional to C/ϵ. This can be seen from the figure, where the total number of iterations is roughly C/ϵ, starting from zero initial prices.

Note also that there is a dependence on the initial prices; if these prices are "near optimal," we expect that the number of iterations needed to solve the problem will be relatively small. This can be seen from Fig. 1.15; if the initial prices satisfy $p_1 \approx p_3 + C$ and $p_2 \approx p_3 + C$, the number of iterations up to termination is quite small.

The preceding observations suggest the idea of *ϵ-scaling*, which consists of applying the algorithm several times, starting with a large value of ϵ and successively reducing ϵ until it is less than some critical value (for example, $1/n$, when a_{ij} are integer). Each application of the algorithm provides good initial prices for the next application. This is a common idea

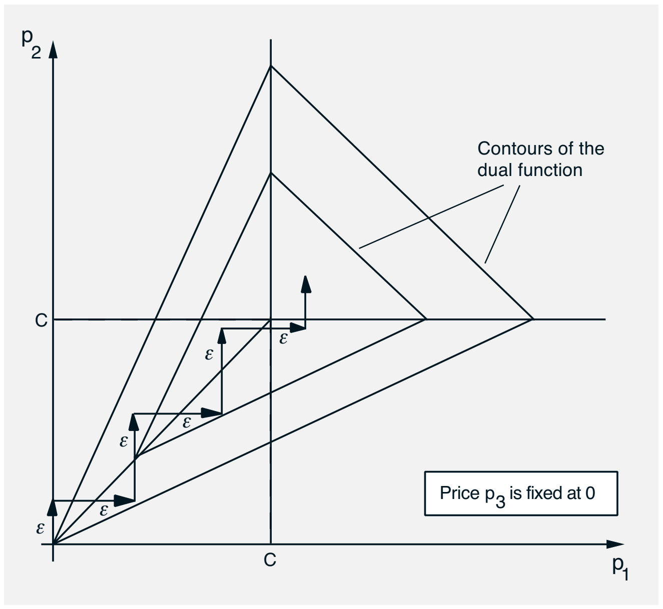


Figure 1.15: A sequence of prices p_1 and p_2 generated by the auction algorithm for the example of Figs. 1.12 and 1.14. The figure shows the equal dual cost surfaces in the space of p_1 and p_2, with p_3 fixed at 0. The arrows indicate the price iterates as given by the table of Fig. 1.14. Termination occurs when the prices reach an ϵ-neighborhood of the point (C, C), and object 3 becomes "sufficiently inexpensive" to receive a bid and to get assigned. The total number of iterations is roughly C/ϵ, starting from zero initial prices.

in nonlinear programming; it is encountered, for example, in barrier and penalty function methods (see Section 8.8). In practice, scaling is typically beneficial, and accelerates the termination of the auction algorithm.

1.3.4 Good, Bad, and Polynomial Algorithms

We have discussed several types of methods, so the natural question arises: is there a best method and what criterion should we use to rank methods?

A practitioner who has a specific type of problem to solve, perhaps repeatedly, with the data and size of the problem within some limited range, will usually be interested in one or more of the following:

(a) Fast solution time.

(b) Flexibility to use good starting solutions (which the practitioner can usually provide, based on his/her knowledge of the problem, or based on a known solution of some similar problem).

(c) The ability to perform sensitivity analysis (resolve the problem with slightly different problem data) quickly.

(d) The ability to take advantage of parallel computing hardware.

Given the diversity of these considerations, it is not surprising that there is no algorithm that will dominate the others in all or even most practical situations. Otherwise expressed, every type of algorithm that we will discuss is best given the right type of practical situation. Thus, to make intelligent choices, the practitioner needs to understand the properties of different algorithms relating to speed of convergence, flexibility, parallelization, and suitability for specific problem structures. For challenging problems, the choice of algorithm is often settled by experimentation with several candidates.

A theoretical analyst may also have difficulty ranking different algorithms for specific types of problems. The most common approach for this purpose is worst-case computational complexity analysis. For example, for the minimum cost flow problem, one tries to bound the number of elementary numerical operations needed by a given algorithm with some measure of the "problem size," that is, with some expression of the form

$$Kf(N, A, C, U, S),$$

where

N is the number of nodes,

A is the number of arcs,

C is the arc cost range $\max_{(i,j)\in\mathcal{A}} |a_{ij}|$,

U is the maximum arc flow range $\max_{(i,j)\in\mathcal{A}}(c_{ij} - b_{ij})$,

S is the supply range $\max_{i\in\mathcal{N}} |s_i|$,

f is some known function,

K is a (usually unknown) constant.

If a bound of this form can be found, we say that the *running time* or *operation count of the algorithm is* $O\big(f(N, A, C, U, S)\big)$. If $f(N, A, C, U, S)$ can be written as a polynomial function of the number of bits needed to express the problem data, the algorithm is said to be *polynomial*. Examples of polynomial complexity bounds are $O\big(N^\alpha A^\beta\big)$ and $O\big(N^\alpha A^\beta \log C\big)$, where α and β are positive integers, and the numbers a_{ij} are assumed integer. The bound $O\big(N^\alpha A^\beta\big)$ is sometimes said to be *strongly polynomial* because it involves only the graph size parameters. A bound of the form $O\big(N^\alpha A^\beta C\big)$ is not polynomial, even assuming that the a_{ij} are integer, because C is not a polynomial expression of $\log C$, the number of bits needed to express a single number a_{ij}. Bounds like $O\big(N^\alpha A^\beta C\big)$, which are polynomial in the problem data rather than in the number of bits needed to express the data, are called *pseudopolynomial*.

A common assumption in theoretical computer science is that polynomial algorithms are "better" than pseudopolynomial, and pseudopolynomial algorithms are "better" than exponential [for example, those with a bound of the form $K2^{g(N,A)}$, where g is a polynomial in N and A]. Furthermore, it is thought that two polynomial algorithms can be compared in terms of the degree of the polynomial bound; e.g., an $O(N^2)$ algorithm is "better" than an $O(N^3)$ algorithm. Unfortunately, quite often this assumption is not supported by computational practice in linear programming and network optimization. Pseudopolynomial and even exponential algorithms are often faster in practice than polynomial ones. In fact, the simplex method for general linear programs is an exponential algorithm, as shown by Klee and Minty [1972] (see also the textbooks by Chvatal [1983], or Bertsimas and Tsitsiklis [1997]), and yet it is used widely, because of its excellent practical properties.

There are two main reasons why worst-case complexity estimates may fail to predict the practical performance of network flow algorithms. First, the estimates, even if they are tight, may be very pessimistic as they may correspond to problem instances that are highly unlikely in practice. (Average complexity estimates would be more appropriate for such situations. However, obtaining these is usually hard, and the statistical assumptions underlying them may be inappropriate for many types of practical problems.) Second, worst-case complexity estimates involve the (usually unknown) constant K, which may dominate the estimate for all except for unrealistically large problem sizes. Thus, a comparison between two algorithms that is based on the size-dependent terms of running time estimates, and does not take into account the corresponding constants may be unreliable.

Despite its shortcomings, computational complexity analysis is valuable because it often illuminates the computational bottlenecks of many algorithms and motivates the use of efficient data structures. For this reason, throughout the book, we will comment on available complexity results, we will prove some of the most important estimates, and we will try to relate these estimates to computational practice. For some classes of problems, however, it turns out that the methods with the best computational complexity are impractical, because they are either too complicated or too slow in practice. In such cases, we will refer to the literature, without providing a detailed discussion.

1.4 NOTES, SOURCES, AND EXERCISES

Network problems are discussed in many books (Berge [1962], Berge and Ghouila-Houri [1962], Ford and Fulkerson [1962], Dantzig [1963], Busacker and Saaty [1965], Hu [1969], Iri [1969], Frank and Frisch 1970], Christofides

[1975], Zoutendijk [1976], Minieka [1978], Jensen and Barnes [1980], Kennington and Helgason [1980], Papadimitriou and Steiglitz [1982], Chvatal [1983], Gondran and Minoux [1984], Luenberger [1984], Rockafellar [1984], Bazaraa, Jarvis, and Sherali [1990], Bertsekas [1991a], Murty [1992], Bertsimas and Tsitsiklis [1997]). Several of these books discuss linear programming first and develop linear network optimization as a special case. An alternative approach that relies heavily on duality, is given by Rockafellar [1984]. The conformal realization theorem (Prop. 1.1) has been developed in different forms in several sources, including Ford and Fulkerson [1962], Busacker and Saaty [1965], and Rockafellar [1984].

The primal cost improvement approach for network optimization was initiated by Dantzig [1951], who specialized the simplex method to the transportation problem. The extensive subsequent work using this approach is surveyed at the end of Chapter 5.

The dual cost improvement approach was initiated by Kuhn [1955] who proposed the *Hungarian method* for the assignment problem. (The name of the algorithm honors its connection with the research of the Hungarian mathematicians Egervary [1931] and König [1931].) Work using this approach is surveyed in Chapter 6.

The auction approach was initiated in Bertsekas [1979a] for the assignment problem, and in Bertsekas [1986a], [1986b] for the minimum cost flow problem. Work using this approach is surveyed at the end of Chapter 7.

E X E R C I S E S

1.1

Consider the graph and the flow vector of Fig. 1.16.

(a) Enumerate the simple paths and the simple forward paths that start at node 1.

(b) Enumerate the simple cycles and the simple forward cycles of the graph.

(c) Is the graph connected? Is it strongly connected?

(d) Calculate the divergences of all the nodes and verify that they add to 0.

(e) Give an example of a simple path flow that starts at node 1, ends at node 5, involves four arcs, and conforms to the given flow vector.

(f) Suppose that all arcs have arc flow bounds -1 and 5. Enumerate all the simple paths that start at node 1, end at node 5, and are unblocked with respect to the given flow vector.

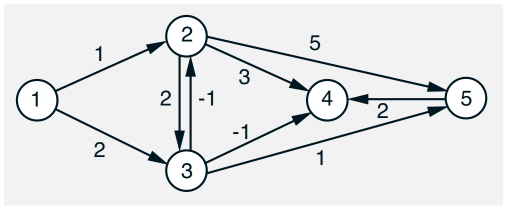


Figure 1.16: Flow vector for Exercise 1.1. The arc flows are the numbers shown next to the arcs.

1.2 (Proof of the Conformal Realization Theorem)

Prove the conformal realization theorem (Prop. 1.1) by completing the details of the following argument. Assume first that x is a circulation. Consider the following procedure by which given x, we obtain a simple cycle flow x' that conforms to x and satisfies

$$\begin{array}{ll} 0 \le x'_{ij} \le x_{ij} & \text{for all arcs } (i,j) \text{ with } 0 \le x_{ij}, \\ x_{ij} \le x'_{ij} \le 0 & \text{for all arcs } (i,j) \text{ with } x_{ij} \le 0, \\ x_{ij} = x'_{ij} & \text{for at least one arc } (i,j) \text{ with } x_{ij} \ne 0; \end{array}$$

(see Fig. 1.17). Choose an arc (i,j) with $x_{ij} \ne 0$. Assume that $x_{ij} > 0$. (A similar procedure can be used when $x_{ij} < 0$.) Construct a sequence of node subsets $T_0, T_1, \ldots$, as follows: Take $T_0 = \{j\}$. For $k = 0, 1, \ldots$, given T_k, let

$$T_{k+1} = \Big\{ n \notin \cup_{p=0}^{k} T_p \mid \text{ there is a node } m \in T_k, \text{ and either an arc } (m,n)$$
$$\text{such that } x_{mn} > 0 \text{ or an arc } (n,m) \text{ such that } x_{nm} < 0 \Big\},$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k such that $x_{mn} > 0$ or $x_{nm} < 0$, respectively. The procedure terminates when T_{k+1} is empty.

At the end of the procedure, trace labels backward from i until node j is reached. (How do we know that i belongs to one of the sets T_k?) In particular, let "(i_1, i)" or "(i, i_1)" be the label of i, let "(i_2, i_1)" or "(i_1, i_2)" be the label of i_1, etc., until a node i_k with label "(i_k, j)" or "(j, i_k)" is found. The cycle $C = (j, i_k, i_{k-1}, \ldots, i_1, i, j)$ is simple, it contains (i,j) as a forward arc, and is such that all its forward arcs have positive flow and all its backward arcs have negative flow. Let $a = \min_{(m,n)\in C} |x_{mn}| > 0$. Then the simple cycle flow x', where

$$x'_{ij} = \begin{cases} a & \text{if } (i,j) \in C^+, \\ -a & \text{if } (i,j) \in C^-, \\ 0 & \text{otherwise,} \end{cases}$$

has the required properties.

Now subtract x' from x. We have $x_{ij} - x'_{ij} > 0$ only for arcs (i,j) with $x_{ij} > 0$, $x_{ij} - x'_{ij} < 0$ only for arcs (i,j) with $x_{ij} < 0$, and $x_{ij} - x'_{ij} = 0$ for at

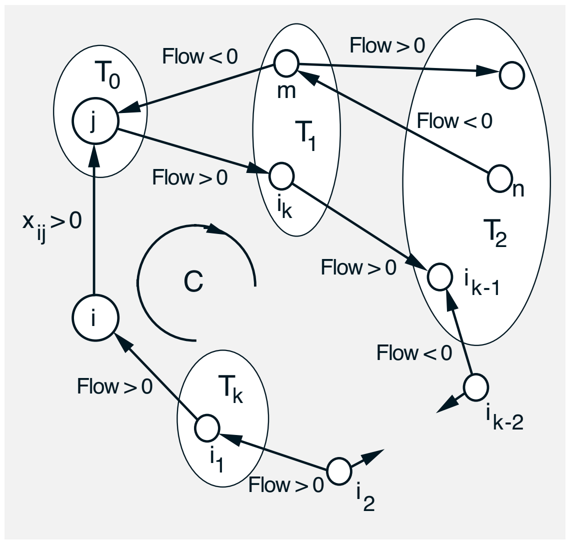


Figure 1.17: Construction of a cycle of arcs with nonzero flow used in the proof of the conformal realization theorem.

least one arc (i, j) with $x_{ij} \neq 0$. If x is integer, then x' and $x - x'$ will also be integer. We then repeat the process (for at most A times) with the circulation x replaced by the circulation $x - x'$ and so on, until the zero flow is obtained.

If x is not a circulation, we form an enlarged graph by introducing a new node s and by introducing for each node $i \in \mathcal{N}$ an arc (s, i) with flow x_{si} equal to the divergence y_i. The resulting flow vector is seen to be a circulation in the enlarged graph (why?). This circulation, by the result just shown, can be decomposed into at most $A + N$ simple cycle flows of the enlarged graph, conforming to the flow vector. Out of these cycle flows, we consider those containing node s, and we remove s and its two incident arcs while leaving the other cycle flows unchanged. As a result we obtain a set of at most $A+N$ path flows of the original graph, which add up to x. These path flows also conform to x, as required.

1.3

Use the algorithm of Exercise 1.2 to decompose the flow vector of Fig. 1.16 into conforming simple path flows.

1.4 (Path Decomposition Theorem)

(a) Use the conformal realization theorem (Prop. 1.1) to show that a forward path P can be decomposed into a (possibly empty) collection of simple forward cycles, together with a simple forward path that has the same start node and end node as P. (Here "decomposition" means that the union of the arcs of the component paths is equal to the set of arcs of P with the multiplicity of repeated arcs properly accounted for.)

(b) Suppose that a graph is strongly connected and that a length a_{ij} is given for every arc (i,j). Show that if all forward cycles have nonnegative length, then there exists a shortest path from any node s to any node t. Show also that if there exists a shortest path from some node s to some node t, then all forward cycles have nonnegative length. Why is the connectivity assumption needed?

1.5 (Cycle Decomposition - Euler Cycles)

Consider a graph such that each of the nodes has even degree.

(a) Give an algorithm to decompose the graph into a collection of simple cycles that are disjoint, in the sense that they share no arcs (although they may share some nodes). (Here "decomposition" means that the union of the arcs of the component cycles is equal to the set of arcs of the graph.) *Hint*: Given a connected graph where each of the nodes has even degree, the deletion of the arcs of any cycle creates some connected subgraphs where each of the nodes has even degree (including possibly some isolated nodes).

(b) Assume in addition that the graph is connected. Show that there is an Euler cycle, i.e., a cycle that contains all the arcs of a graph exactly once. *Hint*: Apply the decomposition of part (a), and successively merge an Euler cycle of a subgraph with a simple cycle.

1.6

In the graph of Fig. 1.16, consider the graph obtained by deleting node 1 and arcs $(1,2)$, $(1,3)$, and $(5,4)$. Decompose this graph into a collection of simple cycles that are disjoint (cf. Exercise 1.5) and construct an Euler cycle.

1.7

(a) Consider an $n \times n$ chessboard, and a rook that is allowed to make the standard moves along the rows and columns. Show that the rook can start at a given square and return to that square after making each of the possible legal moves exactly once and in one direction only [of the two moves (a,b) and (b,a) only one should be made]. *Hint*: Construct an Euler cycle in a suitable graph.

(b) Consider an $n \times n$ chessboard with n even, and a bishop that is allowed to make two types of moves: legal moves (which are the standard moves along the diagonals of its color), and illegal moves (which go from any square of its color to any other square of its color). Show that the bishop can start at a given square and return to that square after making each of the possible legal moves exactly once and in one direction only, plus $n^2/4$ illegal moves.

For every square of its color, there should be exactly one illegal move that either starts or ends at that square.

1.8 (Forward Euler Cycles)

Consider a graph and the question whether there exists a forward cycle that passes through each arc of the graph exactly once. Show that such a cycle exists if and only if the graph is connected and the number of incoming arcs to each node is equal to the number of outgoing arcs from the node.

1.9

Consider an $n \times n$ chessboard with $n \geq 4$. Show that a knight starting at any square can visit every other square, with a move sequence that contains every possible move exactly once [a move (a, b) as well as its reverse (b, a) should be made]. Interpret this sequence as a forward Euler cycle in a suitable graph (cf. Exercise 1.8).

1.10 (Euler Paths)

Consider a graph and the question whether there exists a path that passes through each arc of the graph exactly once. Show that such a path exists if and only if the graph is connected, and either the degrees of all the nodes are even, or else the degrees of all the nodes except two are even.

1.11

In shatranj, the old version of chess, the firz (or vizier, the predecessor to the modern queen) can move one square diagonally in each direction. Show that starting at a corner of an $n \times n$ chessboard where n is even, the firz can reach the opposite corner after making each of the possible moves along its diagonals exactly once and in one direction only [of the two moves (a, b) and (b, a) only one should be made].

1.12

Show that the number of nodes with odd degree in a graph is even.

1.13

Assume that all the nodes of a graph have degree greater than one. Show that the graph must contain a cycle.

1.14

(a) Show that every tree with at least two nodes has at least two nodes with degree one.

(b) Show that a graph is a tree if and only if it is connected and the number of arcs is one less than the number of nodes.

1.15

Consider a volleyball net that consists of a mesh with m squares on the horizontal dimension and n squares on the vertical. What is the maximum number of strings that can be cut before the net falls apart into two pieces.

1.16 (Checking Connectivity)

Consider a graph with A arcs.

(a) Devise an algorithm with $O(A)$ running time that checks whether the graph is connected, and if it is connected, simultaneously constructs a path connecting any two nodes. *Hint*: Start at a node, mark its neighbors, and continue.

(b) Repeat part (a) for the case where we want to check strong connectedness.

(c) Devise an algorithm with $O(A)$ running time that checks whether there exists a cycle that contains two given nodes.

(d) Repeat part (c) for the case where the cycle is required to be forward.

1.17 (Inequality Constrained Minimum Cost Flows)

Consider the following variant of the minimum cost flow problem:

$$\begin{aligned} &\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} \quad \underline{s}_i \le \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} \le \overline{s}_i, \qquad \forall\ i \in \mathcal{N}, \\ &\qquad\qquad b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \end{aligned}$$

where the bounds $\underline{s}_i$ and $\overline{s}_i$ on the divergence of node i are given. Show that this problem can be converted to a standard (equality constrained) minimum cost flow problem by adding an extra node A and an arc (A,i) from this node to every other node i, with feasible flow range $[0, \overline{s}_i - \underline{s}_i]$.

1.18 (Node Throughput Constraints)

Consider the minimum cost flow problem with the additional constraints that the total flow of the outgoing arcs from each node i must lie within a given range $[\underline{t}_i, \overline{t}_i]$, that is,

$$\underline{t}_i \le \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} \le \overline{t}_i.$$

Convert this problem into the standard form of the minimum cost flow problem by splitting each node into two nodes with a connecting arc.

1.19 (Piecewise Linear Arc Costs)

Consider the minimum cost flow problem with the difference that, instead of the linear form $a_{ij}x_{ij}$, each arc's cost function has the piecewise linear form

$$f_{ij}(x_{ij}) = \begin{cases} a_{ij}^1 x_{ij} & \text{if } b_{ij} \le x_{ij} \le m_{ij}, \\ a_{ij}^1 m_{ij} + a_{ij}^2 (x_{ij} - m_{ij}) & \text{if } m_{ij} \le x_{ij} \le c_{ij}, \end{cases}$$

where m_{ij}, a_{ij}^1, and a_{ij}^2 are given scalars satisfying $b_{ij} \le m_{ij} \le c_{ij}$ and $a_{ij}^1 \le a_{ij}^2$.

(a) Show that the problem can be converted to a linear minimum cost flow problem where each arc (i,j) is replaced by two arcs with arc cost coefficients a_{ij}^1 and a_{ij}^2, and arc flow ranges $[b_{ij}, m_{ij}]$ and $[0, c_{ij} - m_{ij}]$, respectively.

(b) Generalize to the case of piecewise linear cost functions with more than two pieces.

1.20 (Asymmetric Assignment and Transportation Problems)

Consider an assignment problem where the number of objects is larger than the number of persons, and we require that each person be assigned to one object. The associated linear program (cf. Example 1.2) is

$$\begin{aligned} &\text{maximize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} \quad \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\ i = 1,\ldots,m, \\ &\qquad\qquad \sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} \le 1, \qquad \forall\ j = 1,\ldots,n, \\ &\qquad\qquad 0 \le x_{ij} \le 1, \qquad \forall\ (i,j) \in \mathcal{A}, \end{aligned}$$

where $m < n$.

(a) Show how to formulate this problem as a minimum cost flow problem by introducing extra arcs and nodes.

(b) Repeat part (a) for the case where there may be some persons that are left unassigned; that is, the constraint $\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1$ is replaced by $\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} \leq 1$. Give an example of a problem with $a_{ij} > 0$ for all $(i,j) \in \mathcal{A}$, which is such that in the optimal assignment some persons are left unassigned, even though there exist feasible assignments that assign every person to some object.

(c) Formulate an asymmetric transportation problem where the total supply is less than the total demand, but some demand may be left unsatisfied, and appropriately modify your answers to parts (a) and (b).

1.21 (Bipartite Matching)

Bipartite matching problems are assignment problems where the coefficients (i,j) are all equal to 1. In such problems, we want to maximize the cardinality of the assignment, that is, the number of assigned pairs (i,j). Formulate a bipartite matching problem as an equivalent max-flow problem.

1.22 (Production Planning)

Consider a problem of scheduling production of a certain item to meet a given demand over N time periods. Let us denote:

x_i: The amount of product stored at the beginning of period i, where $i = 0, \ldots, N-1$. There is a nonnegativity constraint on x_i.

u_i: The amount of product produced during period i. There is a constraint $0 \leq u_i \leq c_i$, where the scalar c_i is given for each i.

d_i: The amount of product demanded during period i. This is a given scalar for each i.

The amount of product stored evolves according to the equation

$$x_{i+1} = x_i + u_i - d_i, \qquad i = 0, \ldots, N-1.$$

Given x_0, we want to find a feasible production sequence $\{u_0, \ldots, u_{N-1}\}$ that minimizes

$$\sum_{i=0}^{N-1} (a_i x_i + b_i u_i),$$

where a_i and b_i are given scalars for each i. Formulate this problem as a minimum cost flow problem. *Hint*: For each i, introduce a node that connects to a special artificial node.

1.23 (Capacity Expansion)

The capacity of a certain facility is to be expanded over N time periods by adding an increment $u_i \in [0, c_i]$ at time period $i = 0, \ldots, N-1$, where c_i is a given scalar. Thus, if x_i is the capacity at the beginning of period i, we have

$$x_{i+1} = x_i + u_i, \qquad i = 0, \ldots, N-1.$$

Given x_0, consider the problem of finding u_i, $i = 0, \ldots, N-1$, such that each x_i lies within a given interval $[\underline{x}_i, \bar{x}_i]$ and the cost

$$\sum_{i=0}^{N-1} (a_i x_i + b_i u_i)$$

is minimized, where a_i and b_i are given scalars for each i. Formulate the problem as a minimum cost flow problem.

1.24 (Dynamic Transhipment Problems)

Consider a transhipment context for the minimum cost flow problem where the problem is to optimally transfer flow from some supply points to some demand points over arcs of limited capacity. In a dynamic version of this context, the transfer is to be performed over N time units, and transferring flow along an arc (i, j) requires time τ_{ij}, which is a given positive integer number of time units. This means that at each time $t = 0, \ldots, N - \tau_{ij}$, we may send from node i along arc (i, j) a flow $x_{ij} \in [0, c_{ij}]$, which will arrive at node j at time $t + \tau_{ij}$. Formulate this problem as a minimum cost flow problem involving a copy of the given graph for each time period.

1.25 (Concentrator Assignment)

We have m communication terminals, each to be connected to one out of a given collection of concentrators. Suppose that there is a cost a_{ij} for connecting terminal i to concentrator j, and that each concentrator j has an upper bound b_j on the number of terminals it can be connected to. Also, each terminal i can be connected to only a given subset of concentrators.

(a) Formulate the problem of finding the minimum cost connection of terminals to concentrators as a minimum cost flow problem. *Hint*: You may use the fact that there exists an integer optimal solution to a minimum cost flow problem with integer supplies and arc flow bounds. (This will be shown in Chapter 5.)

(b) Suppose that a concentrator j can operate in an overload condition with a number of connected terminals greater than b_j, up to a number $\overline{b}_j > b_j$. In this case, however, the cost per terminal connected becomes $\overline{a}_{ij} > a_{ij}$. Repeat part (a).

(c) Suppose that when no terminals are connected to concentrator j there is a given cost savings $c_j > 0$. Can you still formulate the problem as a minimum cost flow problem?

1.26

Consider a round-robin chess tournament involving n players that play each other once. A win scores 1 for the winner and 0 for the loser, while a draw scores 1/2 for each player. We are given a set of final scores $(s_1, \ldots, s_n)$ for the players, from the range $[0, n-1]$, whose sum is $n(n-1)/2$, and we want to check whether these scores are feasible [for example, in a four-player tournament, a set of final scores of $(3, 3, 0, 0)$ is impossible]. Show that this is equivalent to checking feasibility of some transportation problem.

1.27 (k-Color Problem)

Consider the k-color problem, which is to assign one out of k colors to each node of a graph so that for every arc (i, j), nodes i and j have different colors.

(a) Suppose we want to choose the colors of countries in a world map so that no two adjacent countries have the same color. Show that if the number of available colors is k, the problem can be formulated as a k-color problem.

(b) Show that the k-color problem has a solution if and only if the number of nodes can be partitioned in k or less disjoint subsets such that there is no arc connecting a pair of nodes from the same subset.

(c) Show that when the graph is a tree, the 2-color problem has a solution. *Hint*: First color some node i and then color the remaining nodes based on their "distance" from i.

(d) Show that if each node has at most $k-1$ neighbors, the k-color problem has a solution.

1.28 (k-Coloring and Parallel Computation)

Consider the n-dimensional vector $x = (x_1, \ldots, x_n)$ and an iteration of the form

$$x_j := f_j(x), \qquad j = 1, \ldots, n,$$

where $f = (f_1, \ldots, f_n)$ is a given function. The *dependency graph* of f has nodes $1, \ldots, n$ and an arc set such that (i, j) is an arc if the function f_j exhibits a dependence on the component x_i. Consider an ordering $j_1, \ldots, j_n$ of the indices $1, \ldots, n$, and a partition of $\{j_1, \ldots, j_n\}$ into disjoint subsets $J_1, \ldots, J_M$ such that:

(1) For all k, if $j_k \in J_m$, then $j_{k+1} \in J_m \cup \cdots \cup J_M$.

(2) If $j_p, j_q \in J_m$ and $p < q$, then f_{j_q} does not depend on x_{j_p}.

Show that such an ordering and partition exist if and only if the nodes of the dependency graph can be colored with M colors so that there exists no forward cycle with all the nodes on the cycle having the same color. *Note*: This is challenging (see Bertsekas and Tsitsiklis [1989], Section 1.2.4, for discussion and analysis). An ordering and partition of this type can be used to execute Gauss-Seidel iterations in M parallel steps.

1.29 (Replacing Arc Costs with Reduced Costs)

Consider the minimum cost flow problem and let p_j be a scalar price for each node j. Show that if the arc cost coefficients a_{ij} are replaced by $a_{ij} + p_j - p_i$, we obtain a problem that is equivalent to the original (except for a scalar shift in the cost function value).

1.30

Consider the assignment problem.

(a) Show that every k-person exchange can be accomplished with a sequence of $k - 1$ successive two-person exchanges.

(b) In light of the result of part (a), how do you explain that a nonoptimal assignment may not be improvable by any two-person exchange?

1.31 (Dual Cost Improvement Directions)

Consider the assignment problem. Let p_j denote the price of object j, let T be a subset of objects, and let

$$S = \Big\{ i \mid \text{the maximum of } a_{ij} - p_j \text{ over } j \in A(i) \\ \text{is attained by some element of } T \Big\}.$$

Assume that:

(1) For each $i \in S$, the maximum of $a_{ij} - p_j$ over $j \in A(i)$ is attained only by elements of T.

(2) S has more elements than T.

Show that the direction $d = (d_1, \ldots, d_n)$, where $d_j = 1$ if $j \in T$ and $d_j = 0$ if $j \notin T$, is a direction of dual cost improvement. *Note*: Directions of this type are used by the most common dual cost improvement algorithms for the assignment problem.

1.32

Use ϵ-CS to verify that the assignment of Fig. 1.18 is optimal and obtain a bound on how far from optimal the given price vector is. State the dual problem and verify the correctness of the bound by comparing the dual value of the price vector with the optimal dual value.

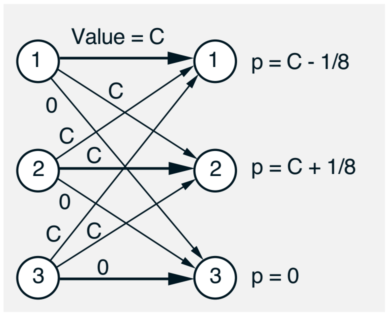


Figure 1.18: Graph of an assignment problem. Objects 1 and 2 have value C for all persons. Object 3 has value 0 for all persons. Object prices are as shown. The thick lines indicate the given assignment.

1.33 (Generic Negative Cycle Algorithm)

Consider the following minimum cost flow problem

$$\begin{aligned} &\text{minimize} && \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} && \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\, i \in \mathcal{N}, \\ & && 0 \le x_{ij} \le c_{ij}, \qquad \forall\,(i,j)\in\mathcal{A}, \end{aligned}$$

and assume that the problem has at least one feasible solution. Consider first the circulation case where $s_i = 0$ for all $i \in \mathcal{N}$. Construct a sequence of flow vectors $x^0, x^1, \ldots$ as follows: Start with $x^0 = 0$. Given x^k, stop if x^k is optimal, and otherwise find a simple cycle C^k that is unblocked with respect to x^k and has negative cost (cf. Prop. 1.2). Increase (decrease) the flow of the forward (backward, respectively) arcs of C^k by the maximum possible increment.

(a) Show that the cost of x^{k+1} is smaller than the cost of x^k by an amount that is proportional to the cost of the cycle C^k and to the increment of the corresponding flow change.

(b) Assume that the flow increment at each iteration is greater or equal to some scalar $\delta > 0$. Show that the algorithm must terminate after a finite number of iterations with an optimal flow vector. *Note*: The assumption of existence of such a δ is essential (see Exercise 3.7 in Chapter 3).

(c) Extend parts (a) and (b) to the general case where we may have $s_i \neq 0$ for some i, by converting the problem to the circulation format (a method for doing this is given in Section 4.1.3).

1.34 (Integer Optimal Solutions of Min-Cost Flow Problems)

Consider the minimum cost flow problem of Exercise 1.33, where the upper bounds c_{ij} are given positive integers and the supplies s_i are given integers. Assume that the problem has at least one feasible solution. Show that there exists an optimal flow vector that is integer. *Hint*: Show that the flow vectors generated by the negative cycle algorithm of Exercise 1.33 are integer.

1.35 (The Original Hamiltonian Cycle)

The origins of the traveling salesman problem can be traced (among others) to the work of the Irish mathematician Sir William Hamilton. In 1856, he developed a system of commutative algebra, which inspired a puzzle marketed as the "Icosian Game." The puzzle is to find a cycle that passes exactly once through each of the 20 nodes of the graph shown in Fig. 1.19, which represents a regular dodecahedron. Find a Hamiltonian cycle on this graph using as first four nodes the ones marked 1-4 (all arcs are considered bidirectional).

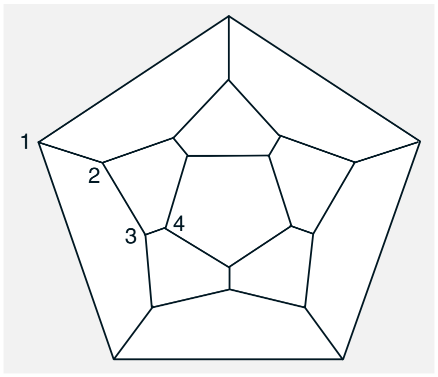


Figure 1.19: Graph for the Icosian Game (cf. Exercise 1.35). The arcs and nodes correspond to the edges and vertices of the regular dodecahedron, respectively. The name "icosian" comes from the Greek word "icosi," which means twenty. Adjacent nodes of the dodecahedron correspond to adjacent faces of the regular icosahedron.

1.36 (Hamiltonian Cycle on the Hypercube)

The hypercube of dimension n is a graph with 2^n nodes, each corresponding to an n-bit string where each bit is either a 0 or a 1. There is a bidirectional arc connecting every pair of nodes whose n-bit strings differ by a single bit. Show that for every $n \geq 2$, the hypercube contains a Hamiltonian cycle. *Hint*: Use induction.

1.37 (Hardy's Theorem)

Let $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$ be monotonically nondecreasing sequences of numbers. Consider the problem of associating with each $i = 1, \ldots, n$ a distinct index j_i in a way that maximizes $\sum_{i=1}^{n} a_i b_{j_i}$. Formulate this as an assignment problem and show that it is optimal to select $j_i = i$ for all i. *Hint*: Use the complementary slackness conditions with prices defined by $p_1 = 0$ and $p_k = p_{k-1} + a_k(b_k - b_{k-1})$ for $k = 2, \ldots, n$.

2

The Shortest Path Problem

Contents

2.1. Problem Formulation and Applications

2.2. A Generic Shortest Path Algorithm

2.3. Label Setting (Dijkstra) Methods
- 2.2.1. Performance of Label Setting Methods
- 2.3.2. The Binary Heap Method
- 2.3.3. Dial's Algorithm

2.4. Label Correcting Methods
- 2.4.1. The Bellman-Ford Method
- 2.4.2. The D'Esopo-Pape Algorithm
- 2.4.3. The SLF and LLL Algorithms
- 2.4.4. The Threshold Algorithm
- 2.4.5. Comparison of Label Setting and Label Correcting

2.5. Single Origin/Single Destination Methods
- 2.5.1. Label Setting
- 2.5.2. Label Correcting

2.6. Auction Algorithms

2.7. Multiple Origin/Multiple Destination Methods

2.8. Notes, Sources, and Exercises

The shortest path problem is a classical and important combinatorial problem that arises in many contexts. We are given a directed graph $(\mathcal{N}, \mathcal{A})$ with nodes numbered $1, \ldots, N$. Each arc $(i, j) \in \mathcal{A}$ has a cost or "length" a_{ij} associated with it. The length of a forward path $(i_1, i_2, \ldots, i_k)$ is the length of its arcs

$$\sum_{n=1}^{k-1} a_{i_n i_{n+1}}.$$

This path is said to be *shortest* if it has minimum length over all forward paths with the same origin and destination nodes. The length of a shortest path is also called the *shortest distance*. The shortest path problem deals with finding shortest distances between selected pairs of nodes. [Note that here we are optimizing over forward paths; when we refer to a path (or a cycle) in connection with the shortest path problem, we implicitly assume that the path (or the cycle) is forward.]

The range of applications of the shortest path problem is very broad. In the next section, we will provide some representative examples. We will then develop a variety of algorithms. Most of these algorithms can be viewed as primal cost or dual cost improvement algorithms for an appropriate special case of the minimum cost flow problem, as we will see later. However, the shortest path problem is simple, so we will discuss it based on first principles, and without much reference to cost improvement. This serves a dual purpose. First, it provides an opportunity to illustrate some basic graph concepts in the context of a problem that is simple and rich in intuition. Second, it allows the early development of some ideas and results that will be used later in a variety of other algorithmic contexts.

2.1 PROBLEM FORMULATION AND APPLICATIONS

The shortest path problem appears in a large variety of contexts. We discuss a few representative applications.

Example 2.1. Routing in Data Networks

Data network communication involves the use of a network of computers (nodes) and communication links (arcs) that transfer packets (groups of bits) from their origins to their destinations. The most common method for selecting the path of travel (or route) of packets is based on a shortest path formulation. In particular, each communication link is assigned a positive scalar which is viewed as its length. A shortest path routing algorithm routes each packet along a minimum length (or shortest) path between the origin and destination nodes of the packet.

There are several possibilities for selecting the link lengths. The simplest is for each link to have unit length, in which case a shortest path is simply a path with minimum number of links. More generally, the length of a link, may depend on its transmission capacity and its projected traffic load. The idea here is that a shortest path should contain relatively few and uncongested links, and therefore be desirable for routing. Sophisticated routing algorithms also allow the length of each link to change over time and to depend on the prevailing congestion level of the link. Then a shortest path may adapt to temporary overloads and route packets around points of congestion. Within this context, the shortest path routing algorithm operates continuously, solving the shortest path problem with lengths that vary over time.

A peculiar feature of shortest path routing algorithms is that they are often implemented using distributed and asynchronous communication and computation. In particular, each node of the communication network monitors the traffic conditions of its adjacent links, calculates estimates of its shortest distances to various destinations, and passes these estimates to other nodes who adjust their own estimates, etc. This process is based on standard shortest path algorithms that will be discussed in this chapter, but it is also executed asynchronously, and with out-of-date information because of communication delays between the nodes. Despite this fact, it turns out that these distributed asynchronous algorithms maintain much of the validity of their synchronous counterparts (see the textbooks by Bertsekas and Tsitsiklis [1989], and Bertsekas and Gallager [1992] for related analysis).

There is an important connection between shortest path problems and problems of deterministic discrete-state dynamic programming, which involve sequential decision making over a finite number of time periods. The following example shows that dynamic programming problems can be formulated as shortest path problems. The reverse is also possible; that is, any shortest path problem can be formulated as a dynamic programming problem (see e.g., Bertsekas [1995a], Ch. 2).

Example 2.2. Dynamic Programming

Here we have a discrete-time dynamic system involving N stages. The state of the system at the start of the kth stage is denoted by x_k and takes values in a given finite set, which may depend on the index k. The initial state x_0 is given. During the kth stage, the state of the system changes from x_k to x_{k+1} according to an equation of the form

$$x_{k+1} = f_k(x_k, u_k), \tag{2.1}$$

where u_k is a control that takes values from a given finite set, which may depend on the index k. This transition involves a cost $g_k(x_k, u_k)$. The final transition from x_{N-1} to x_N, involves an additional terminal cost $G(x_N)$. Here, the functions f_k, g_k, and G are given.

Given a control sequence $(u_0, \ldots, u_{N-1})$, the corresponding state sequence $(x_0, \ldots, x_N)$ is determined from the given initial state x_0 and the system of Eq. (2.1). The objective in dynamic programming is to find a control sequence and a corresponding state sequence such that the total cost

$$G(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k)$$

is minimized.

For an example, consider an inventory system that operates over N time periods, and let x_k and u_k denote the number of items held in stock and number of items purchased at the beginning of period k, respectively. We require that u_k be an integer from a given range $[0, r_k]$. We assume that the stock evolves according to the equation

$$x_{k+1} = x_k + u_k - v_k,$$

where v_k is a known integer demand for period k; this is the system equation [cf. Eq. (2.1)]. A negative x_k here indicates unsatisfied demand that is backordered. A common type of cost used in inventory problems has the form

$$g_k(x_k, u_k) = h_k(x_k) + c_k u_k,$$

where c_k is a given cost per unit stock at period k, and $h_k(x_k)$ is a cost either for carrying excess inventory ($x_k > 0$) or for backordering demand ($x_k < 0$). For example $h_k(x_k) = \max\{a_k x_k, -b_k x_k\}$ or $h_k(x_k) = d_k x_k^2$, where a_k, b_k, and d_k are positive scalars, are both reasonable choices for cost function. Finally, we could take $G(x_N) = 0$ to indicate that the final stock x_N has no value [otherwise $G(x_N)$ indicates the cost (or negative salvage value) of x_N]. The objective in this problem is roughly to determine the sequence of purchases over time to minimize the costs of excess inventory and backordering demand over the N time periods.

To convert the dynamic programming problem to a shortest path problem, we introduce a graph such as the one of Fig. 2.1, where the arcs correspond to transitions between states at successive stages and each arc has a cost associated with it. To handle the final stage, we also add an artificial terminal node t. Each state x_N at stage N is connected to the terminal node t with an arc having cost $G(x_N)$. Control sequences correspond to paths originating at the initial state x_0 and terminating at one of the nodes corresponding to the final stage N. The optimal control sequence corresponds to a shortest path from node x_0 to node t. For an extensive treatment of dynamic programming and associated shortest path algorithms we refer to Bertsekas [1995a].

Shortest path problems arise often in contexts of scheduling and sequencing. The following two examples are typical.

Example 2.3. Project Management

Consider the planning of a project involving several activities, some of which must be completed before others can begin. The duration of each activity is

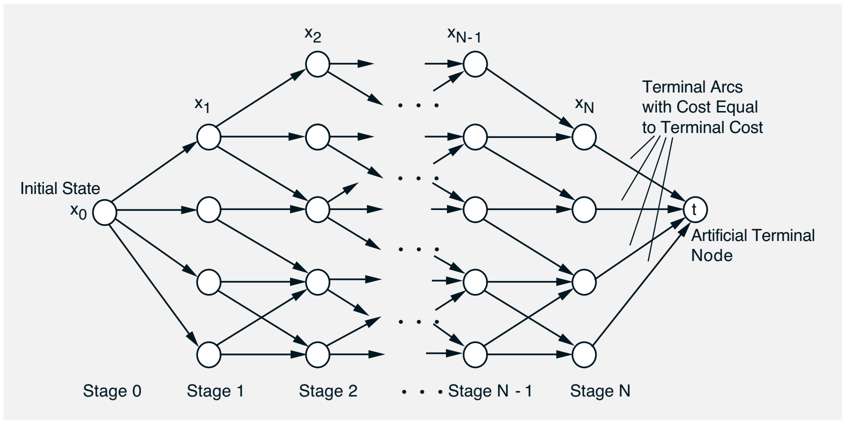


Figure 2.1: Converting a deterministic finite-state N-stage dynamic programming problem to a shortest path problem. Nodes correspond to states. An arc with start and end nodes x_k and x_{k+1}, respectively, corresponds to a transition of the form $x_{k+1} = f_k(x_k, u_k)$. The length of this arc is equal to the cost of the corresponding transition $g_k(x_k, u_k)$. The problem is equivalent to finding a shortest path from the initial state/node x_0 to the artificial terminal node t. Note that the state space and the possible transitions between states may depend on the stage index k.

known in advance. We want to find the time required to complete the project, as well as the *critical* activities, those that even if slightly delayed will result in a corresponding delay of completion of the overall project.

The problem can be represented by a graph where nodes represent completion of some phase of the project (cf. Fig. 2.2). An arc (i, j) represents an activity that starts once phase i is completed and has known duration $t_{ij} > 0$. A phase (node) j is completed when all activities or arcs (i, j) that are incoming to j are completed. Two special nodes 1 and N represent the start and end of the project, respectively. Node 1 has no incoming arcs, while node N has no outgoing arcs. Furthermore, there is at least one path from node 1 to every other node. An important characteristic of an activity network is that it is acyclic. This is inherent in the problem formulation and the interpretation of nodes as phase completions.

For any path $p = \big\{(1, j_1), (j_1, j_2,), \ldots, (j_k, i)\big\}$ from node 1 to a node i, let D_p be the duration of the path defined as the sum of durations of its activities; that is,

$$D_p = t_{ij_1} + t_{j_1 j_2} + \cdots + t_{j_k i}.$$

Then the time T_i required to complete phase i is

$$T_i = \max_{\substack{\text{paths } p \\ \text{from } 1 \text{ to } i}} D_p.$$

The maximum above is attained by *some* path because there can be only a finite number of paths from 1 to i, since the network is acyclic. Thus to find

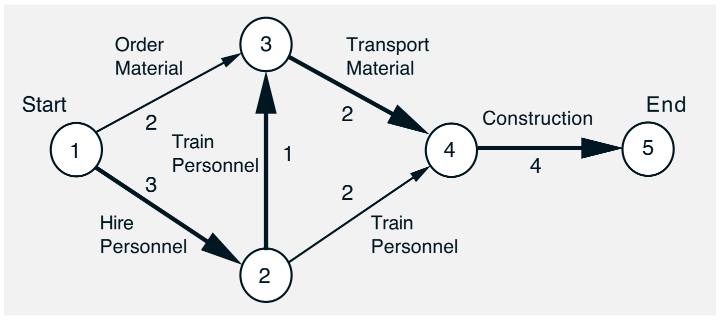


Figure 2.2: Example graph of an activity network. Arcs (i, j) represent activities and are labeled by the corresponding duration t_{ij}. Nodes represent completion of some phase of the project. A phase is completed if all activities associated with incoming arcs at the corresponding node are completed. The project is completed when all phases are completed. The project duration time is the longest sum of arc durations over paths that start at node 1 and end at node 5. The path of longest duration, also called a *critical path*, is shown with thick line. Because the graph is acyclic, finding this path is a shortest path problem with the length of each arc (i, j) being $-t_{ij}$. Activities on the critical path have the property that if any one of them is delayed, a corresponding delay of completion of the overall project will result.

T_i, we should find the *longest* path from 1 to i. Because the graph is acyclic, this problem may also be viewed as a shortest path problem with the length of each arc (i, j) being $-t_{ij}$. In particular, finding the duration of the project is equivalent to finding the shortest path from 1 to N. For further discussion of project management problems, we refer to the literature, e.g., the textbook by Elmaghraby [1978].

Example 2.4. The Paragraphing Problem

This problem arises in a word processing context, where we want to break down a given paragraph consisting of N words into lines for "optimal" appearance and readability. Suppose that we have a heuristic rule, which assigns to any sequence of words a cost that expresses the undesirability of grouping these words together in a line. Based on such a rule, we can assign a cost c_{ij} to a line starting with word i and ending with word $j-1$ of the given paragraph. An optimally divided paragraph is one for which the sum of the costs of its lines is minimal.

We can formulate this as a shortest path problem. There are N nodes, which correspond to the N words of the paragraph, and there is an arc (i, j) with cost c_{ij} connecting any two words i and j with $i < j$. The arcs of the shortest path from node/word 1 to node/word N correspond to the lines of the optimally broken down paragraph.

The exercises contain a number of additional examples that illustrate the broad range of applications of the shortest path problem.

2.2 A GENERIC SHORTEST PATH ALGORITHM

The shortest path problem can be posed in a number of ways; for example, finding a shortest path from a single origin to a single destination, or finding a shortest path from each of several origins to each of several destinations. We focus initially on problems with a single origin and many destinations. For concreteness, we take the origin node to be node 1. The arc lengths a_{ij} are given scalars. They may be negative and/or noninteger, although on occasion we will assume in our analysis that they are nonnegative and/or integer, in which case we will state so explicitly.

In this section, we develop a broad class of shortest path algorithms for the single origin/all destinations problem. These algorithms maintain and adjust a vector $(d_1, d_2, \ldots, d_N)$, where each d_j, called the *label of node* j, is either a scalar or ∞. The use of labels is motivated by a simple optimality condition, which is given in the following proposition.

Proposition 2.1: Let $d_1, d_2, \ldots, d_N$ be scalars satisfying

$$d_j \leq d_i + a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{2.2}$$

and let P be a path starting at a node i_1 and ending at a node i_k. If

$$d_j = d_i + a_{ij}, \qquad \text{for all arcs } (i,j) \text{ of } P, \tag{2.3}$$

then P is a shortest path from i_1 to i_k.

Proof: By adding Eq. (2.3) over the arcs of P, we see that the length of P is equal to the difference $d_{i_k} - d_{i_1}$ of labels of the end node and start node of P. By adding Eq. (2.2) over the arcs of any other path P' starting at i_1 and ending at i_k, we see that the length of P' must be no less than $d_{i_k} - d_{i_1}$. Therefore, P is a shortest path. **Q.E.D.**

The conditions (2.2) and (2.3) are called the *complementary slackness (CS) conditions for the shortest path problem*. This terminology is motivated by the connection of the shortest path problem with the minimum cost flow problem (cf. Section 1.2.1); we will see in Chapter 4 that the CS conditions of Prop. 2.1 are a special case of a general optimality condition (also called CS condition) for the equivalent minimum cost flow problem (in fact they are a special case of a corresponding CS condition for general linear programs; see e.g., Bertsimas and Tsitsiklis [1997], Dantzig [1963]). Furthermore, we will see that the scalars d_i in Prop. 2.1 are related to dual variables.

Let us now describe a prototype shortest path method that contains several interesting algorithms as special cases. In this method, we start with some vector of labels $(d_1, d_2, \ldots, d_N)$, we successively select arcs (i, j) that violate the CS condition (2.2), i.e., $d_j > d_i + a_{ij}$, and we set

$$d_j := d_i + a_{ij}.$$

This is continued until the CS condition $d_j \leq d_i + a_{ij}$ is satisfied for all arcs (i, j).

A key idea is that, in the course of the algorithm, d_i can be interpreted for all i as the length of some path P_i from 1 to i.† Therefore, if $d_j > d_i + a_{ij}$ for some arc (i, j), the path obtained by extending path P_i by arc (i, j), which has length $d_i + a_{ij}$, is a better path than the current path P_j, which has length d_j. Thus, the algorithm finds successively better paths from the origin to various destinations.

Instead of selecting arcs in arbitrary order to check violation of the CS condition, it is usually most convenient and efficient to select nodes, one-at-a-time according to some order, and simultaneously check violation of the CS condition for all of their outgoing arcs. The corresponding algorithm, referred to as *generic*, maintains a list of nodes V, called the *candidate list*, and a vector of labels $(d_1, d_2, \ldots, d_N)$, where each d_j is either a real number or ∞. Initially,

$$V = \{1\},$$

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\, i \neq 1.$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows:

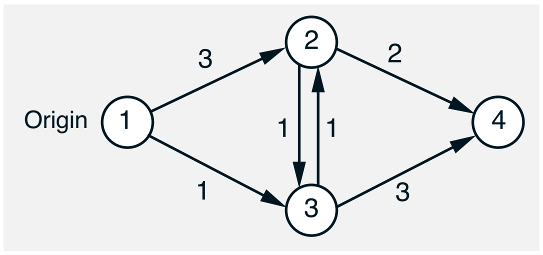
Iteration of the Generic Shortest Path Algorithm

Remove a node i from the candidate list V. For each outgoing arc $(i, j) \in \mathcal{A}$, if $d_j > d_i + a_{ij}$, set

$$d_j := d_i + a_{ij}$$

and add j to V if it does not already belong to V.

† In the case of the origin node 1, we will interpret the label d_1 as either the length of a cycle that starts and ends at 1, or (in the case $d_1 = 0$) the length of the trivial "path" from 1 to itself.



Iteration #	**Candidate List V**	**Node Labels**	**Node out of V**
1	$\{1\}$	$(0,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,3,1,\infty)$	2
3	$\{3,4\}$	$(0,3,1,5)$	3
4	$\{4,2\}$	$(0,2,1,4)$	4
5	$\{2\}$	$(0,2,1,4)$	2
	$\emptyset$	$(0,2,1,4)$	

Figure 2.3: Illustration of the generic shortest path algorithm. The numbers next to the arcs are the arc lengths. Note that node 2 enters the candidate list twice. If in iteration 2 node 3 was removed from V instead of node 2, each node would enter V only once. Thus, the order in which nodes are removed from V is significant.

It can be seen that, in the course of the algorithm, the labels are monotonically nonincreasing. Furthermore, we have

$$d_i < \infty \qquad \Longleftrightarrow \qquad i \text{ has entered } V \text{ at least once.}$$

Figure 2.3 illustrates the algorithm. The following proposition gives its main properties.

Proposition 2.2: Consider the generic shortest path algorithm.

(a) At the end of each iteration, the following conditions hold:

(i) If $d_j < \infty$, then d_j is the length of some path that starts at 1 and ends at j.

(ii) If $i \notin V$, then either $d_i = \infty$ or else

$$d_j \leq d_i + a_{ij}, \qquad \forall\ j \text{ such that } (i,j) \in \mathcal{A}.$$

(b) If the algorithm terminates, then upon termination, for all j with $d_j < \infty$, d_j is the shortest distance from 1 to j and

$$d_j = \begin{cases} \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\} & \text{if } j \neq 1, \\ 0 & \text{if } j = 1. \end{cases} \tag{2.4}$$

Furthermore, upon termination we have $d_j = \infty$ if and only if there is no path from 1 to j.

(c) If the algorithm does not terminate, then there exists some node j and a sequence of paths that start at 1, end at j, and have lengths that diverge to $-\infty$.

(d) The algorithm terminates if and only if there is no path that starts at 1 and contains a cycle with negative length.

Proof: (a) We prove (i) by induction on the iteration count. Indeed, (i) holds at the end of the first iteration since the nodes $j \neq 1$ with $d_j < \infty$ are those for which $(1, j)$ is an arc and their labels are $d_j = a_{1j}$, while for the origin 1, we have $d_1 = 0$, which by convention is viewed as the length of the trivial "path" from 1 to itself. Suppose that (i) holds at the start of some iteration at which the node removed from V is i. Then $d_i < \infty$ (which is true for all nodes of V by the rules of the algorithm), and (by the induction hypothesis) d_i is the length of some path P_i starting at 1 and ending at i. When a label d_j changes as a result of the iteration, d_j is set to $d_i + a_{ij}$, which is the length of the path consisting of P_i followed by arc (i, j). Thus property (i) holds at the end of the iteration, completing the induction proof.

To prove (ii), note that for any i, each time i is removed from V, the condition $d_j \leq d_i + a_{ij}$ is satisfied for all $(i, j) \in \mathcal{A}$ by the rules of the algorithm. Up to the next entrance of i into V, d_i stays constant, while the labels d_j for all j with $(i, j) \in \mathcal{A}$ cannot increase, thereby preserving the condition $d_j \leq d_i + a_{ij}$.

(b) We first introduce the sets

$$I = \{i \mid d_i < \infty \text{ upon termination}\},$$

$$\overline{I} = \{i \mid d_i = \infty \text{ upon termination}\},$$

and we show that we have $j \in \overline{I}$ if and only if there is no path from 1 to j. Indeed, if $i \in I$, we have $d_i < \infty$ and therefore $d_j < \infty$ for all j such that (i, j) is an arc in view of condition (ii) of part (a), so that $j \in I$. It follows that there is no path from any node of I (and in particular, node 1) to any node of $\overline{I}$. Conversely, if there is no path from 1 to j, it follows from condition (i) of part (a) that we cannot have $d_j < \infty$ upon termination, so $j \in \overline{I}$.

We show now that for all $j \in I$, upon termination, d_j is the shortest distance from 1 to j and Eq. (2.4) holds. Indeed, conditions (i) and (ii) of part (a) imply that upon termination we have, for all $i \in I$,

$$d_j \leq d_i + a_{ij}, \qquad \forall\ j \text{ such that } (i,j) \in \mathcal{A}, \tag{2.5}$$

while d_i is the length of some path from 1 to i, denoted P_i. Fix a node $m \in I$, and consider any path P from 1 to m. By adding the condition (2.5) over the arcs of P, we see that the length of P is no less than $d_m - d_1$, which is less or equal to d_m (we have $d_1 \leq 0$, since initially $d_1 = 0$ and all node labels are monotonically nonincreasing). Hence P_m is a shortest path from 1 to m and the shortest distance is d_m. Furthermore, the equality $d_j = d_i + a_{ij}$ must hold for all arcs (i,j) on the shortest paths P_m, $m \in I$, implying that $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$ for all $j \in I$ with $j \neq 1$, while $d_1 = 0$.

(c) If the algorithm never terminates, some label d_j must decrease strictly an infinite number of times, generating a corresponding sequence of distinct paths P_j as per condition (i) of part (a). Each of these paths can be decomposed into a simple path from 1 to j plus a collection of simple cycles, as in Exercise 1.4 of Chapter 1. Since the number of simple paths from 1 to j is finite, and the length of P_j is monotonically decreasing, it follows that P_j eventually must involve a cycle with negative length. By replicating this cycle a sufficiently large number of times, one can obtain paths from 1 to j with arbitrarily small length.

(d) Using part (c), we have that the algorithm will terminate if and only if there is a lower bound on the length of all paths that start at node 1. Thus, the algorithm will terminate if and only if there is no path that starts at node 1 and contains a cycle with negative length. **Q.E.D.**

When some arc lengths are negative, Prop. 2.2 points to a way to detect existence of a path that starts at the origin 1 and contains a cycle of negative length. If such a path exists, it can be shown under mild assumptions that the label of at least one node will diverge to $-\infty$ (see Exercise 2.32). We can thus monitor whether for some j we have

$$d_j < (N-1) \min_{(i,j)\in\mathcal{A}} a_{ij}.$$

When this condition occurs, the path from 1 to j whose length is equal to d_j [as per Prop. 2.2(a)] must contain a negative cycle [if it were simple, it would consist of at most $N-1$ arcs, and its length could not be smaller than $(N-1)\min_{(i,j)\in\mathcal{A}} a_{ij}$; a similar argument would apply if it were not simple but it contained only cycles of nonnegative length].

Bellman's Equation and Shortest Path Construction

When all cycles have nonnegative length and there exists a path from node 1 to every node j, then Prop. 2.2 shows that the generic algorithm terminates and that, upon termination, all labels are equal to the corresponding shortest distances, and satisfy $d_1 = 0$ and

$$d_j = \min_{(i,j)\in\mathcal{A}} \{d_i + a_{ij}\}, \qquad \forall\ j \neq 1. \tag{2.6}$$

This is known as *Bellman's equation* and it has an intuitive meaning: it indicates that the shortest distance from 1 to j is obtained by optimally choosing the predecessor i of node j in order to minimize the sum of the shortest distance from 1 to i and the length of arc (i, j). It also indicates that if P_j is a shortest path from 1 to j, and a node i belongs to P_j, then the portion of P_j from 1 to i, is a shortest path from 1 to i.

From Bellman's equation, we can obtain the shortest paths (in addition to the shortest path lengths) if all cycles not including node 1 have strictly positive length. To do this, select for each $j \neq 1$ one arc (i, j) that attains the minimum in $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$ and consider the subgraph consisting of these $N - 1$ arcs; see Fig. 2.4. To find the shortest path to any node j, start from j and follow the corresponding arcs of the subgraph backward until node 1 is reached. Note that the same node cannot be reached twice before node 1 is reached, since a cycle would be formed that, on the basis of Eqs. (2.6), would have zero length. [To see this, let $(i_1, i_2, \ldots, i_k, i_1)$ be the cycle and add the equations

$$d_{i_1} = d_{i_2} + a_{i_2 i_1}$$

$$\cdots$$

$$d_{i_{k-1}} = d_{i_k} + a_{i_k i_{k-1}}$$

$$d_{i_k} = d_{i_1} + a_{i_1 i_k}$$

obtaining $a_{i_2 i_1} + \cdots + a_{i_k i_{k-1}} + a_{i_1 i_k} = 0$.] Since the subgraph is connected and has $N - 1$ arcs, it must be a spanning tree. We call this subgraph a *shortest path spanning tree*, and we note its special structure: it has a root (node 1) and every arc of the tree is directed away from the root. The preceding argument can also be used to show that Bellman's equation has no solution other than the shortest distances; see Exercise 2.5.

A shortest path spanning tree can also be constructed in the process of executing the generic shortest path algorithm by recording the arc (i, j) every time d_j is decreased to $d_i + a_{ij}$; see Exercise 2.4.

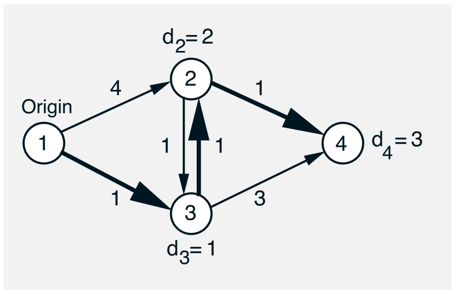


Figure 2.4: Example of construction of shortest path spanning tree. The arc lengths are shown next to the arcs, and the shortest distances are shown next to the nodes. For each $j \neq 1$, we select an arc (i, j) such that

$$d_j = d_i + a_{ij}$$

and we form the shortest path spanning tree. The arcs selected in this example are $(1, 3)$, $(3, 2)$, and $(2, 4)$.

Advanced Initialization

The generic algorithm need not be started with the initial conditions

$$V = \{1\}, \qquad d_1 = 0, \qquad d_i = \infty, \quad \forall\, i \neq 1,$$

in order to work correctly. Any set of labels $(d_1, \ldots, d_N)$ and candidate list V can be used initially, as long as they satisfy the conditions of Prop. 2.2(a). It can be seen that the proof of the remaining parts of Prop. 2.2 go through under these conditions.

In particular, the algorithm works correctly if the labels and the candidate list are initialized so that $d_1 = 0$ and:

(a) For each node i, d_i is either ∞ or else it is the length of a path from 1 to i.

(b) The candidate list V contains all nodes i such that

$$d_i + a_{ij} < d_j \text{ for some } (i, j) \in \mathcal{A}. \tag{2.7}$$

This kind of initialization is very useful in reoptimization contexts, where we have to solve a large number of similar problems that differ slightly from each other; for example they may differ by just a few arc lengths or they may have a slightly different node set. The lengths of the shortest paths of one problem can be used as the starting labels for another problem, and substantial computational savings may be obtained, because it is likely that many of the nodes will maintain their shortest path lengths and will never enter V.

Another important situation where an advanced initialization is very useful arises if, by using heuristics or an available solution of a similar shortest path problem, we can construct a set of "good" paths from node 1 to the other nodes. Then we can use the lengths of these paths as the initial labels in the generic shortest path algorithm and start with a candidate list consisting of the nodes where the CS condition is violated [cf. Eq. (2.7)].

Finally, let us note another technique that is sometimes useful in reoptimization settings. Suppose that we have some scalars $\delta_1, \ldots, \delta_N$ and we change the arc lengths to

$$\hat{a}_{ij} = a_{ij} + \delta_i - \delta_j.$$

Then it can be seen that the length of any path from a node m to a node n will be increased by $\delta_m - \delta_n$, while the shortest paths will be unaffected. Thus it may be advantageous to use the modified arc lengths $\hat{a}_{ij}$ instead of the original lengths a_{ij}, if this will enhance the application of a suitable shortest path algorithm. For example, we may be able with proper choice of δ_i, to reduce the arc cost range $\max_{(i,j)} |\hat{a}_{ij}|$ (this is helpful in some algorithms) or to make $\hat{a}_{ij}$ nonnegative (see Section 2.7 for an application of this idea).

Implementations of the Generic Algorithm

There are many implementations of the generic algorithm. They differ in how they select the node to be removed from the candidate list V, and they are broadly divided into two categories:

(a) *Label setting methods.* In these methods, the node i removed from V is a node with minimum label. Under the assumption that *all arc lengths are nonnegative*, these methods have a remarkable property: each node will enter V at most *once*, as we will show shortly; its label has its permanent or final value at the first time it is removed from V. The most time-consuming part of these methods is calculating the minimum label node in V at each iteration; there are several implementations, that use a variety of creative procedures to obtain this minimum.

(b) *Label correcting methods.* In these methods the choice of the node i removed from V is less sophisticated than in label setting methods, and requires less calculation. However, a node may enter V multiple times.

There are several worst-case complexity bounds for label setting and label correcting methods. The best bounds for the case of nonnegative arc lengths correspond to label setting methods. The best practical methods, however, are not necessarily the ones with the best complexity bounds, as will be discussed in the next two sections.

In practice, when the arc lengths are nonnegative, the best label setting methods and the best label correcting methods are competitive. As a general rule, a sparse graph favors the use of a label correcting over a label setting method for reasons that will be explained later (see the discussion at the end of Section 2.4). An important advantage of label correcting methods is that they are more general, since they do not require nonnegativity of the arc lengths.

2.3 LABEL SETTING (DIJKSTRA) METHODS

In this section we discuss various implementations of the label setting approach. The prototype label setting method, first published by Dijkstra [1959] but also discovered independently by several other researchers, is the special case of the generic algorithm where the node i removed from the candidate list V at each iteration has minimum label, that is,

$$d_i = \min_{j \in V} d_j.$$

For convenient reference, let us state this method explicitly.

Initially, we have

$$V = \{1\},$$

$$d_1 = 0, \qquad d_i = \infty, \quad \forall\, i \neq 1.$$

The method proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows:

Iteration of the Label Setting Method

Remove from the candidate list V a node i such that

$$d_i = \min_{j \in V} d_j.$$

For each outgoing arc $(i, j) \in \mathcal{A}$, if $d_j > d_i + a_{ij}$, set

$$d_j := d_i + a_{ij}$$

and add j to V if it does not already belong to V.

Figure 2.5 illustrates the label setting method. Some insight into the method can be gained by considering the set W of nodes that have already been in V but are not currently in V:

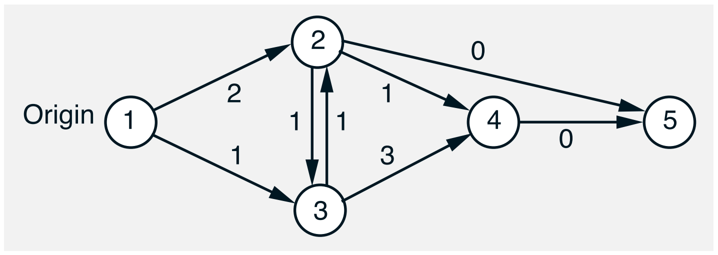
$$W = \{i \mid d_i < \infty, i \notin V\}.$$

We will prove later, in Prop. 2.3(a), that as a consequence of the policy of removing from V a minimum label node, W contains nodes with "small" labels throughout the algorithm, in the sense that

$$d_j \leq d_i, \qquad \text{if } j \in W \text{ and } i \notin W. \tag{2.8}$$

Using this property and the assumption $a_{ij} \geq 0$, it can be seen that when a node i is removed from V, we have, for all $j \in W$ for which (i, j) is an arc,

$$d_j \leq d_i + a_{ij}.$$



Iteration #	**Candidate List V**	**Node Labels**	**Node out of V**
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	3
3	$\{2,4\}$	$(0,2,1,4,\infty)$	2
4	$\{4,5\}$	$(0,2,1,3,2)$	5
5	$\{4\}$	$(0,2,1,3,2)$	4
	$\emptyset$	$(0,2,1,3,2)$	

Figure 2.5: Example illustrating the label setting method. At each iteration, the node with the minimum label is removed from V. Each node enters V only once.

Hence, once a node enters W, it stays in W and its label does not change further. Thus, W can be viewed as the set of *permanently labeled nodes*, that is, the nodes that have acquired a final label, which by Prop. 2.2, must be equal to their shortest distance from the origin.

The following proposition makes the preceding argument precise and proves some additional facts.

Proposition 2.3: Assume that all arc lengths are nonnegative.

(a) For any iteration of the label setting method, the following hold for the set

$$W = \{i \mid d_i < \infty, i \notin V\}. \tag{2.9}$$

(i) No node belonging to W at the start of the iteration will enter the candidate list V during the iteration.

(ii) At the end of the iteration, we have $d_i \leq d_j$ for all $i \in W$ and $j \notin W$.

(iii) For each node j, consider simple paths that start at 1, end at j, and have all their other nodes in W at the end of the iteration. Then the label d_j at the end of the iteration is equal to the length of the shortest of these paths ($d_j = \infty$ if no such path exists).

(b) The label setting method will terminate, and all nodes with a final label that is finite will be removed from the candidate list V exactly once in order of increasing shortest distance from node 1; that is, if the final labels of i and j are finite and satisfy $d_i < d_j$, then i will be removed before j.

Proof: (a) Properties (i) and (ii) will be proved simultaneously by induction on the iteration count. Clearly (i) and (ii) hold for the initial iteration at which node 1 exits V and enters W.

Suppose that (i) and (ii) hold for iteration $k-1$, and suppose that during iteration k, node i satisfies $d_i = \min_{j \in V} d_j$ and exits V. Let W and $\overline{W}$ be the set of Eq. (2.9) at the start and at the end of iteration k, respectively. Let d_j and $\overline{d}_j$ be the label of each node j at the start and at the end of iteration k, respectively. Since by the induction hypothesis we have $d_j \leq d_i$ for all $j \in W$, and $a_{ij} \geq 0$ for all arcs (i,j), it follows that $d_j \leq d_i + a_{ij}$ for all arcs (i,j) with $j \in W$. Hence, a node $j \in W$ cannot enter V at iteration k. This completes the induction proof of property (i), and shows that

$$\overline{W} = W \cup \{i\}.$$

Thus, at iteration k, the only labels that may change are the labels d_j of nodes $j \notin \overline{W}$ such that (i,j) is an arc; the label $\overline{d}_j$ at the end of the iteration will be $\min\{d_j, d_i + a_{ij}\}$. Since $a_{ij} \geq 0$, $d_i \leq d_j$ for all $j \notin W$, and $d_i = \overline{d}_i$, we must have $\overline{d}_i \leq \overline{d}_j$ for all $j \notin \overline{W}$. Since by the induction hypothesis we have $d_m \leq d_i$ and $d_m = \overline{d}_m$ for all $m \in \overline{W}$, it follows that $\overline{d}_m \leq \overline{d}_j$ for all $m \in \overline{W}$ and $j \notin \overline{W}$. This completes the induction proof of property (ii).

To prove property (iii), choose any node j and consider the subgraph consisting of the nodes $W \cup \{j\}$ together with the arcs that have both end nodes in $W \cup \{j\}$. Consider also a modified shortest path problem involving this subgraph, and the same origin and arc lengths as in the original shortest path problem. In view of properties (i) and (ii), the label setting method applied to the modified shortest path problem yields the same sequence of nodes exiting V and the same sequence of labels as when applied to the original problem up to the current iteration. By Prop. 2.2, the label setting method for the modified problem terminates with the labels equal to the shortest distances of the modified problem at the current iteration. This means that the labels at the end of the iteration have the property stated in the proposition.

(b) Since there is no cycle with negative length, by Prop. 2.2(d), we see that the label setting method will terminate. At each iteration the node removed from V is added to W, and according to property (i) (proved above), no node from W is ever returned to V. Therefore, each node with a final label that is finite will be removed from V and simultaneously entered in W exactly once, and, by the rules of the algorithm, its label cannot change after its entrance in W. Property (ii) then shows that each new node added to W has a label at least as large as the labels of the nodes already in W. Therefore, the nodes are removed from V in the order stated in the proposition. **Q.E.D.**

2.3.1 Performance of Label Setting Methods

In label setting methods, the candidate list V is typically maintained with the help of some data structure that facilitates the removal and the addition of nodes, and also facilitates finding the minimum label node from the list. The choice of data structure is crucial for good practical performance as well as for good theoretical worst-case performance.

To gain some insight into this, we first consider a somewhat naive implementation that will serve as a yardstick for comparison. By Prop. 2.3, there will be exactly N iterations, and in each of these, the candidate list V will be searched for a minimum label node. Suppose this is done by examining all nodes in sequence, checking whether they belong to V, and finding one with minimum label among those who do. Searching V in this way requires $O(N)$ operations per iteration, for a total of $O(N^2)$ operations. Also during the algorithm, we must examine each arc (i,j) exactly once to check whether the condition $d_j > d_i + a_{ij}$ holds, and to set $d_j := d_i + a_{ij}$ if it does. This requires $O(A)$ operations, which is dominated by the preceding $O(N^2)$ estimate.

The $O(A)$ operation count for arc examination is unavoidable and cannot be reduced [each arc (i,j) must be checked at least once just to verify the optimality condition $d_j \leq d_i + a_{ij}$]. However, the $O(N^2)$ operation count for minimum label searching can be reduced considerably by using appropriate data structures. The best estimates of the worst-case running time that have been thus obtained are $O(A+N\log N)$ and $O(A+N\sqrt{\log C})$, where C is the arc length range $C = \max_{(i,j)\in\mathcal{A}} a_{ij}$; see Fredman and Tarjan [1984], and Ahuja, Mehlhorn, Orlin, and Tarjan [1990]. On the basis of present experience, however, the implementations that perform best in practice have considerable less favorable running time estimates. The explanation for this is that the $O(\cdot)$ estimates involve a different constant for each method and also correspond to worst-case problem instances. Thus, the worst-case complexity estimates may not provide a reliable practical comparison of various methods. We now discuss two of the most popular implementations of the label setting method.

2.3.2 The Binary Heap Method

Here the nodes are organized as a binary heap on the basis of label values and membership in V; see Fig. 2.6. The node at the top of the heap is the node of V that has minimum label, and the label of every node in V is no larger than the labels of all the nodes that are in V and are its descendants in the heap. Nodes that are not in V may be in the heap but may have no descendants that are in V.

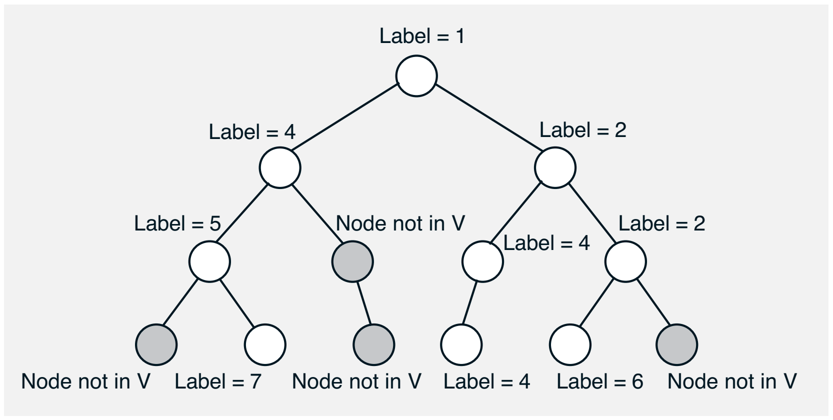


Figure 2.6: A binary heap organized on the basis of node labels is a binary balanced tree such that the label of each node of V is no larger than the labels of all its descendants that are in V. Nodes that are not in V may have no descendants that are in V. The topmost node, called the *root*, has the minimum label. The tree is balanced in that the numbers of arcs in the paths from the root to any nodes with no descendants differ by at most 1. If the label of some node decreases, the node must be moved upward toward the root, requiring $O(\log N)$ operations. [It takes $O(1)$ operations to compare the label of a node i with the label of one of its descendants j, and to interchange the positions of i and j if the label of j is smaller. Since there are $\log N$ levels in the tree, it takes at most $\log N$ such comparisons and interchanges to move a node upward to the appropriate position once its label is decreased.] Similarly, when the topmost node is removed from V, moving the node downward to the appropriate level in the heap requires at most $\log N$ steps and $O(\log N)$ operations. (Each step requires the interchange of the position of the node and the position of one of its descendants. The descendant must be in V for the step to be executed; if both descendants are in V, the one with smaller label is selected.)

At each iteration, the top node of the heap is removed from V. Furthermore, the labels of some nodes already in V may decrease, so these may have to be repositioned in the heap; also, some other nodes may enter

V for the first time and have to be inserted in the heap at the right place. It can be seen that each of these removals, repositionings, and insertions can be done in $O(\log N)$ time. There are a total of N removals and N node insertions, so the number of operations for maintaining the heap is $O\big((N+R)\log N\big)$, where R is the total number of node repositionings. There is at most one repositioning per arc, since each arc is examined at most once, so we have $R \leq A$ and the total operation count for maintaining the heap is $O(A\log N)$. This dominates the $O(A)$ operation count to examine all arcs, so the worst-case running time of the method is $O(A\log N)$. On the other hand, practical experience indicates that the number of node repositionings R is usually a small multiple of N, and considerably less than the upper bound A. Thus, the running time of the method in practice typically grows approximately like $O(A + N\log N)$.

2.3.3 Dial's Algorithm

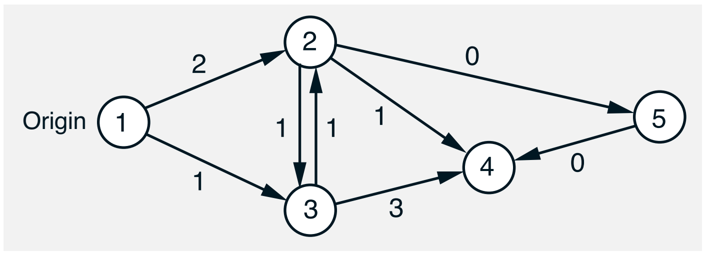
This algorithm, due to Dial [1969], requires that all arc lengths are *nonnegative integers*. It uses a naive yet often surprisingly effective method for finding the minimum label node in V. The idea is to maintain for every possible label value, a list of the nodes that have that value. Since every finite label is equal to the length of some path with no cycles [Prop. 2.3(a), part (iii)], the possible label values range from 0 to $(N-1)C$, where

$$C = \max_{(i,j)\in\mathcal{A}} a_{ij}.$$

Thus, we may scan the $(N-1)C+1$ possible label values (in ascending order) and look for a label value with nonempty list, instead of scanning the candidate list V.

To visualize the algorithm, it is useful to think of each integer in the range $[0,(N-1)C]$ as some kind of container, referred to as a *bucket*. Each bucket b holds the nodes with label equal to b. Tracing steps, we see that the method starts with the origin node 1 in bucket 0 and all other buckets empty. At the first iteration, each node j with $(1,j)\in\mathcal{A}$ enters the candidate list V and is inserted in bucket a_{1j}. After we are done with bucket 0, we proceed to check bucket 1. If it is nonempty, we repeat the process, removing from V all nodes with label 1 and moving other nodes to smaller numbered buckets as required; if not, we check bucket 2, and so on. Figure 2.7 illustrates the method with an example.

Let us now consider the efficient implementation of the algorithm. We first note that a doubly linked list (see Fig. 2.8) can be used to maintain the set of nodes belonging to a given bucket, so that checking the emptiness of a bucket and inserting or removing a node from a bucket are easy, requiring $O(1)$ operations. With such a data structure, the time required for minimum label node searching is $O(NC)$, and the time required for adjusting node labels and repositioning nodes between buckets is $O(A)$. Thus the



Iter. #	**Cand. List V**	**Node Labels**	**Buck. 0**	**Buck. 1**	**Buck. 2**	**Buck. 3**	**Buck. 4**	**Out of V**
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1	–	–	–	–	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	1	3	2	–	–	3
3	$\{2,4\}$	$(0,2,1,4,\infty)$	1	3	2	–	4	2
4	$\{4,5\}$	$(0,2,1,3,2)$	1	3	2,5	4	–	5
5	$\{4\}$	$(0,2,1,2,2)$	1	3	2,4,5	–	–	4
	$\varnothing$	$(0,2,1,2,2)$	1	3	2,4,5	–	–	

Figure 2.7: An example illustrating Dial's method.

overall running time is $O(A+NC)$. The algorithm is pseudopolynomial, but for small values of C (much smaller than N) it performs very well in practice.

In problems where the minimum arc length

$$\overline{a} = \min_{(i,j)\in\mathcal{A}} a_{ij}$$

is greater than 1, the performance of the algorithm can be improved by using a device suggested by Denardo and Fox [1979]. The idea is that the label of a node cannot be reduced below $b+\overline{a}$ while searching bucket b, so that no new nodes will be added to buckets $b+1,\ldots,b+\overline{a}-1$ while searching bucket b. As a result, buckets $b,b+1,\ldots,b+\overline{a}-1$ can be lumped into a single bucket. To take advantage of this idea, we can use

$$\left\lceil \frac{(N-1)C+1}{\overline{a}} \right\rceil$$

buckets, and follow the strategy of placing node i into bucket b if

$$\overline{a}b \le d_i \le \overline{a}(b+1)-1.$$

The running time of the algorithm is then reduced to $O\big(A+(NC/\overline{a})\big)$.

Bucket b	0	1	2	3	4	5	6	7	8
Contents of b	1	–	3,4,5	2,7	–	6	–	–	–
$FIRST(b)$	1	0	3	2	0	6	0	0	0

Node i	1	2	3	4	5	6	7
Label d_i	0	3	2	2	2	5	3
$NEXT(i)$	0	7	4	5	0	0	0
$PREVIOUS(i)$	0	0	0	3	4	0	2

Figure 2.8: Illustration of a doubly linked list data structure to maintain the candidate list V in buckets. In this example, the nodes in V are numbered $1, 2, \ldots, 7$, and the buckets are numbered $0, 1, \ldots, 8$. A node i belongs to bucket b if $d_i = b$.

As shown in the first table, for each bucket b we maintain the first node of the bucket in an array element $FIRST(b)$, where $FIRST(b) = 0$ if bucket b is empty.

As shown in the second table, for every node i we maintain two array elements, $NEXT(i)$ and $PREVIOUS(i)$, giving the next node and the preceding node, respectively, of node i in the bucket where i is currently residing [$NEXT(i) = 0$ or $PREVIOUS(i) = 0$ if i is the last node or the first node in its bucket, respectively].

Another useful idea is that it is sufficient to maintain only $C + 1$ buckets, rather than $(N - 1)C + 1$, thereby significantly saving in memory. The reason is that if we are currently searching bucket b, then all buckets beyond $b + C$ are known to be empty. To see this, note that the label d_j of any node j must be of the form $d_i + a_{ij}$, where i is a node that has already been removed from the candidate list. Since $d_i \leq b$ and $a_{ij} \leq C$, it follows that $d_j \leq b + C$.

The idea of using buckets to maintain the nodes of the candidate list can be generalized considerably. In particular, buckets of width larger than $\max\{1, \min_{(i,j)\in\mathcal{A}} a_{ij}\}$ may be used. This results in fewer buckets to search over, thereby alleviating the $O(NC)$ bottleneck of the running time of the algorithm. There is a price for this, namely the need to search for a minimum label node within the current bucket. This search can be speeded up by using buckets with nonuniform widths, and by breaking down buckets of large width into buckets of smaller width at the right moment. With intelligent strategies of this type, one may obtain label setting methods with very good polynomial complexity bounds; see Johnson [1977], Denardo and Fox [1979], Ahuja, Mehlhorn, Orlin, and Tarjan [1990]. In practice, however, the simpler algorithm of Dial has been more popular than these methods.

2.4 LABEL CORRECTING METHODS

We now turn to the analysis of label correcting methods. In these methods, the selection of the node to be removed from the candidate list V is simpler and requires less overhead than in label setting methods, at the expense of multiple entrances of nodes in V. All of these methods use some type of queue to maintain the candidate list V. They differ in the way the queue is structured, and in the choice of the queue position into which nodes are inserted. In this section, we will discuss some of the most interesting possibilities.

2.4.1 The Bellman-Ford Method

The simplest label correcting method uses a first-in first-out rule to update the queue that is used to store the candidate list V. In particular, a node is always removed from the top of the queue, and a node, upon entrance in the candidate list, is placed at the bottom of the queue. Thus, it can be seen that the method operates in *cycles* of iterations: the first cycle consists of just iterating on node 1; in each subsequent cycle, the nodes that entered the candidate list during the preceding cycle, are removed from the list in the order that they were entered. We will refer to this method as the *Bellman-Ford method*, because it is closely related to a method proposed by Bellman [1957] and Ford [1956] based on dynamic programming ideas (see Exercise 2.6).

The complexity analysis of the method is based on the following property, which we will prove shortly:

Bellman-Ford Property

For each node i and integer $k \geq 1$, let

$$d_i^k = \text{Shortest distance from 1 to } i \text{ using paths that have } k \text{ arcs or less},$$

where $d_i^k = \infty$ if there is no path from 1 to i with k arcs or less. Then the label d_i at the end of the kth cycle of iterations of the Bellman-Ford method is less or equal to d_i^k.

In the case where all cycles have nonnegative length, the shortest distance of every node can be achieved with a path having $N-1$ arcs or less, so the above Bellman-Ford property implies that the method finds all the shortest distances after at most $N-1$ cycles. Since each cycle of iterations requires a total of $O(A)$ operations (each arc is examined at most once in each cycle), the running time of the Bellman-Ford method is $O(NA)$.

To prove the Bellman-Ford property, we first note that

$$d_j^{k+1} = \min\left\{d_j^k,\ \min_{(i,j)\in\mathcal{A}}\{d_i^k + a_{ij}\}\right\}, \qquad \forall\ j,\ k \geq 1, \tag{2.10}$$

since d_j^{k+1} is either the length of a path from 1 to j with k arcs or less, in which case it is equal to d_j^k, or else it is the length of some path that starts at 1 goes to a predecessor node i with k arcs or less, and then goes to j using arc (i,j). We now prove the Bellman-Ford property by induction. At the end of the 1st cycle, we have for all i,

$$d_i = \begin{cases} 0 & \text{if } i = 1, \\ a_{1i} & \text{if } i \neq 1 \text{ and } (1,i) \in \mathcal{A}, \\ \infty & \text{if } i \neq 1 \text{ and } (1,i) \notin \mathcal{A}, \end{cases}$$

while

$$d_i^1 = \begin{cases} a_{1i} & \text{if } (1,i) \in \mathcal{A}, \\ \infty & \text{if } (1,i) \notin \mathcal{A}, \end{cases}$$

so that $d_i \leq d_i^1$ for all i. Let d_i and V be the node labels and the contents of the candidate list at the end of the kth cycle, respectively. Let also $\overline{d}_i$ be the node labels at the end of the $(k+1)$st cycle. We assume that $d_i \leq d_i^k$ for all i, and we will show that $\overline{d}_i \leq d_i^{k+1}$ for all i. Indeed, by condition (ii) of Prop. 2.2(a), we have

$$d_j \leq d_i + a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \notin V,$$

and since $\overline{d}_j \leq d_j$, it follows that

$$\overline{d}_j \leq d_i + a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \notin V. \tag{2.11}$$

We also have

$$\overline{d}_j \leq d_i + a_{ij}, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \in V, \tag{2.12}$$

since at the time when i is removed from V, its current label, call it $\tilde{d}_i$, satisfies $\tilde{d}_i \leq d_i$, and the label of j is set to $\tilde{d}_i + a_{ij}$ if it exceeds $\tilde{d}_i + a_{ij}$. By combining Eqs. (2.11) and (2.12), we see that

$$\overline{d}_j \leq \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\} \leq \min_{(i,j)\in\mathcal{A}}\{d_i^k + a_{ij}\}, \qquad \forall\ j, \tag{2.13}$$

where the second inequality follows by the induction hypothesis. We also have $\overline{d}_j \leq d_j \leq d_j^k$ by the induction hypothesis, so Eq. (2.13) yields

$$\overline{d}_j \leq \min \left\{ d_j^k,\ \min_{(i,j)\in\mathcal{A}} \{d_i^k + a_{ij}\} \right\} = d_j^{k+1},$$

where the last equality holds by Eq. (2.10). This completes the induction proof of the Bellman-Ford property.

The Bellman-Ford method can be used to detect the presence of a negative cycle. Indeed, from Prop. 2.2, we see that the method fails to terminate if and only if there exists a path that starts at 1 and contains a negative cycle. Thus in view of the Bellman-Ford property, such a path exists if and only if the algorithm has not terminated by the end of $N - 1$ cycles.

The best practical implementations of label correcting methods are more sophisticated than the Bellman-Ford method. Their worst-case running time is no better than the $O(NA)$ time of the Bellman-Ford method, and in some cases it is considerably slower. Yet their practical performance is often considerably better. We will discuss next three different types of implementations.

2.4.2 The D'Esopo-Pape Algorithm

In this method, a node is always removed from the top of the queue used to maintain the candidate list V. A node, upon entrance in the queue, is placed at the bottom of the queue if it has never been in the queue before; otherwise it is placed at the top.

The idea here is that when a node i is removed from the queue, its label affects the labels of a subset B_i of the neighbor nodes j with $(i, j) \in \mathcal{A}$. When the label of i changes again, it is likely that the labels of the nodes in B_i will require updating also. It is thus argued that it makes sense to place the node at the top of the queue so that the labels of the nodes in B_i get a chance to be updated as quickly as possible.

While this rationale is not quite convincing, it seems to work well in practice for a broad variety of problems, including types of problems where there are some negative arc lengths. On the other hand, special examples have been constructed (Kershenbaum [1981], Shier and Witzgall [1981]), where the D'Esopo-Pape algorithm performs very poorly. In particular, in these examples, the number of entrances of some nodes in the candidate list V is not polynomial. Computational studies have also shown that for some classes of problems, the practical performance of the D'Esopo-Pape algorithm can be very poor (Bertsekas [1993a]). Pallottino [1984], and Gallo and Pallottino [1988] give a polynomial variant of the algorithm, whose practical performance, however, is roughly similar to the one of the original version.

2.4.3 The SLF and LLL Algorithms

These methods are motivated by the hypothesis that when the arc lengths are nonnegative, the queue management strategy should try to place nodes with small labels near the top of the queue. For a supporting heuristic argument, note that for a node j to reenter V, some node i such that $d_i + a_{ij} < d_j$ must first exit V. Thus, the smaller d_j was at the previous exit of j from V the less likely it is that d_i+a_{ij} will subsequently become less than d_j for some node $i \in V$ and arc (i,j). In particular, if $d_j \leq \min_{i\in V} d_i$ and the arc lengths a_{ij} are nonnegative, it is impossible that subsequent to the exit of j from V we will have $d_i + a_{ij} < d_j$ for some $i \in V$.

We can think of Dijkstra's method as implicitly placing at the top of an imaginary queue the node with the smallest label, thereby resulting in the minimal number N of iterations. The methods of this section attempt to emulate approximately the minimum label selection policy of Dijkstra's algorithm with a much smaller computational overhead. They are primarily suitable for the case of nonnegative arc lengths. While they will work even when there are some negative arc lengths as per Prop. 2.2, there is no reason to expect that in this case they will terminate faster (or slower) than any of the other label correcting methods that we will discuss.

A simple strategy for placing nodes with small label near the top of the queue is the *Small Label First method* (SLF for short). Here the candidate list V is maintained as a double ended queue Q. At each iteration, the node exiting V is the top node of Q. The rule for inserting new nodes is given below:

SLF Strategy

Whenever a node j enters Q, its label d_j is compared with the label d_i of the top node i of Q. If $d_j \leq d_i$, node j is entered at the top of Q; otherwise j is entered at the bottom of Q.

The SLF strategy provides a rule for inserting nodes in Q, but always removes (selects for iteration) nodes from the top of Q. A more sophisticated strategy is to make an effort to remove from Q nodes with small labels. A simple possibility, called the *Large Label Last method* (LLL for short) works as follows: At each iteration, when the node at the top of Q has a larger label than the average node label in Q (defined as the sum of the labels of the nodes in Q divided by the cardinality $|Q|$ of Q), this node is not removed from Q, but is instead repositioned to the bottom of Q.

LLL Strategy

Let i be the top node of Q, and let

$$a = \frac{\sum_{j \in Q} d_j}{|Q|}.$$

If $d_i > a$, move i to the bottom of Q. Repeat until a node i such that $d_i \leq a$ is found and is removed from Q.

It is simple to combine the SLF queue insertion and the LLL node removal strategies, thereby obtaining a method referred to as SLF/LLL.

Experience suggests that, assuming nonnegative arc lengths, the SLF, LLL, and combined SLF/LLL algorithms perform substantially faster than the Bellman-Ford and the D'Esopo-Pape methods. The strategies are also well-suited for parallel computation (see Bertsekas, Guerriero, and Musmanno [1996]). The combined SLF/LLL method consistently requires a smaller number of iterations than either SLF or LLL, although the gain in number of iterations is sometimes offset by the extra overhead.

Regarding the theoretical worst-case performance of the SLF and the combined SLF/LLL algorithms, an example has been constructed by Chen and Powell [1997], showing that these algorithms do not have polynomial complexity in their pure form. However, nonpolynomial behavior seems to be an extremely rare phenomenon in practice. In any case, one may construct polynomial versions of the SLF and LLL algorithms, when the arc lengths are nonnegative. A simple approach is to first sort the outgoing arcs of each node by length. That is, when a node i is removed from Q, first examine the outgoing arc from i that has minimum length, then examine the arc of second minimum length, etc. This approach, due to Chen and Powell [1997], can be shown to have complexity $O(NA^2)$ (see Exercise 2.9). Note, however, that sorting the outgoing arcs of a node by length may involve significant overhead.

There is also another approach to construct polynomial versions of the SLF and LLL algorithms (as well as other label correcting methods), which leads to $O(NA)$ complexity, assuming nonnegative arc lengths. To see how this works, suppose that in the generic label correcting algorithm, there is a set of increasing iteration indices $t_1, t_2, \ldots, t_{n+1}$ such that $t_1 = 1$, and for $i = 1, \ldots, n$, all nodes that are in V at the start of iteration t_i are removed from V at least once prior to iteration t_{i+1}. Because all arc lengths are nonnegative, this guarantees that the minimum label node of V at the start of iteration t_i will never reenter V after iteration t_{i+1}. Thus the candidate list must have no more than $N - i$ nodes at the start of iteration t_{i+1}, and must become empty prior to iteration t_{N+1}. Thus, if the running time of the algorithm between iterations t_i and t_{i+1} is bounded by R, the total running time of the algorithm will be bounded by NR, and if R is polynomially bounded, the running time of the algorithm will also be polynomially bounded.

Specializing now to the SLF and LLL cases, assume that between iterations t_i and t_{i+1}, each node is inserted at the top of Q for a number of times that is bounded by a constant and that (in the case of SLF/LLL) the total number of repositionings is bounded by a constant multiple of A. Then it can be seen that the running time of the algorithm between iterations t_i and t_{i+1} is $O(A)$, and therefore the complexity of the algorithm is $O(NA)$.

To modify SLF or SLF/LLL so that they have an $O(NA)$ worst-case complexity, based on the preceding result, it is sufficient that we fix an integer $k > 1$, and that we separate the iterations of the algorithm in successive blocks of kN iterations each. We then impose an additional restriction that, within each block of kN iterations, each node can be inserted at most $k-1$ times at the top of Q [that is, after the $(k-1)$th insertion of a node to the top of Q within a given block of kN iterations, all subsequent insertions of that node within that block of kN iterations must be at the bottom of Q]. In the case of SLF/LLL, we also impose the additional restriction that the total number of repositionings within each block of kN iterations should be at most kA (that is, once the maximum number of kA repositionings is reached, the top node of Q is removed from Q regardless of the value of its label). The worst-case running times of the modified algorithms are then $O(NA)$. In practice, it is highly unlikely that the restrictions introduced into the algorithms to guarantee $O(NA)$ complexity will ever be exercised if k is larger than a small number such as 3 or 4.

2.4.4 The Threshold Algorithm

Similar to the SLF/LLL methods, the premise of this algorithm is also that, for nonnegative arc lengths, the number of iterations is reduced by removing from the candidate list V nodes with relatively small label. In the threshold algorithm, V is organized into two distinct queues Q' and Q'' using a *threshold* parameter s. The queue Q' contains nodes with "small" labels; that is, it contains only nodes whose labels are no larger than s. At each iteration, a node is removed from Q', and any node j to be added to the candidate list is inserted at the bottom of Q' or Q'' depending on whether $d_j \leq s$ or $d_j > s$, respectively. When the queue Q' is exhausted, the entire candidate list is repartitioned. The threshold is adjusted, and the queues Q' and Q'' are recalculated, so that Q' consists of the nodes with labels that are no larger than the new threshold.

To understand how the threshold algorithm works, consider the case of nonnegative arc lengths, and suppose that at time t the candidate list is repartitioned based on a new threshold value s, and that at some subsequent time $t' > t$ the queue Q' gets exhausted. Then at time t', all the nodes of the candidate list have label greater than s. In view of the nonnegativity of the arc lengths, this implies that all nodes with label less than or equal to s will not reenter the candidate list after time t'. In particular, all nodes that exited the candidate list between times t and t' become permanently labeled at time t' and never reenter the candidate list. We may thus interpret the threshold algorithm as a *block version of Dijkstra's method*, whereby a whole subset of nodes becomes permanently labeled when the queue Q' gets exhausted.

The preceding interpretation suggests that the threshold algorithm is suitable primarily for the case of nonnegative arc lengths (even though it will work in general). Furthermore, the performance of the algorithm is quite sensitive to the method used to adjust the threshold. For example, if s is taken to be equal to the current minimum label, the method is identical to Dijkstra's algorithm; if s is larger than all node labels, Q'' is empty and the algorithm reduces to the generic label correcting method. With an effective choice of threshold, the practical performance of the algorithm is very good. A number of heuristic approaches have been developed for selecting the threshold (see Glover, Klingman, and Phillips [1985], and Glover, Klingman, Phillips, and Schneider [1985]). If all arc lengths are nonnegative, a bound $O(NA)$ on the operation count of the algorithm can be shown; see Exercise 2.8(c).

Combinations of the Threshold and the SLF/LLL Methods

We mentioned earlier that the threshold algorithm may be interpreted as a block version of Dijkstra's method, whereby attention is restricted to the subset of nodes that belong to the queue Q', until this subset becomes permanently labeled. The algorithm used to permanently label the nodes of Q' is essentially the Bellman-Ford algorithm restricted to the subgraph defined by Q'. It is possible to use a different algorithm for this purpose, based for example on the SLF and LLL strategies. This motivates combinations of the threshold and the SLF/LLL algorithms.

In particular, the LLL strategy can be used when selecting a node to exit the queue Q' in the threshold algorithm (the top node of Q' is repositioned to the bottom of Q' if its label is found smaller than the average label in Q'). Furthermore, whenever a node enters the queue Q', it is added, according to the SLF strategy, at the bottom or the top of Q' depending on whether its label is greater than the label of the top node of Q' or not. The same policy is used when transferring to Q' the nodes of Q'' whose labels do not exceed the current threshold parameter. Thus the nodes of Q'' are transferred to Q' one-by-one, and they are added to the top or the bottom of Q' according to the SLF strategy. Finally, the SLF strategy is also followed when a node enters the queue Q''.

Generally, the threshold strategy and the SLF/LLL strategy are complementary and work synergistically. Computational experience suggests that their combination performs extremely well in practice, and typically results in an average number of iterations per node that is only slightly larger than the minimum of 1 achieved by Dijkstra's method. At the same time, these combined methods require considerably less overhead than Dijkstra's method.

2.4.5 Comparison of Label Setting and Label Correcting

Let us now try to compare the two major special cases of the generic algorithm, label setting and label correcting methods, assuming that the arc lengths are nonnegative.

We mentioned earlier that label setting methods offer a better guarantee of good performance than label correcting methods, because their worst-case running time is more favorable. In practice, however, there are several considerations that argue in favor of label correcting methods. One such consideration is that label correcting methods, because of their inherent flexibility, are better suited for exploiting advanced initialization.

Another consideration is that when the graph is acyclic, label correcting methods can be adapted to exploit the problem's structure, so that each node enters and exits the candidate list only once, thereby nullifying the major advantage of label setting methods (see Exercise 2.10). The corresponding running time is $O(A)$, which is the minimum possible. Note that an important class of problems involving an acyclic graph is dynamic programming (cf. Fig. 2.1).

A third consideration is that in practice, the graphs of shortest path problems are often sparse; that is, the number of arcs is much smaller than the maximum possible N^2. In this case, efficient label correcting methods tend to have a faster practical running time than label setting methods. To understand the reason, note that all shortest path methods require the unavoidable $O(A)$ operations needed to scan once every arc, plus some additional time which we can view as "overhead." The overhead of the popular label setting methods is roughly proportional to N in practice (perhaps times a slowly growing factor, like $\log N$), as argued earlier for the binary heap method and Dial's algorithm. On the other hand, the overhead of label correcting methods grows linearly with A (times a factor that likely grows slowly), because for the most popular methods, the average number of node entrances in the queue per node is typically not much larger than 1. Thus, we may conclude that the overhead ratio of label correcting to label setting methods is roughly

$$\frac{A}{N} \cdot (\text{a constant factor}).$$

The constant factor above depends on the particular method used and may vary slowly with the problem size, but is typically much less than 1. Thus, the overhead ratio favors label correcting methods for a sparse graph ($A << N^2$), and label setting methods for a dense graph ($A \approx N^2$). This is consistent with empirical observations.

Let us finally note that label setting methods can take better advantage of situations where only a small subset of the nodes are destinations, as will be seen in the next section. This is also true of the auction algorithms to be discussed in Section 2.6.

2.5 SINGLE ORIGIN/SINGLE DESTINATION METHODS

In this section, we discuss the adaptation of our earlier single origin/all destination algorithms to the case where there is only one destination, call it t, and we want to find the shortest distance from the origin node 1 to t. We could of course use our earlier all-destinations algorithms, but some improvements are possible.

2.5.1 Label Setting

Suppose that we use the label setting method. Then we can stop the method when the destination t becomes permanently labeled; further computation will not improve the label d_t (Exercise 2.13 sharpens this criterion in the case where $\min_{\{i|(i,t)\in\mathcal{A}\}} a_{ij} > 0$). If t is closer to the origin than many other nodes, the saving in computation time will be significant. Note that this approach can also be used when there are several destinations. The method is stopped when all destinations have become permanently labeled.

Another possibility is to use a *two-sided label setting method*; that is, a method that simultaneously proceeds from the origin to the destination *and* from the destination to the origin. In this method, we successively label permanently the closest nodes to the origin (with their shortest distance *from* the origin) and the closest nodes to the destination (with their shortest distance *to* the destination). It can be shown that when some node gets permanently labeled from both sides, the labeling can stop; by combining the forward and backward paths of each labeled node and by comparing the resulting origin-to-destination paths, one can obtain a shortest path. Exercise 2.14 develops in some detail this approach, which can often lead to a dramatic reduction in the total number of iterations. However, the approach does not work when there are multiple destinations.

2.5.2 Label Correcting

Unfortunately, when label correcting methods are used, it may not be easy to realize the savings just discussed in connection with label setting. The difficulty is that even after we discover several paths to the destination t (each marked by an entrance of t into V), we cannot be sure that better paths will not be discovered later. In the presence of additional problem structure, however, the number of times various nodes will enter V can be reduced considerably, as we now explain.

Suppose that at the start of the algorithm we have, for each node i, an *underestimate* u_i of the shortest distance from i to t (we require $u_t = 0$). For example, if all arc lengths are nonnegative we may take $u_i = 0$ for all i. (We do not exclude the possibility that $u_i = -\infty$ for some i, which corresponds to the case where no underestimate is available for the shortest distance of i.) The following is a modified version of the generic shortest path algorithm.

Initially

$$V = \{1\},$$

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\, i \neq 1.$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows.

Iteration of the Generic Single Origin/Single Destination Algorithm

Remove a node i from V. For each outgoing arc $(i, j) \in \mathcal{A}$, if

$$d_i + a_{ij} < \min\{d_j, d_t - u_j\},$$

set

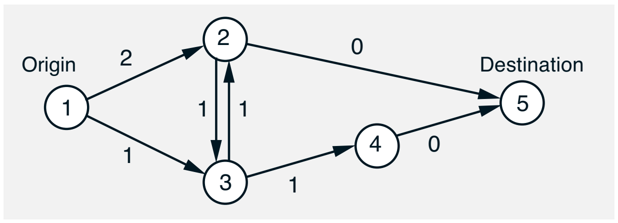
$$d_j := d_i + a_{ij}$$

and add j to V if it does not already belong to V.

The preceding iteration is the same as the one of the all-destinations generic algorithm, except that the test $d_i + a_{ij} < d_j$ for entering a node j into V is replaced by the more stringent test $d_i + a_{ij} < \min\{d_j, d_t - u_j\}$. (In fact, when the trivial underestimate $u_j = -\infty$ is used for all $j \neq t$ the two iterations coincide.) To understand the idea behind the iteration, note that the label d_j corresponds at all times to the best path found thus far from 1 to j (cf. Prop. 2.2). Intuitively, the purpose of entering node j in V when its label is reduced is to generate shorter paths to the destination that pass through node j. If P_j is the path from 1 to j corresponding to $d_i + a_{ij}$, then $d_i + a_{ij} + u_j$ is an underestimate of the shortest path length among the collection of paths $\mathcal{P}_j$ that first follow path P_j to node j and then follow some other path from j to t. However, if

$$d_i + a_{ij} + u_j \geq d_t,$$

the current best path to t, which corresponds to d_t, is at least as short as any of the paths in the collection $\mathcal{P}_j$, which have P_j as their first component.



Iter. #	Candidate List V	Node Labels	Node out of V
1	$\{1\}$	$(0,\infty,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,2,1,\infty,\infty)$	2
3	$\{3,5\}$	$(0,2,1,\infty,2)$	3
4	$\{5\}$	$(0,2,1,\infty,2)$	5
	$\varnothing$	$(0,2,1,\infty,2)$	

Figure 2.9: Illustration of the generic single origin/single destination algorithm. Here the destination is $t = 5$ and the underestimates of shortest distances to t are $u_i = 0$ for all i. Note that at iteration 3, when node 3 is removed from V, the label of node 4 is not improved to $d_4 = 2$ and node 4 is not entered in V. The reason is that $d_3 + a_{34}$ (which is equal to 2) is not smaller than $d_5 - u_4$ (which is also equal to 2). Note also that upon termination the label of a node other than t may not be equal to its shortest distance (e.g. d_4).

It is unnecessary to consider such paths, and for this reason node j need not be entered in V. In this way, the number of node entrances in V may be sharply reduced.

Figure 2.9 illustrates the algorithm. The following proposition proves its validity.

Proposition 2.4: Consider the generic single origin/single destination algorithm.

(a) At the end of each iteration, if $d_j < \infty$, then d_j is the length of some path that starts at 1 and ends at j.

(b) If the algorithm terminates, then upon termination, either $d_t < \infty$, in which case d_t is the shortest distance from 1 to t, or else there is no path from 1 to t.

(c) If the algorithm does not terminate, there exist paths of arbitrarily small length that start at 1.

Proof: (a) The proof is identical to the corresponding part of Prop. 2.2.

(b) If upon termination we have $d_t = \infty$, then the extra test $d_i + a_{ij} + u_j < d_t$ for entering V is always passed, so the algorithm generates the same label sequences as the generic (all destinations) shortest path algorithm. Therefore, Prop. 2.2(b) applies and shows that there is no path from 1 to t. It will thus be sufficient to prove this part assuming that we have $d_t < \infty$ upon termination.

Let $\overline{d}_j$ be the final values of the labels d_j obtained upon termination and suppose that $\overline{d}_t < \infty$. Assume, to arrive at a contradiction, that there is a path $P_t = (1, j_1, j_2, \ldots, j_k, t)$ that has length L_t with $L_t < \overline{d}_t$. For $m = 1, \ldots, k$, let L_{j_m} be the length of the path $P_m = (1, j_1, j_2, \ldots, j_m)$.

Let us focus on the node j_k preceding t on the path P_t. We claim that $L_{j_k} < \overline{d}_{j_k}$. Indeed, if this were not so, then j_k must have been removed at some iteration from V with a label d_{j_k} satisfying $d_{j_k} \leq L_{j_k}$. If d_t is the label of t at the start of that iteration, we would then have

$$d_{j_k} + a_{j_k t} \leq L_{j_k} + a_{j_k t} = L_t < \overline{d}_t \leq d_t,$$

implying that the label of t would be reduced at that iteration from d_t to $d_{j_k} + a_{j_k t}$, which is less than the final label $\overline{d}_t$ – a contradiction.

Next we focus on the node j_{k-1} preceding j_k and t on the path P_t. We use a similar (though not identical) argument to show that $L_{j_{k-1}} < \overline{d}_{j_{k-1}}$. Indeed, if this were not so, then j_{k-1} must have been removed at some iteration from V with a label $d_{j_{k-1}}$ satisfying $d_{j_{k-1}} \leq L_{j_{k-1}}$. If d_{j_k} and d_t are the labels of j_k and t at the start of that iteration, we would then have

$$d_{j_{k-1}} + a_{j_{k-1} j_k} \leq L_{j_{k-1}} + a_{j_{k-1} j_k} = L_{j_k} < \overline{d}_{j_k} \leq d_{j_k},$$

and since $L_{j_k} + u_{j_k} \leq L_t < \overline{d}_t \leq d_t$, we would also have

$$d_{j_{k-1}} + a_{j_{k-1} j_k} < d_t - u_{j_k}.$$

From the above two equations, it follows that the label of j_k would be reduced at that iteration from d_{j_k} to $d_{j_{k-1}} + a_{j_{k-1} t}$, which is less than the final label $\overline{d}_{j_k}$ – a contradiction.

Proceeding similarly, we obtain $L_{j_m} < \overline{d}_{j_m}$ for all $m = 1, \ldots, k$, and in particular $a_{1 j_1} = L_{j_1} < \overline{d}_{j_1}$. Since

$$a_{1 j_1} + u_{j_1} \leq L_t < \overline{d}_t,$$

and d_t is monotonically nonincreasing throughout the algorithm, we see that at the first iteration we will have $a_{1 j_1} < \min\{d_{j_1}, d_t - u_{j_1}\}$, so j_1 will enter V with the label $a_{1 j_1}$, which cannot be less than the final label $\overline{d}_{j_1}$. This is a contradiction; the proof of part (b) is complete.

(c) The proof is identical to the proof of Prop. 2.2(c). **Q.E.D.**

There are a number of possible implementations of the algorithm of this subsection, which parallel the ones given earlier for the many destinations problem. An interesting possibility to speed up the algorithm arises when an *overestimate* v_j of the shortest distance from j to t is known *a priori*. (We require that $v_t = 0$. Furthermore, we set $v_j = \infty$ if no overestimate is known for j.) The idea is that the method still works if the test $d_i + a_{ij} < d_t - u_j$ is replaced by the possibly sharper test $d_i + a_{ij} < D - u_j$, where D is any overestimate of the shortest distance from 1 to t with $D \leq d_t$ (check the proof of Prop. 2.4). We can obtain estimates D that may be strictly smaller than d_t by using the scalars v_j as follows: each time the label of a node j is reduced, we check whether $d_j + v_j < D$; if this is so, we replace D by $d_j + v_j$. In this way, we make the test for future admissibility into the candidate list V more stringent and save some unnecessary node entrances in V.

Advanced Initialization

We finally note that similar to the all-destinations case, the generic single origin/single destination method need not be started with the initial conditions

$$V = \{1\}, \qquad d_1 = 0, \qquad d_i = \infty, \quad \forall\, i \neq 1.$$

The algorithm works correctly using several other initial conditions. One possibility is to use for each node i, an initial label d_i that is either ∞ or else it is the length of a path from 1 to i, and to take $V = \{i \mid d_i < \infty\}$. A more sophisticated alternative is to initialize V so that it contains all nodes i such that

$$d_i + a_{ij} < \min\{d_j, d_t - u_j\} \text{ for some } (i,j) \in \mathcal{A}.$$

This kind of initialization can be extremely useful when a "good" path

$$P = (1, i_1, \ldots, i_k, t)$$

from 1 to t is known or can be found heuristically, and the arc lengths are nonnegative so that we can use the underestimate $u_i = 0$ for all i. Then we can initialize the algorithm with

$$d_i = \begin{cases} \text{Length of portion of path } P \text{ from 1 to } i & \text{if } i \in P, \\ \infty & \text{if } i \notin P, \end{cases}$$

$$V = \{1, i_1, \ldots, i_k\}.$$

If P is a near-optimal path and consequently the initial value d_t is near its final value, the test for future admissibility into the candidate list V will be relatively tight from the start of the algorithm and many unnecessary entrances of nodes into V may be saved. In particular, it can be seen that all nodes whose shortest distances from the origin are greater or equal to the length of P will never enter the candidate list.

2.6 AUCTION ALGORITHMS

In this section, we discuss another class of algorithms for finding a shortest path from an origin s to a destination t. These are called *auction* algorithms because they can be shown to be closely related to the naive auction algorithm for the assignment problem discussed in Section 1.3 (see Bertsekas [1991a], Section 4.3.3, or Bertsekas [1991b]). The main algorithm is very simple. It maintains a single path starting at the origin. At each iteration, the path is either *extended* by adding a new node, or *contracted* by deleting its terminal node. When the destination becomes the terminal node of the path, the algorithm terminates.

To get an intuitive sense of the algorithm, think of a mouse moving in a graph-like maze, trying to reach a destination. The mouse criss-crosses the maze, either advancing or backtracking along its current path. Each time the mouse backtracks from a node, it records a measure of the desirability of revisiting and advancing from that node in the future (this will be represented by a suitable variable). The mouse revisits and proceeds forward from a node when the node's measure of desirability is judged superior to those of other nodes. The algorithm emulates efficiently this search process using simple data structures.

The algorithm maintains a path $P = \big((s,i_1),(i_1,i_2),\ldots,(i_{k-1},i_k)\big)$ with no cycles, and modifies P using two operations, *extension* and *contraction*. If i_{k+1} is a node not on P and (i_k,i_{k+1}) is an arc, an *extension of P by i_{k+1}* replaces P by the path $\big((s,i_1),(i_1,i_2),\ldots,(i_{k-1},i_k),(i_k,i_{k+1})\big)$. If P does not consist of just the origin node s, a *contraction of P* replaces P by the path $\big((s,i_1),(i_1,i_2),\ldots,(i_{k-2},i_{k-1})\big)$.

We introduce a variable p_i for each node i, called the *price of node i*. We denote by p the price vector consisting of all node prices. The algorithm maintains a price vector p satisfying together with P the following property

$$p_i \le a_{ij} + p_j, \qquad \text{for all arcs } (i,j), \tag{2.14}$$

$$p_i = a_{ij} + p_j, \qquad \text{for all arcs } (i,j) \text{ of } P. \tag{2.15}$$

If we view the prices p_i as the negative of the labels d_i that we used earlier, we see that the above conditions are equivalent to the CS conditions (2.2) and (2.4). Consequently, we will also refer to Eqs. (2.14) and (2.15) as the CS conditions. We assume that the initial pair (P,p) satisfies CS. This is not restrictive, since the default pair

$$P = (s), \qquad p_i = 0, \quad \text{for all } i$$

satisfies CS in view of the nonnegative arc length assumption. To define the algorithm we also need to assume that *all cycles have positive length*; Exercise 2.17 indicates how this assumption can be relaxed.

It can be shown that if a pair (P, p) satisfies the CS conditions, then the portion of P between node s and any node $i \in P$ is a shortest path from s to i, while $p_s - p_i$ is the corresponding shortest distance. To see this, note that by Eq. (2.15), $p_i - p_k$ is the length of the portion of P between i and k, and that every path connecting i to k must have length at least equal to $p_i - p_k$ [add Eq. (2.14) along the arcs of the path].

The algorithm proceeds in iterations, transforming a pair (P, p) satisfying CS into another pair satisfying CS. At each iteration, the path P is either extended by a new node or is contracted by deleting its terminal node. In the latter case the price of the terminal node is increased strictly. A degenerate case occurs when the path consists of just the origin node s; in this case the path is either extended or is left unchanged with the price p_s being strictly increased. The iteration is as follows.

Iteration of the Auction Algorithm

Let i be the terminal node of P. If

$$p_i < \min_{\{j \mid (i,j) \in \mathcal{A}\}} \{a_{ij} + p_j\},$$

go to Step 1; else go to Step 2.

Step 1 (Contract path): Set

$$p_i := \min_{\{j \mid (i,j) \in \mathcal{A}\}} \{a_{ij} + p_j\},$$

and if $i \neq s$, contract P. Go to the next iteration.

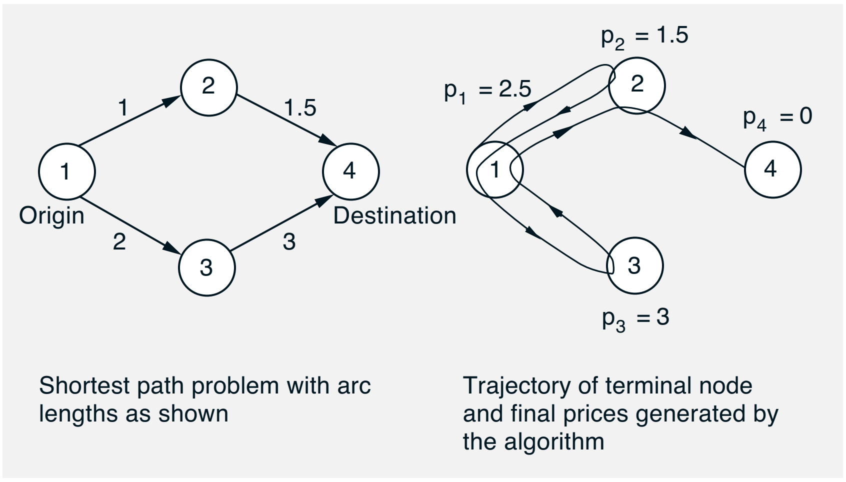
Step 2 (Extend path): Extend P by node j_i where

$$j_i = \arg \min_{\{j \mid (i,j) \in \mathcal{A}\}} \{a_{ij} + p_j\}$$

(ties are broken arbitrarily). If j_i is the destination t, stop; P is the desired shortest path. Otherwise, go to the next iteration.

It is easily seen that the algorithm maintains CS. Furthermore, the addition of the node j_i to P following an extension does not create a cycle, since otherwise, in view of the condition $p_i \leq a_{ij} + p_j$, for every arc (i, j) of the cycle we would have $p_i = a_{ij} + p_j$. By adding this equality along the cycle, we see that the length of the cycle must be zero, which is not possible by our assumptions.

Figure 2.10 illustrates the algorithm. It can be seen from the example of this figure that the terminal node traces the tree of shortest paths from the origin to the nodes that are closer to the origin than the given destination. This behavior is typical when the initial prices are all zero (see Exercise 2.19).


1
2
1.5
4
1
Origin
2
3
3
Destination
Shortest path problem with arc lengths as shown
p_2 = 1.5
p_1 = 2.5
p_4 = 0
p_3 = 3
Trajectory of terminal node and final prices generated by the algorithm

Iteration #	**Path P prior to iteration**	**Price vector p prior to iteration**	**Type of action during iteration**
1	(1)	$(0,0,0,0)$	contraction at 1
2	(1)	$(1,0,0,0)$	extension to 2
3	$(1,2)$	$(1,0,0,0)$	contraction at 2
4	(1)	$(1,1.5,0,0)$	contraction at 1
5	(1)	$(2,1.5,0,0)$	extension to 3
6	$(1,3)$	$(2,1.5,0,0)$	contraction at 3
7	(1)	$(2,1.5,3,0)$	contraction at 1
8	(1)	$(2.5,1.5,3,0)$	extension to 2
9	$(1,2)$	$(2.5,1.5,3,0)$	extension to 4
10	$(1,2,4)$	$(2.5,1.5,3,0)$	stop

Figure 2.10: An example illustrating the auction algorithm starting with $P = (1)$ and $p = 0$.

There is an interesting interpretation of the CS conditions in terms of a mechanical model, due to Minty [1957]. Think of each node as a ball, and for every arc (i,j), connect i and j with a string of length a_{ij}. (This requires that $a_{ij} = a_{ji} > 0$, which we assume for the sake of the interpretation.) Let the resulting balls-and-strings model be at an arbitrary position in three-dimensional space, and let p_i be the vertical coordinate of node i. Then the CS condition $p_i - p_j \leq a_{ij}$ clearly holds for all arcs (i,j), as illustrated in Fig. 2.11(b). If the model is picked up and left to hang from the origin node (by gravity – strings that are tight are perfectly vertical), then for all the tight strings (i,j) we have $p_i - p_j = a_{ij}$, so any tight chain of strings corresponds to a shortest path between the end nodes of the chain, as illustrated in Fig. 2.11(c). In particular, the length of the tight chain connecting the origin node s to any other node i is $p_s - p_i$ and is also equal to the shortest distance from s to i.

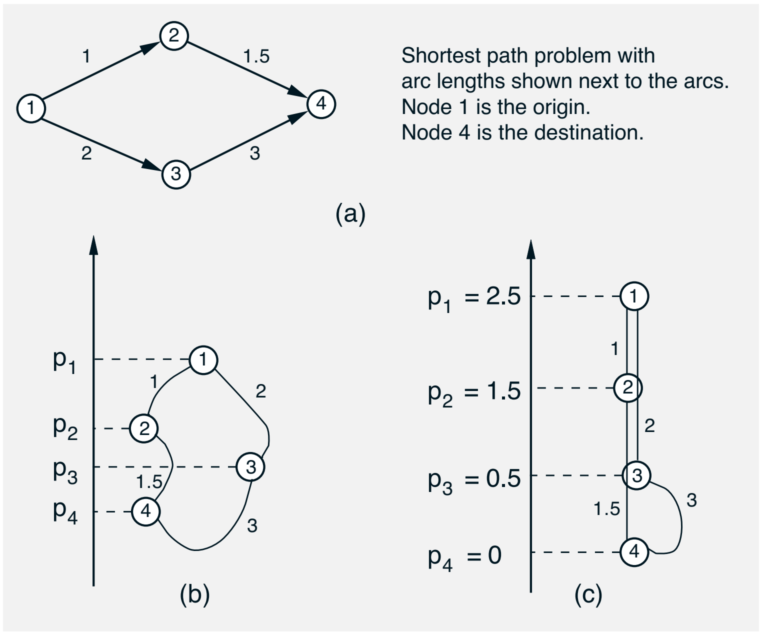

Shortest path problem with
arc lengths shown next to the arcs.
Node 1 is the origin.
Node 4 is the destination.
(a)
p1
p2
p3
p4
(b)
p1 = 2.5
p2 = 1.5
p3 = 0.5
p4 = 0
(c)

Figure 2.11: Illustration of the CS conditions for the shortest path problem. If each node is a ball, and for every arc (i,j), nodes i and j are connected with a string of length a_{ij}, the vertical coordinates p_i of the nodes satisfy $p_i - p_j \leq a_{ij}$, as shown in (b) for the problem given in (a). If the model is picked up and left to hang from the origin node s, then $p_s - p_i$ gives the shortest distance to each node i, as shown in (c).

The algorithm can also be interpreted in terms of the balls-and-strings model; it can be viewed as a process whereby nodes are raised in stages as illustrated in Fig. 2.12. Initially all nodes are resting on a flat surface. At each stage, we raise the *last* node in a tight chain that starts at the origin to the level at which at least one more string becomes tight.

The following proposition establishes the validity of the auction algorithm.

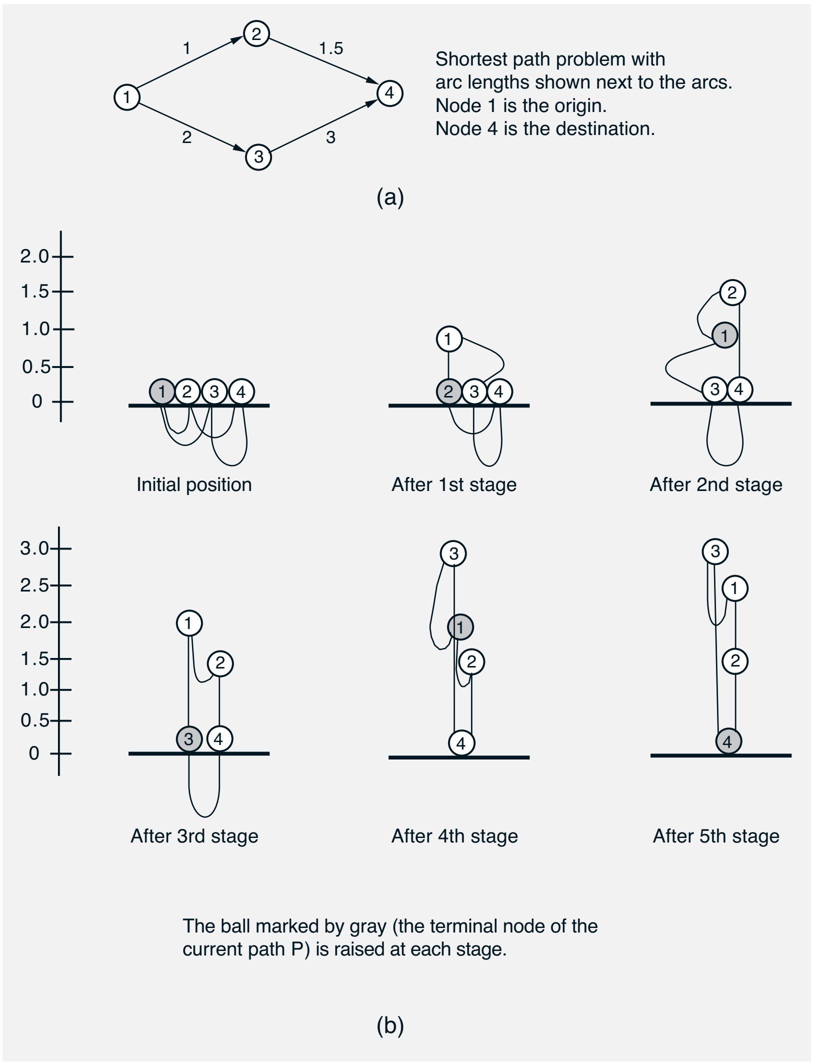

1
1.5
2
3
Shortest path problem with arc lengths shown next to the arcs. Node 1 is the origin. Node 4 is the destination.
(a)
Initial position
After 1st stage
After 2nd stage
After 3rd stage
After 4th stage
After 5th stage
The ball marked by gray (the terminal node of the current path P) is raised at each stage.
(b)

Figure 2.12: Illustration of the auction algorithm in terms of the balls-and-strings model for the problem shown in (a). The model initially rests on a flat surface, and various balls are then raised in stages. At each stage we raise a single ball $i \neq t$ (marked by gray), which is at a lower level than the origin s and can be reached from s through a sequence of tight strings; i should not have any tight string connecting it to another ball, which is at a lower level, that is, i should be the last ball in a tight chain hanging from s. (If s does not have any tight string connecting it to another ball, which is at a lower level, we use $i = s$.) We then raise i to the first level at which one of the strings connecting it to a ball at a lower level becomes tight. Each stage corresponds to a contraction plus all the extensions up to the next contraction. The ball i, which is being raised, corresponds to the terminal node of the current path P.

Proposition 2.5: If there exists at least one path from the origin to the destination, the auction algorithm terminates with a shortest path from the origin to the destination. Otherwise the algorithm never terminates and $p_s \to \infty$.

Proof: We first show by induction that (P, p) satisfies the CS conditions

$$p_i \le a_{ij} + p_j, \qquad \text{for all arcs } (i, j), \tag{2.16}$$

$$p_i = a_{ij} + p_j, \qquad \text{for all arcs } (i, j) \text{ of } P, \tag{2.17}$$

throughout the algorithm. Indeed, the initial pair satisfies CS by assumption. Consider an iteration that starts with a pair (P, p) satisfying CS and produces a pair $(\overline{P}, \overline{p})$. Let i be the terminal node of P. If

$$p_i = a_{ij_i} + p_{j_i} = \min_{\{j \mid (i,j) \in \mathcal{A}\}} \bigl\{a_{ij} + p_j\bigr\}, \tag{2.18}$$

then $\overline{P}$ is the extension of P by the node j_i and $\overline{p} = p$, implying that the CS condition (2.17) holds for all arcs of P as well as arc (i, j_i) [since j_i attains the minimum in Eq. (2.18)].

Suppose next that

$$p_i < \min_{\{j \mid (i,j) \in \mathcal{A}\}} \bigl\{a_{ij} + p_j\bigr\}.$$

Then if P is the degenerate path (s), the CS conditions hold vacuously. Otherwise, $\overline{P}$ is obtained by contracting P, and for all nodes $j \in \overline{P}$, we have $\overline{p}_j = p_j$, implying the CS conditions (2.16) and (2.17) for arcs outgoing from nodes of $\overline{P}$. Also, for the terminal node i, we have

$$\overline{p}_i = \min_{\{j \mid (i,j) \in \mathcal{A}\}} \bigl\{a_{ij} + p_j\bigr\},$$

implying the CS condition (2.16) for arcs outgoing from that node as well. Finally, since $\overline{p}_i > p_i$ and $\overline{p}_k = p_k$ for all $k \neq i$, we have $\overline{p}_k \le a_{kj} + \overline{p}_j$ for all arcs (k, j) outgoing from nodes $k \notin P$. This completes the induction proof that (P, p) satisfies CS throughout the algorithm.

Assume first that there is a path from node s to the destination t. By adding the CS condition (2.16) along that path, we see that $p_s - p_t$ is an underestimate of the (finite) shortest distance from s to t. Since p_s is monotonically nondecreasing, and p_t is fixed throughout the algorithm, it follows that p_s must stay bounded.

We next claim that p_i must stay bounded for all i. Indeed, in order to have $p_i \to \infty$, node i must become the terminal node of P infinitely often.

Each time this happens, $p_s - p_i$ is equal to the shortest distance from s to i, which is a contradiction since p_s is bounded.

We next show that the algorithm terminates. Indeed, it can be seen with a straightforward induction argument that for every node i, p_i is either equal to its initial value, or else it is the length of some path starting at i plus the initial price of the final node of the path; we call this the *modified length* of the path. Every path from s to i can be decomposed into a path with no cycles together with a finite number of cycles, each having positive length by assumption, so the number of distinct modified path lengths within any bounded interval is bounded. Now p_i was shown earlier to be bounded, and each time i becomes the terminal node by extension of the path P, p_i is strictly larger over the preceding time i became the terminal node of P, corresponding to a strictly larger modified path length. It follows that the number of times i can become a terminal node by extension of the path P is bounded. Since the number of path contractions between two consecutive path extensions is bounded by the number of nodes in the graph, the number of iterations of the algorithm is bounded, implying that the algorithm terminates.

Assume now that there is no path from node s to the destination. Then, the algorithm will never terminate, so by the preceding argument, some node i will become the terminal node by extension of the path P infinitely often and $p_i \to \infty$. At the end of iterations where this happens, $p_s - p_i$ must be equal to the shortest distance from s to i, implying that $p_s \to \infty$. **Q.E.D.**

Nonpolynomial Behavior and Graph Reduction

A drawback of the auction algorithm as described above is that its running time can depend on the arc lengths. A typical situation arises in graphs involving a cycle with relatively small length, as illustrated in Fig. 2.13. It is possible to turn the algorithm into one that is polynomial, by using some variations of the algorithm. In these variations, in addition to the extension and contraction operations, an additional *reduction* operation is introduced whereby some unnecessary arcs of the graph are deleted. We briefly describe the simplest of these variations, and we refer to Bertsekas, Pallottino, and Scutellà [1995] for other more sophisticated variations and complexity analysis.

This variant of the auction algorithm has the following added feature: each time that a node j becomes the terminal node of the path P through an extension using arc (i, j), all incoming arcs (k, j) of j with $k \neq i$ are deleted from the graph. Also, each time that a node j with no outgoing arcs becomes the terminal node of P, the path P is contracted and the node j is deleted from the graph. It can be seen that the arc deletion process leaves the shortest distance from s to t unaffected, and that the

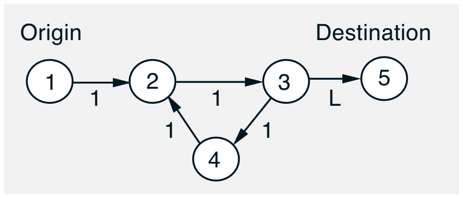


Figure 2.13: Example graph for which the number of iterations of the algorithm is not polynomially bounded. The lengths are shown next to the arcs and $L > 1$. By tracing the steps of the algorithm starting with $P = (1)$ and $p = 0$, we see that the price of node 3 will be first increased by 1 and then it will be increased by increments of 3 (the length of the cycle) as many times as is necessary for p_3 to reach or exceed L.

algorithm terminates either by finding a shortest path from s to t or by deleting s, depending on whether there exists at least one path from s to t or not. It can also be seen that this is also true even if there are cycles of zero length. Thus, in addition to addressing the nonpolynomial behavior, the graph reduction process deals effectively with the case where there are zero length cycles.

As an illustration, the reader may apply the algorithm with graph reduction to the example of Fig. 2.13. After the first iteration when node 2 becomes the terminal node of P for the first time, the arc $(4, 2)$ is deleted, and the cycle $(2, 3, 4, 2)$ that caused the nonpolynomial behavior is eliminated. Furthermore, once node 4 becomes the terminal node of P, it gets deleted because it no longer has any outgoing arcs. The number of iterations required is greatly reduced.

The effect of graph reduction may be enhanced by introducing a further idea due to Cerulli (see Cerulli, Festa, and Raiconi [1997a]). In particular, if in the process of eliminating arcs, a node i is left with only one outgoing arc (i, j), it may be "combined" with node j. This can be done efficiently, and may result in significant computational savings for some problem types (particularly those involving a sparse graph).

In addition to graph reduction, there are a number of ideas that can be used to implement efficiently the auction algorithm; see Bertsekas [1991b], Bertsekas, Pallottino, and Scutellà [1995], and Cerulli, Festa, and Raiconi [1997b].

The Case of Multiple Destinations or Multiple Origins

To solve the problem with multiple destinations and a single origin, one can simply run the algorithm until every destination becomes the terminal node of the path at least once. Also, to solve the problem with multiple origins and a single destination, one can combine several versions of the algorithm – one for each origin. However, the different versions can share a common price vector, since regardless of the origin considered, the condition $p_i \leq a_{ij} + p_j$ is always maintained. There are several ways to operate such a method; they differ in the policy used for switching between different origins. One possibility is to run the algorithm for one origin and, after the shortest path is obtained, to switch to the next origin (without changing the price vector), and so on, until all origins are exhausted. Another possibility, which is probably preferable in most cases, is to rotate between different origins, switching from one origin to another, if a contraction at the origin occurs or the destination becomes the terminal node of the current path.

The Reverse Algorithm

For problems with one origin and one destination, a two-sided version of the algorithm is particularly effective. This method maintains, in addition to the path P, another path R that *ends at the destination*. To understand this version, we first note that in shortest path problems, one can exchange the role of origins and destinations by reversing the direction of all arcs. It is therefore possible to use a destination-oriented version of the auction algorithm that maintains a path R that *ends* at the destination and changes at each iteration by means of a contraction or an extension. This algorithm, called the *reverse algorithm*, is mathematically equivalent to the earlier (forward) auction algorithm. Initially, in the reverse algorithm, R is any path ending at the destination, and p is any price vector satisfying CS together with R; for example,

$$R = (t), \qquad p_i = 0, \quad \text{for all } i,$$

if all arc lengths are nonnegative.

Iteration of the Reverse Algorithm

Let j be the starting node of R. If

$$p_j > \max_{\{i \mid (i,j) \in \mathcal{A}\}} \{p_i - a_{ij}\},$$

go to Step 1; else go to Step 2.

Step 1: (Contract path) Set

$$p_j := \max_{\{i \mid (i,j) \in \mathcal{A}\}} \{p_i - a_{ij}\},$$

and if $j \neq t$, contract R, (that is, delete the starting node j of R). Go to the next iteration.

Step 2: (Extend path) Extend R by node i_j, (that is, make i_j the starting node of R, preceding j), where

$$i_j = \arg \max_{\{i \mid (i,j) \in \mathcal{A}\}} \{p_i - a_{ij}\}$$

(ties are broken arbitrarily). If i_j is the origin s, stop; R is the desired shortest path. Otherwise, go to the next iteration.

The reverse algorithm is most helpful when it is combined with the forward algorithm. In a combined algorithm, initially we have a price vector p, and two paths P and R, satisfying CS together with p, where P starts at the origin and R ends at the destination. The paths P and R are extended and contracted according to the rules of the forward and the reverse algorithms, respectively, and the combined algorithm terminates when P and R have a common node. Both P and R satisfy CS together with p throughout the algorithm, so when P and R meet, say at node i, the composite path consisting of the portion of P from s to i followed by the portion of R from i to t will be shortest.

Combined Forward/Reverse Auction Algorithm

Step 1: (Run forward algorithm) Execute several iterations of the forward algorithm (subject to the termination condition), at least one of which leads to an increase of the origin price p_s. Go to Step 2.

Step 2: (Run reverse algorithm) Execute several iterations of the reverse algorithm (subject to the termination condition), at least one of which leads to a decrease of the destination price p_t. Go to Step 1.

The combined forward/reverse algorithm can also be interpreted in terms of the balls-and-strings model of Fig. 2.11. Again, all nodes are resting initially on a flat surface. When the forward part of the algorithm is used, we raise nodes in stages as illustrated in Fig. 2.12. When the reverse part of the algorithm is used, we *lower* nodes in stages; at each stage, we lower the *top* node in a tight chain that ends at the destination to the level at which at least one more string becomes tight.

The combined forward/reverse auction algorithm can be easily extended to handle single-origin/many-destination problems. One may start the reverse portion of the algorithm from any destination for which a shortest path has not yet been found. Based on experiments with randomly generated problems, the combined forward/reverse auction algorithm (with graph reduction to eliminate nonpolynomial behavior) outperforms substantially and often dramatically its closest competitors for single-origin/few-destination problems (see Bertsekas [1991b], and Bertsekas, Pallottino, and Scutellà [1995]). The intuitive reason for this is that through the mechanism of the reverse portion of the algorithm, the selected destinations are reached by the forward portion faster than other nodes, thereby leading to faster termination.

2.7 MULTIPLE ORIGIN/MULTIPLE DESTINATION METHODS

In this section, we consider the all-pairs shortest path problem, where we want to find a shortest path from each node to each other node. The *Floyd-Warshall algorithm* is specifically designed for this problem, and it is not any faster when applied to the single destination problem. It starts with the initial condition

$$D_{ij}^0 = \begin{cases} a_{ij} & \text{if } (i,j) \in \mathcal{A}, \\ \infty & \text{otherwise,} \end{cases}$$

and generates sequentially for all $k = 0, 1, \ldots, N-1$, and all nodes i and j,

$$D_{ij}^{k+1} = \begin{cases} \min\left\{ D_{ij}^k,\ D_{i(k+1)}^k + D_{(k+1)j}^k \right\} & \text{if } j \neq i, \\ \infty & \text{otherwise.} \end{cases}$$

An induction argument shows that D_{ij}^k gives the shortest distance from node i to node j using only nodes from 1 to k as intermediate nodes. Thus, D_{ij}^N gives the shortest distance from i to j (with no restriction on the intermediate nodes). There are N iterations, each requiring $O(N^2)$ operations, for a total of $O(N^3)$ operations.

Unfortunately, the Floyd-Warshall algorithm cannot take advantage of sparsity of the graph. It appears that for sparse problems it is typically better to apply a single origin/all destinations algorithm separately for each origin. If all the arc lengths are nonnegative, a label setting method can be used separately for each origin. If there are negative arc lengths (but no negative length cycles), one can of course apply a label correcting method separately for each origin, but there is another alternative that results in a superior worst-case complexity. It is possible to apply a label correcting method only *once* to a single origin/all destinations problem and obtain an equivalent all-pairs shortest path problem with nonnegative arc lengths; the latter problem can be solved using N separate applications of a label setting method. This alternative is based on the following proposition, which applies to the general minimum cost flow problem.

Proposition 2.7: Every minimum cost flow problem with arc costs a_{ij} such that all simple forward cycles have nonnegative cost is equivalent to another minimum cost flow problem involving the same graph and nonnegative arc costs $\hat{a}_{ij}$ of the form

$$\hat{a}_{ij} = a_{ij} + d_i - d_j, \qquad \forall\ (i,j) \in \mathcal{A},$$

where the scalars d_i can be found by solving a single origin/all destinations shortest path problem. The two problems are equivalent in the sense that they have the same constraints, and the cost function of one is the same as the cost function of the other plus a constant.

Proof: Let $(\mathcal{N}, \mathcal{A})$ be the graph of the given problem. Introduce a new node 0 and an arc $(0,i)$ for each $i \in \mathcal{N}$, thereby obtaining a new graph $(\mathcal{N}', \mathcal{A}')$. Consider the shortest path problem involving this graph, with arc lengths a_{ij} for the arcs $(i,j) \in \mathcal{A}$ and 0 for the arcs $(0,i)$. Since all incident arcs of node 0 are outgoing, all simple forward cycles of $(\mathcal{N}', \mathcal{A}')$ are also simple forward cycles of $(\mathcal{N}, \mathcal{A})$ and, by assumption, have nonnegative length. Since any forward cycle can be decomposed into a collection of simple forward cycles (cf. Exercise 1.4 in Chapter 1), all forward cycles (not necessarily simple) of $(\mathcal{N}', \mathcal{A}')$ have nonnegative length. Furthermore, there is at least one path from node 0 to every other node i, namely the path consisting of arc $(0,i)$. Therefore, the shortest distances d_i from node 0 to all other nodes i can be found by a label correcting method, and by Prop. 2.2, we have

$$\hat{a}_{ij} = a_{ij} + d_i - d_j \geq 0, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Let us now view $\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij} x_{ij}$ as the cost function of a minimum cost flow problem involving the graph $(\mathcal{N}, \mathcal{A})$ and the constraints of the original problem. We have

$$\begin{aligned}
\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij} x_{ij} &= \sum_{(i,j)\in\mathcal{A}} \big(a_{ij} + d_i - d_j\big) x_{ij} \\
&= \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} + \sum_{i\in\mathcal{N}} d_i \left(\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} \right) \\
&= \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} + \sum_{i\in\mathcal{N}} d_i s_i,
\end{aligned}$$

where s_i is the supply of node i. Thus, the two cost functions $\sum_{(i,j)\in\mathcal{A}} \hat{a}_{ij} x_{ij}$ and $\sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij}$ differ by the constant $\sum_{i\in\mathcal{N}} d_i s_i$. **Q.E.D.**

It can be seen now that the all-pairs shortest path problem can be solved by using a label correcting method to solve the single origin/all destinations problem described in the above proof, thereby obtaining the scalars d_i and

$$\hat{a}_{ij} = a_{ij} + d_i - d_j, \qquad \forall\ (i,j) \in \mathcal{A},$$

and by then applying a label setting method N times to solve the all-pairs shortest path problem involving the nonnegative arc lengths $\hat{a}_{ij}$. The shortest distance D_{ij} from i to j is obtained by subtracting $d_i - d_j$ from the shortest distance from i to j found by the label setting method. To estimate the running time of this approach, note that the label correcting method requires $O(NA)$ computation using the Bellman-Ford method, and each of the N applications of the label setting method require less than $O(N^2)$ computation (the exact count depends on the method used). Thus the overall running time is less that the $O(N^3)$ required by the Floyd-Warshall algorithm, at least for sparse graphs.

Still another possibility for solving the all-pairs shortest path problem is to solve N separate single origin/all destinations problems but to also use the results of the computation for one origin to start the computation for the next origin; see our earlier discussion of initialization of label correcting methods and also the discussion at the end of Section 5.2.

2.8 NOTES, SOURCES, AND EXERCISES

The work on the shortest path problem is very extensive, so we will restrict ourselves to citing the references that relate most to the material presented. Literature surveys are given by Dreyfus [1969], Deo and Pang [1984], and Gallo and Pallottino [1988]. The latter reference also contains codes for the most popular shortest path methods, and extensive computational comparisons. A survey of applications in transportation networks is given in Pallottino and Scutellà [1997a]. Parallel computation aspects of shortest path algorithms, including asynchronous versions of some of the algorithms developed here, are discussed in Bertsekas and Tsitsiklis [1989], and Kumar, Grama, Gupta, and Karypis [1994].

The generic algorithm was proposed as a unifying framework of many of the existing shortest path algorithms in Pallottino [1984], and Gallo and Pallottino [1986]. The first label setting method was suggested in Dijkstra [1959], and also independently in Dantzig [1960], and Whitting and Hillier [1960]. The binary heap method was proposed by Johnson [1972]. Dial's algorithm (Dial [1969]) received considerable attention after the appearance of the paper by Dial, Glover, Karney, and Klingman [1979]; see also Denardo and Fox [1979].

The Bellman-Ford algorithm was proposed in Bellman [1957] and Ford [1956] in the form given in Exercise 2.6, where the labels of all nodes are iterated simultaneously. The D'Esopo-Pape algorithm appeared in Pape [1974] based on an earlier suggestion of D'Esopo. The SLF and SLF/LLL methods were proposed by Bertsekas [1993a], and by Bertsekas, Guerriero, and Musmanno [1996]. Chen and Powell [1997] gave a simple polynomial version of the SLF method (Exercise 2.9). The threshold algorithm was developed by Glover, Klingman, and Phillips [1985], Glover, Klingman, Phillips, and Schneider [1985], and Glover, Glover, and Klingman [1986].

Two-sided label setting methods for the single origin/single destination problem (Exercise 2.14) were proposed by Nicholson [1966]; see also Helgason, Kennington, and Stewart [1993], which contains extensive computational results. The idea of using underestimates of the shortest distance to the destination in label correcting methods originated with the A^* algorithm, a shortest path algorithm that is popular in artificial intelligence (see Nilsson [1971], [1980], and Pearl [1984]).

The Floyd-Warshall algorithm was given in Floyd [1962] and uses a theorem due to Warshall [1962]. Alternative algorithms for the all-pairs problem are given in Dantzig [1967] and Tabourier [1973]. Reoptimization approaches that use the results of a shortest path computation for one origin to initialize the computation for other origins are given by Gallo and Pallottino [1982], and Florian, Nguyen, and Pallottino [1981].

The auction algorithm for shortest paths is due to Bertsekas [1991b]. The idea of graph reduction was proposed by Pallottino and Scutellà [1991], and an $O(N^3)$ implementation of an auction algorithm with graph reduction was given by Bertsekas, Pallottino, and Scutellà [1995]. An analysis of a parallel asynchronous implementation is given by Polymenakos and Bertsekas [1994]. Some variants of the auction algorithm that use slightly different price updating schemes have been proposed in Cerulli, De Leone, and Piacente [1992], and Bertsekas [1992b] (see Exercise 2.33). A method that combines the auction algorithm with some dual price iterations was given by Pallottino and Scutellà [1997b].

EXERCISES

2.1

Consider the graph of Fig. 2.14. Find a shortest path from 1 to all nodes using the binary heap method, Dial's algorithm, the D'Esopo-Pape algorithm, the SLF method, and the SLF/LLL method.

2.2

Suppose that the only arcs that have negative lengths are outgoing from the origin node 1. Show how to adapt Dijkstra's algorithm so that it solves the all-destinations shortest path problem in at most $N - 1$ iterations.

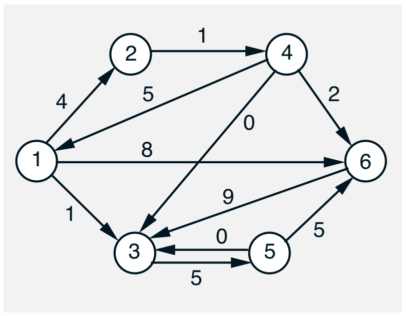


Figure 2.14: Graph for Exercise 2.1. The arc lengths are the numbers shown next to the arcs.

2.3

Give an example of a problem where the generic shortest path algorithm will reduce the label of node 1 to a negative value.

2.4 (Shortest Path Tree Construction)

Consider the single origin/all destinations shortest path problem and assume that all cycles have nonnegative length. Consider the generic algorithm of Section 2.2, and assume that each time a label d_j is decreased to $d_i + a_{ij}$ the arc (i, j) is stored in an array $PRED(j)$. Consider the subgraph of the arcs $PRED(j)$, $j \in \mathcal{N}$, $j \neq 1$. Show that at the end of each iteration this subgraph is a tree rooted at the origin, and that upon termination it is a tree of shortest paths.

2.5 (Uniqueness of Solution of Bellman's Equation)

Assume that all cycles have positive length. Show that if the scalars $d_1, d_2, \ldots, d_N$ satisfy

$$d_j = \min_{(i,j)\in\mathcal{A}} \{d_i + a_{ij}\}, \qquad \forall\, j \neq 1,$$

$$d_1 = 0,$$

then for all j, d_j is the shortest distance from 1 to j. Show by example that this need not be true if there is a cycle of length 0. *Hint*: Consider the arcs (i, j) attaining the minimum in the above equation and consider the paths formed by these arcs.

2.6 (The Original Bellman-Ford Method)

Consider the single origin/all destinations shortest path problem. The Bellman-Ford method, as originally proposed by Bellman and Ford, updates the labels of all nodes simultaneously in a single iteration. In particular, it starts with the initial conditions

$$d_1^0 = 0, \qquad d_j^0 = \infty, \qquad \forall\, j \neq 1,$$

and generates d_j^k, $k = 1, 2, \ldots$, according to

$$d_1^k = 0, \qquad d_j^k = \min_{(i,j)\in\mathcal{A}} \{d_i^{k-1} + a_{ij}\}, \qquad \forall\, j \neq 1.$$

(a) Show that for all $j \neq 1$ and $k \geq 1$, d_j^k is the shortest distance from 1 to j using paths with k arcs or less, where $d_j^k = \infty$ means that all the paths from 1 to j have more than k arcs.

(b) Assume that all cycles have nonnegative length. Show that the algorithm terminates after at most N iterations, in the sense that for some $k \leq N$ we have $d_j^k = d_j^{k-1}$ for all j. Conclude that the running time of the algorithm is $O(NA)$.

2.7 (The Bellman-Ford Method with Arbitrary Initialization)

Consider the single origin/all destinations shortest path problem and the following variant of the Bellman-Ford method of Exercise 2.6:

$$d_1^k = 0, \qquad d_j^k = \min_{(i,j)\in\mathcal{A}} \{d_i^{k-1} + a_{ij}\}, \qquad \forall\, j \neq 1,$$

where each of the initial iterates d_i^0 is an arbitrary scalar or ∞, except that $d_1^0 = 0$. We say that the algorithm *terminates after k iterations* if $d_i^k = d_i^{k-1}$ for all i.

(a) Given nodes $i \neq 1$ and $j \neq 1$, define

$$w_{ij}^k = \text{minimum path length over all paths starting at } i, \text{ ending at } j,$$
$$\text{and having } k \text{ arcs } (w_{ij}^k = \infty \text{ if there is no such path}).$$

For $i = 1$ and $j \neq 1$, define

$$w_{1j}^k = \text{minimum path length over all paths from 1 to } j \text{ having } k \text{ arcs or less}$$
$$(w_{1j}^k = \infty \text{ if there is no such path}).$$

Show by induction that

$$d_j^k = \min_{i=1,\ldots,N} \{d_j^0 + w_{ij}^k\}, \qquad \forall\, j = 2, \ldots, N, \text{ and } k \geq 1.$$

(b) Assume that there exists a path from 1 to every node i and that all cycles have positive length. Show that the method terminates at some iteration k, with d_i^k equal to the shortest distances d_i^*. *Hint*: For all $i \neq 1$ and $j \neq 1$, $\lim_{k\to\infty} w_{ij}^k = \infty$, while for all $j \neq 1$, $w_{1j}^k = d_j^*$ for all $k \geq N-1$.

(c) Under the assumptions of part (b), show that if $d_i^0 \geq d_i^*$ for all $i \neq 1$, the method terminates after at most $m^* + 1$ iterations, where

$$m^* = \max_{i\neq 1} m_i \leq N - 1,$$

and m_i is the smallest number of arcs contained in a shortest path from 1 to i.

(d) Under the assumptions of part (b), let

$$\beta = \max_{i \neq 1}\{d_i^* - d_i^0\},$$

and assume that $\beta > 0$. Show that the method terminates after at most $\overline{k} + 1$ iterations, where $\overline{k} = N - 1$ if the graph is acyclic, and $\overline{k} = N - 2 - \lceil \beta / L \rceil$ if the graph has cycles, where

$$L = \min_{\text{All simple cycles}} \frac{\text{Length of the cycle}}{\text{Number of arcs on the cycle}},$$

is the, so called, *minimum cycle mean* of the graph. *Note*: See Section 4.1 of Bertsekas and Tsitsiklis [1989] for related analysis, and an example showing that the given upper bound on the number of iterations for termination is tight.

(e) (Finding the minimum cycle mean) Consider the following Bellman-Ford-like algorithm:

$$d^k(i) = \min_{(i,j) \in \mathcal{A}}\{a_{ij} + d^{k-1}(j)\}, \qquad \forall\ i = 1, \ldots, N,$$

$$d^0(i) = 0, \qquad \forall\ i = 1, \ldots, N.$$

We assume that there exists at least one cycle, but we do not assume that all cycles have positive length. Show that the minimum cycle mean L of part (d) is given by

$$L = \min_{i=1,\ldots,N} \max_{k=0,\ldots,N-1} \frac{d^N(i) - d^k(i)}{N - k}.$$

Hint: Show that $d^k(i)$ is equal to the minimum path length over all paths that start at i and have k arcs.

2.8 (Complexity of the Generic Algorithm)

Consider the generic algorithm, assuming that all arc lengths are nonnegative.

(a) Consider a node j satisfying at some time

$$d_j \leq d_i, \qquad \forall\ i \in V.$$

Show that this relation will be satisfied at all subsequent times and that j will never again enter V. Furthermore, d_j will remain unchanged.

(b) Suppose that the algorithm is structured so that it removes from V a node of minimum label at least once every k iterations (k is some integer). Show that the algorithm will terminate in at most kN iterations.

(c) Show that the running time of the threshold algorithm is $O(NA)$. *Hint*: Define a cycle to be a sequence of iterations between successive repartitionings of the candidate list V. In each cycle, the node of V with minimum label at the start of the cycle will be removed from V during the cycle.

2.9 (Complexity of the SLF Method)

The purpose of this exercise, due to Chen and Powell [1997], is to show one way to use the SLF method so that it has polynomial complexity. Suppose that the outgoing arcs of each node have been presorted in increasing order by length. The effect of this, in the context of the generic shortest path algorithm, is that when a node i is removed from the candidate list, we first examine the outgoing arc from i that has minimum length, then we examine the arc of second minimum length, etc. Show an $O(NA^2)$ complexity bound for the method.

2.10 (Label Correcting for Acyclic Graphs)

Consider the problem of finding shortest paths from the origin node 1 to all destinations, and assume that the graph does not contain any forward cycles. Let T_k be the set of nodes i such that every path from 1 to i has k arcs or more, and there exists a path from 1 to i with exactly k arcs. For each i, if $i \in T_k$ define $INDEX(i) = k$. Consider a label setting method that selects a node i from the candidate list that has minimum $INDEX(i)$.

(a) Show that the method terminates and that each node visits the candidate list at most once.

(b) Show that the sets T_k can be constructed in $O(A)$ time, and that the running time of the algorithm is also $O(A)$.

2.11

Consider the graph of Fig. 2.14. Find a shortest path from node 1 to node 6 using the generic single origin/single destination method of Section 2.5 with all distance underestimates equal to zero.

2.12

Consider the problem of finding a shortest path from the origin 1 to a single destination t, subject to the constraint that the path includes a given node s. Show how to solve this problem using the single origin/single destination algorithms of Section 2.5.

2.13 (Label Setting for Few Destinations)

Consider a label setting approach for finding shortest paths from the origin node 1 to a selected subset of destinations T. Let

$$\overline{a} = \min_{\{(i,t)\in\mathcal{A} \mid t\in T\}} a_{it},$$

and assume that $\overline{a} > 0$. Show that one may stop the method when the node of minimum label in V has a label d_{min} that satisfies

$$d_{min} + \overline{a} \geq \max_{t\in T} d_t.$$

2.14 (Two-Sided Label Setting)

Consider the shortest path problem from an origin node 1 to a destination node t, and assume that all arc lengths are nonnegative. This exercise considers an algorithm where label setting is applied simultaneously and independently from the origin and from the destination. In particular, the algorithm maintains a subset of nodes W, which are permanently labeled from the origin, and a subset of nodes V, which are permanently labeled from the destination. When W and V have a node i in common the algorithm terminates. The idea is that a shortest path from 1 to t cannot contain a node $j \notin W \cup V$; any such path must be longer than a shortest path from 1 to i followed by a shortest path from i to t (unless j and i are equally close to both 1 and to t).

Consider two subsets of nodes W and V with the following properties:

(1) $1 \in W$ and $t \in V$.

(2) W and V have nonempty intersection.

(3) If $i \in W$ and $j \notin W$, then the shortest distance from 1 to i is less than or equal to the shortest distance from 1 to j.

(4) If $i \in V$ and $j \notin V$, then the shortest distance from i to t is less than or equal to the shortest distance from j to t.

Let d_i^1 be the shortest distance from 1 to i using paths all the nodes of which, with the possible exception of i, lie in W ($d_i^1 = \infty$ if no such path exists), and let d_i^t be the shortest distance from i to t using paths all the nodes of which, with the possible exception of i, lie in V ($d_i^t = \infty$ if no such path exists).

(a) Show that such W, V, d_i^1, and d_i^t can be found by applying a label setting method simultaneously for the single origin problem with origin node 1 and for the single destination problem with destination node t.

(b) Show that the shortest distance D_{1t} from 1 to t is given by

$$D_{1t} = \min_{i\in W} \left\{ d_i^1 + d_i^t \right\} = \min_{i\in W\cup V} \left\{ d_i^1 + d_i^t \right\} = \min_{i\in V} \left\{ d_i^1 + d_i^t \right\}.$$

(c) Show that the nonempty intersection condition (2) can be replaced by the condition $\min_{i\in W} \left\{ d_i^1 + d_i^t \right\} \le \max_{i\in W} d_i^1 + \max_{i\in V} d_i^t$.

2.15

Apply the forward/reverse auction algorithm to the example of Fig. 2.13, and show that it terminates in a number of iterations that does not depend on the large arc length L. Construct a related example for which the number of iterations of the forward/reverse algorithm is not polynomially bounded.

2.16 (Finding an Initial Price Vector)

In order to initialize the auction algorithm, one needs a price vector p satisfying the condition

$$p_i \le a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A}. \tag{2.19}$$

Such a vector may not be available if some arc lengths are negative. Furthermore, even if all arc lengths are nonnegative, there are many cases where it is important to use a favorable initial price vector in place of the default choice $p = 0$. This possibility arises in a reoptimization context with slightly different arc length data, or with a different origin and/or destination. This exercise gives an algorithm to obtain a vector p satisfying the condition (2.19), starting from another vector $\overline{p}$ satisfying the same condition for a different set of arc lengths $\overline{a}_{ij}$.

Suppose that we have a vector $\bar{p}$ and a set of arc lengths $\{\bar{a}_{ij}\}$, satisfying $\overline{p}_i \leq \overline{a}_{ij} + \overline{p}_j$ for all arcs (i, j), and we are given a new set of arc lengths $\{a_{ij}\}$. (For the case where some arc lengths a_{ij} are negative, this situation arises with $\overline{p} = 0$ and $\overline{a}_{ij} = \max\{0, a_{ij}\}$.) Consider the following algorithm that maintains a subset of arcs $\mathcal{E}$ and a price vector p, and terminates when $\mathcal{E}$ is empty. Initially

$$\mathcal{E} = \{(i,j) \in \mathcal{A} \mid a_{ij} < \bar{a}_{ij},\ i \neq t\}, \qquad p = \bar{p}.$$

The typical iteration is as follows:

Step 1 (Select arc to scan): If $\mathcal{E}$ is empty, stop; otherwise, remove an arc (i, j) from $\mathcal{E}$ and go to Step 2.

Step 2 (Add affected arcs to $\mathcal{E}$): If $p_i > a_{ij} + p_j$, set

$$p_i := a_{ij} + p_j$$

and add to $\mathcal{E}$ every arc (k, i) with $k \neq t$ that does not already belong to $\mathcal{E}$.

Assuming that each node i is connected to the destination t with at least one path, and that all cycle lengths are positive, show that the algorithm terminates with a price vector p satisfying

$$p_i \leq a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \neq t.$$

2.17 (Extension for the Case of Zero Length Cycles)

Extend the auction algorithm for the case where all arcs have nonnegative length but some cycles may consist exclusively of zero length arcs. *Hint*: Any cycle of zero length arcs generated by the algorithm can be treated as a single node. An alternative is the idea of graph reduction discussed in Section 2.6.

2.18

Consider the two single origin/single destination shortest path problems shown in Fig. 2.15.

(a) Show that the number of iterations required by the forward auction algorithm is estimated accurately by

$$n_t - 1 + \sum_{i \in \mathcal{I},\, i \neq t} (2n_i - 1),$$

where n_i is the number of nodes in a shortest path from 1 to i. Show also that the corresponding running times are $O(N^2)$.

(b) Show that for the problem of Fig. 2.15(a) the running time of the forward/reverse auction algorithm (with a suitable "reasonable" rule for switching between the forward and reverse algorithms) is $O(N^2)$ (the number of iterations is roughly half the corresponding number for the forward algorithm). Show also that for the problem of Fig. 2.15(b) the running time of the forward/reverse algorithm is $O(N)$.

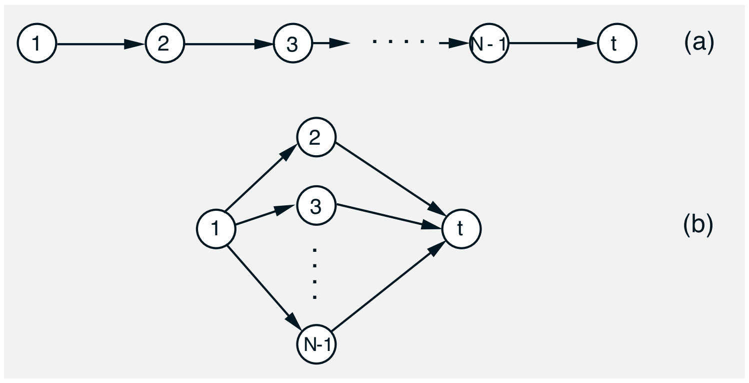


Figure 2.15: Shortest path problems for Exercise 2.18. In problem (a) the arc lengths are equal to 1. In problem (b), the length of each arc $(1, i)$ is i, and the length of each arc (i, t) is N.

2.19

In the auction algorithm of Section 2.6, let k_i be the first iteration at which node i becomes the terminal node of the path P. Show that if $k_i < k_j$, then the shortest distance from 1 to i is less or equal to the shortest distance from 1 to j.

2.20 (A Forward/Reverse Version of Dijkstra's Algorithm)

Consider the single origin/single destination shortest path problem and assume that all arc lengths are nonnegative. Let node 1 be the origin, let node t be the destination, and assume that there exists at least one path from 1 to t. This exercise provides a forward/reverse version of Dijkstra's algorithm, which is motivated by the balls-and-strings model analogy of Figs. 2.11 and 2.12. In particular, the algorithm may be interpreted as alternately lifting the model upward from the origin (the following Step 1), and pulling the model downward from the destination (the following Step 2). The algorithm maintains a price vector p and two node subsets W_1 and W_t. Initially, p satisfies the CS condition

$$p_i \leq a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{2.20}$$

$W_1 = \{1\}$, and $W_t = \{t\}$. One may view W_1 and W_t as the sets of permanently labeled nodes from the origin and from the destination, respectively. The algorithm terminates when W_1 and W_t have a node in common. The typical iteration is as follows:

Step 1 (Forward Step): Find

$$\gamma^+ = \min\{a_{ij} + p_j - p_i \mid (i,j) \in \mathcal{A},\ i \in W_1,\ j \notin W_1\}$$

and let

$$V_1 = \{j \notin W_1 \mid \gamma^+ = a_{ij} + p_j - p_i \text{ for some } i \in W_1\}.$$

Set

$$p_i := \begin{cases} p_i + \gamma^+ & \text{if } i \in W_1, \\ p_i & \text{if } i \notin W_1. \end{cases}$$

Set

$$W_1 := W_1 \cup V_1.$$

If W_1 and W_t have a node in common, terminate the algorithm; otherwise, go to Step 2.

Step 2 (Backward Step): Find

$$\gamma^- = \min\{a_{ji} + p_i - p_j \mid (j,i) \in \mathcal{A},\ i \in W_t,\ j \notin W_t\}$$

and let

$$V_t = \{j \notin W_t \mid \gamma^- = a_{ji} + p_i - p_j \text{ for some } i \in W_t\}.$$

Set

$$p_i := \begin{cases} p_i - \gamma^- & \text{if } i \in W_t, \\ p_i & \text{if } i \notin W_t. \end{cases}$$

Set

$$W_t := W_t \cup V_t.$$

If W_1 and W_t have a node in common, terminate the algorithm; otherwise, go to Step 1.

(a) Show that throughout the algorithm, the condition (2.20) is maintained. Furthermore, for all $i \in W_1$, $p_1 - p_i$ is equal to the shortest distance from 1 to i. Similarly, for all $i \in W_t$, $p_i - p_t$ is equal to the shortest distance from i to t. *Hint*: Show that if $i \in W_1$, there exists a path from 1 to i such that $p_m = a_{mn} + p_n$ for all arcs (m,n) of the path.

(b) Show that the algorithm terminates and that upon termination, $p_1 - p_t$ is equal to the shortest distance from 1 to t.

(c) Show how the algorithm can be implemented so that its running time is $O(N^2)$. *Hint*: Let d_{mn} denote the shortest distance from m to n. Maintain the labels

$$v_j^+ = \min\{d_{1i} + a_{ij} \mid i \in W_1,\ (i,j) \in \mathcal{A}\}, \qquad \forall\ j \notin W_1,$$

$$v_j^- = \min\{a_{ji} + d_{it} \mid i \in W_t,\ (j,i) \in \mathcal{A}\}, \qquad \forall\ j \notin W_t.$$

Let p_j^0 be the initial price of node j. Show that

$$\gamma^+ = \min\left\{ \min_{j \notin W_1,\, j \notin W_t} \bigl(v_j^+ + p_j^0\bigr),\ p_t + \min_{j \notin W_1,\, j \in W_t} \bigl(v_j^+ + d_{jt}\bigr) \right\} - p_1, \quad (2.21)$$

$$\gamma^- = \min\left\{ \min_{j \notin W_1,\, j \notin W_t} \bigl(v_j^- - p_j^0\bigr),\ -p_1 + \min_{j \in W_1,\, j \notin W_t} \bigl(v_j^- + d_{1j}\bigr) \right\} + p_t. \quad (2.22)$$

Use these relations to calculate γ^+ and γ^- in $O(N)$ time.

(d) Show how the algorithm can be implemented using binary heaps so that its running time is $O(A \log N)$. *Hint*: One possibility is to use four heaps to implement the minimizations in Eqs. (2.21) and (2.22).

(e) Apply the two-sided version of Dijkstra's algorithm with arc lengths $a_{ij} + p_j - p_i$ of Exercise 2.14, and with the termination criterion of part (c) of that exercise. Show that the resulting algorithm is equivalent to the one of the present exercise.

2.21

Consider the all-pairs shortest path problem, and suppose that the minimum distances d_{ij} to go from any i to any j have been found. Suppose that a *single* arc length a_{mn} is reduced to a value $\overline{a}_{mn} < a_{mn}$. Show that if $d_{nm} + \overline{a}_{mn} \geq 0$, the new shortest distances can be obtained by

$$\overline{d}_{ij} = \min\{d_{ij},\ d_{im} + \overline{a}_{mn} + d_{nj}\}.$$

What happens if $d_{nm} + \overline{a}_{mn} < 0$?

2.22 (The Doubling Algorithm)

The *doubling algorithm* for solving the all-pairs shortest path problem is given by

$$D_{ij}^1 = \begin{cases} a_{ij} & \text{if } (i,j) \in \mathcal{A}, \\ 0 & \text{if } i = j, \\ \infty & \text{otherwise,} \end{cases}$$

$$D_{ij}^{2k} = \begin{cases} \min_m \bigl\{D_{im}^k + D_{mj}^k\bigr\} & \text{if } i \neq j,\ k = 1, 2, \ldots, \lfloor \log(N-1) \rfloor, \\ 0 & \text{if } i = j,\ k = 1, 2, \ldots, \lfloor \log(N-1) \rfloor. \end{cases}$$

Show that for $i \neq j$, D_{ij}^k gives the shortest distance from i to j using paths with 2^{k-1} arcs or fewer. Show also that the running time is $O\bigl(N^3 \log m^*\bigr)$, where m^* is the maximum number of arcs in a shortest path.

2.23 (Dynamic Programming)

Consider the dynamic programming problem of Example 2.2. The standard dynamic programming algorithm is given by the recursion

$$J_k(x_k) = \min_{u_k} \big\{ g_k(x_k, u_k) + J_{k+1}(x_{k+1}) \big\}, \qquad k = 0, \ldots, N-1,$$

starting with

$$J_N(x_N) = G(x_N).$$

(a) In terms of the shortest path reformulation in Fig. 2.1, interpret $J_k(x_k)$ as the shortest distance from node x_k at stage k to the terminal node t.

(b) Show that the dynamic programming algorithm can be viewed as a special case of the generic label correcting algorithm with a special order for selecting nodes to exit the candidate list.

(c) Assume that $g_k(x_k, u_k) \geq 0$ for all x_k, u_k, and k. Suppose that by using some heuristic we can construct a "good" suboptimal control sequence $(u_0, u_1, \ldots, u_{N-1})$. Discuss how to use this sequence for initialization of a single origin/single destination label correcting algorithm (cf. the discussion of Section 2.5).

2.24 (Forward Dynamic Programming)

Given a problem of finding a shortest path from node s to node t, we can obtain an equivalent "reverse" shortest path problem, where we want to find a shortest path from t to s in a graph derived from the original by reversing the direction of all the arcs, while keeping their length unchanged. Apply this transformation to the dynamic programming problem of Example 2.2 and Exercise 2.23, and derive a dynamic programming algorithm that proceeds forwards rather than backwards in time.

2.25 (k Shortest Node-Disjoint Paths)

The purpose of this exercise, due to Castañon [1990], is to formulate a class of multiple shortest path problems and to indicate the method for their solution. Consider a graph with an origin 1, a destination t, and a length for each arc. We want to find k paths from 1 to t which share no node other 1 and t and which are such that the sum of the k path lengths is minimum. Formulate this problem as a minimum cost flow problem. (For an auction algorithm that solves this problem, see Bertsekas and Castañon [1993c].) *Hint*: Replace each node i other than 1 and t with two nodes i and i' and a connecting arc (i, i') with flow bounds $0 \leq x_{ii'} \leq 1$.

2.26 (k-Level Shortest Path Problems)

The purpose of this exercise, due to Shier [1979], and Guerriero, Lacagnina, Musmanno, and Pecorella [1997], is to introduce an approach for extending the generic algorithm to the solution of a class of multiple shortest path problems. Consider the single origin/many destinations shortest path context, where node 1 is the origin, assuming that no cycles of negative length exist. Let $d_i(1)$ denote the shortest distance from node 1 to node i. Sequentially, for $k = 2, 3, \ldots$, denote by $d_i(k)$ the minimum of the lengths of paths from 1 to i that have length greater than $d_i(k-1)$ [if there is no path from 1 to i with length greater than $d_i(k-1)$, then $d_i(k) = \infty$]. We call $d_i(k)$ the *k-level shortest distance* from 1 to i.

(a) Show that for $k > 1$, $\{d_i(k) \mid i = 1, \ldots, N\}$ are the k-level shortest distances if and only if $d_i(k-1) \le d_i(k)$ with strict inequality if $d_i(k-1) < \infty$, and furthermore

$$d_i(k) = \min_{(i,j)\in\mathcal{A}} \big\{ l_i(k,j) + a_{ij} \big\}, \qquad i = 1, \ldots, N,$$

where

$$l_i(k,j) = \begin{cases} d_i(k-1) & \text{if } d_j(k-1) < d_i(k-1) + a_{ij}, \\ d_i(k) & \text{if } d_j(k-1) = d_i(k-1) + a_{ij}. \end{cases}$$

(b) Extend the generic shortest path algorithm of Section 2.2 so that it simultaneously finds the k-level shortest distances for all $k = 1, 2, \ldots, K$, where K is some positive integer.

2.27 (Clustering)

We have a set of N objects $1, \ldots, N$ arranged in a given order. We want to group these objects in clusters that contain consecutive objects. For each subset $i, i+1, \ldots, i+k$, there is an associated cost $c(i,k)$. We want to find the grouping that minimizes the sum of the clusters' cost. Use the ideas of the paragraphing problem (Example 2.4) to formulate this problem as a shortest path problem.

2.28 (Path Bottleneck Problem)

Consider the framework of the shortest path problem. For any path P, define the *bottleneck arc* of P as an arc that has maximum length over all arcs of P. Consider the problem of finding a path connecting two given nodes and having minimum length of bottleneck arc. Derive an analog of Prop. 2.1 for this problem. Consider also a single origin/all destinations version of this problem. Develop an analog of the generic algorithm of Section 2.2 and prove an analog of Prop. 2.2. *Hint*: Replace $d_i + a_{ij}$ with $\max\{d_i, a_{ij}\}$.

2.29 (Shortest Path Problems with Negative Cycles)

Consider the problem of finding a simple forward path between an origin and a destination node that has minimum length. Show that even if there are negative cycles, the problem can be formulated as a minimum cost flow problem involving node throughput constraints of the form

$$0 \leq \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} \leq 1, \qquad \forall\ i.$$

2.30 (Minimum Weight Spanning Trees)

Given a graph $(\mathcal{N}, \mathcal{A})$ and a weight w_{ij} for each arc (i,j), consider the problem of finding a spanning tree with minimum sum of arc weights. This is not a shortest path problem and in fact it is not even a special case of the minimum cost flow problem. However, it has a similar graph structure to the one of the shortest path problem. Note that the orientation of the arcs does not matter here. In particular, if (i,j) and (j,i) are arcs, any one of them can participate in a spanning tree solution, and the arc having greater weight can be a priori eliminated.

(a) Consider the problem of finding a shortest path from node 1 to all nodes with arc lengths equal to w_{ij}. Give an example where the shortest path spanning tree is not a minimum weight spanning tree.

(b) Let us define a *fragment* to be a subgraph of a minimum weight spanning tree; for example the subgraph consisting of any subset of nodes and no arcs is a fragment. Given a fragment F, let us denote by $A(F)$ the set of arcs (i,j) such that either i or j belong to F, and if (i,j) is added to F no cycle is closed. Show that if F is a fragment, then by adding to F an arc of $A(F)$ that has minimum weight over all arcs of $A(F)$ we obtain a fragment.

(c) Consider a *greedy algorithm* that starts with some fragment, and at each iteration, adds to the current fragment F an arc of $A(F)$ that has minimum weight over all arcs of $A(F)$. Show that the algorithm terminates with a minimum weight spanning tree.

(d) Show that the complexity of the greedy algorithm is $O(NA)$, where N is the number of nodes and A is the number of arcs.

(e) The Prim-Dijkstra algorithm is the special case of the greedy algorithm where the initial fragment consists of a single node. Provide an $O(N^2)$, implementation of this algorithm. *Hint*: Together with the kth fragment F_k, maintain for each $j \notin F_k$ the node $n_k(i) \in F_k$ such that the arc connecting j and $n_k(i)$ has minimum weight.

2.31 (Shortest Path Problems with Losses)

Consider a vehicle routing/shortest path-like problem where a vehicle wants to go on a forward path from an origin node 1 to a destination node t in a graph that has no forward cycles. For each arc (i,j) there is a given length a_{ij}, but there is also a given probability $p_{ij} \in [0,1]$ that the vehicle will be destroyed in crossing the arc. The length of a path is now a random variable, and is equal to the sum of the arc lengths on the path up to the time the vehicle reaches its destination or gets destroyed, whichever comes first. We want to find a forward path $P = (1, i_1, \ldots, i_k, t)$ whose expected length, given by

$$\overline{p}_{1i_1}\big(a_{1i_1} + \overline{p}_{i_1 i_2}\big(a_{i_1 i_2} + \overline{p}_{i_2 i_3}(\cdots + \overline{p}_{i_k t} a_{i_k t})\cdots\big),$$

is minimized, where $\overline{p}_{ij} = 1 - p_{ij}$ is the probability of survival in crossing the arc (i,j). Give an algorithm of the dynamic programming type for solving this problem (cf. Exercise 2.5). Does the problem always make sense when the graph has some forward cycles?

2.32

Consider the one origin-all destinations problem and the generic algorithm of Section 2.2. Assume that there exists a path that starts at node 1 and contains a cycle with negative length. Assume also that the generic algorithm is operated so that if a given node belongs to the candidate list for an infinite number of iterations, then it also exits the list an infinite number of times. Show that there exists at least one node j such that the sequence of labels d_j generated by the algorithm diverge to $-\infty$. *Hint*: Argue that if the limits $\overline{d}_j$ of all the label nodes are finite, then we have $\overline{d}_j \le \overline{d}_i + a_{ij}$ for all arcs (i,j).

2.33 (A Modified Auction Algorithm for Shortest Paths)

Consider the problem of finding a shortest path from node 1 to a node t, assuming that there exists at least one such path and that all cycles have positive length. This exercise deals with a modified version of the auction algorithm, which was developed in Bertsekas [1992b], motivated by a similar earlier algorithm by Cerulli, De Leone, and Piacente [1994]. This modified version aims to use larger price increases than the original method. The algorithm maintains a price vector p and a simple path P that starts at the origin, and is initialized with $P = (1)$ and any price vector p satisfying

$$p_1 = \infty,$$

$$p_i \le a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \ne 1.$$

The algorithm terminates when the destination t becomes the terminal node of P. To describe the algorithm, define

$$A(i) = \{j \mid (i,j) \in \mathcal{A}\} \cup \{i\}, \qquad \forall\ i \in \mathcal{N},$$

$$a_{ii} = 0, \qquad \forall\ i \in \mathcal{N}.$$

The typical iteration is as follows:

> Let i be the terminal node of P, and let j_i be such that
>
> $$j_i = \arg\min_{j \in A(i)} \bigl\{a_{ij} + p_j\bigr\},$$
>
> with the extra requirement that $j_i \neq i$ whenever possible; that is, we choose $j_i \neq i$ whenever the minimum above is attained for some $j \neq i$. Set
>
> $$p_{j_i} := \min_{j \in A(i),\, j \neq j_i} \bigl\{a_{ij} + p_j\bigr\} - a_{ij_i}.$$
>
> If $j_i = i$ contract P; otherwise extend P by node j_i.

Note that if a contraction occurs, we have $j_i = i \neq 1$ and the price of the terminal node p_i is strictly increased. Note also that when an extension occurs from the terminal node i to a neighbor $j_i \neq i$, the price p_{j_i} *may be increased strictly*, while in the original auction algorithm there is no price change. Furthermore, the CS condition $p_i \leq a_{ij} + p_j$ for all (i, j) is *not* maintained. Show that:

(a) The algorithm maintains the conditions

$$\pi_i = a_{ij} + p_j, \qquad \forall\ (i, j) \in P,$$

$$\pi_i = p_i, \qquad \forall\ i \notin P,$$

where

$$\pi_i = \min\left\{p_i,\ \min_{\{j \mid (i,j) \in \mathcal{A}\}} \{a_{ij} + p_j\}\right\}, \qquad \forall\ i \in \mathcal{N}.$$

(b) Throughout the algorithm, P is a shortest path between its endnodes. *Hint*: Show that if $\tilde{P}$ is another path with the same endnodes, we have

$$\begin{aligned}\text{Length of } \tilde{P} - \text{ Length of } P &= \sum_{\{k \mid k \in \tilde{P},\, k \notin P\}} (\pi_k - p_k) - \sum_{\{k \mid k \in P,\, k \notin \tilde{P}\}} (\pi_k - p_k) \\ &\geq 0.\end{aligned}$$

(c) The algorithm terminates with a shortest path from 1 to t. *Note*: This is challenging. A proof is given in Bertsekas [1992b].

(d) Convert the shortest path problem to an equivalent assignment problem for which the conditions of part (a) are the complementary slackness conditions. Show that the algorithm is essentially equivalent to a naive auction algorithm applied to the equivalent assignment problem.

2.34 (Continuous Space Shortest Path Problems)

Consider a continuous-time dynamic system whose state $x(t) = \big(x_1(t), x_2(t)\big)$ evolves in two-dimensional space according to the differential equations

$$\dot{x}_1(t) = u_1(t), \qquad \dot{x}_2(t) = u_2(t)$$

where for each time t, $u(t) = \big(u_1(t), u_2(t)\big)$ is a two-dimensional control vector with unit norm. We want to find a state trajectory that starts at a given point $x(0)$, ends at another given point $x(T)$, and minimizes

$$\int_0^T r\big(x(t)\big)dt,$$

where $r(\cdot)$ is a given nonnegative and continuous function. The final time T and the control trajectory $\{u(t) \mid 0 \le t \le T\}$ are subject to optimization. Suppose we discretize the plane with a mesh of size δ that passes through $x(0)$ and $x(T)$, and we introduce a shortest path problem of going from $x(0)$ to $x(T)$ using moves of the following type: from each mesh point $\overline{x} = (\overline{x}_1, \overline{x}_2)$ we can go to each of the mesh points $(\overline{x}_1 + \delta, \overline{x}_2)$, $(\overline{x}_1 - \delta, \overline{x}_2)$, $(\overline{x}_1, \overline{x}_2 + \delta)$, and $(\overline{x}_1, \overline{x}_2 - \delta)$, at a cost $r(\overline{x})\delta$. Show by example that this is a bad discretization of the original problem in the sense that the shortest distance need not approach the optimal cost of the original problem as $\delta \to 0$. *Note*: This exercise illustrates a common pitfall. The difficulty is that the control constraint set (the surface of the unit sphere) should be finely discretized as well. For a proper treatment of the problem of discretization, see the original papers by Gonzalez and Rofman [1985], and Falcone [1987], the survey paper by Kushner [1990], the monograph by Kushner and Dupuis [1992], and the references cited there. For analogs of the label setting and label correcting algorithms of the present chapter, see the papers by Tsitsiklis [1995], and by Polymenakos, Bertsekas, and Tsitsiklis [1998].

3

The Max-Flow Problem

Contents

3.1. The Max-Flow and Min-Cut Problems
    3.1.1. Cuts in a Graph
    3.1.2. The Max-Flow/Min-Cut Theorem
    3.1.3. The Maximal and Minimal Saturated Cuts
    3.1.4. Decomposition of Infeasible Network Problems
3.2. The Ford-Fulkerson Algorithm
3.3. Price-Based Augmenting Path Algorithms
    3.3.1. A Price-Based Path Construction Algorithm
    3.3.2. A Price-Based Max-Flow Algorithm
3.4. Notes, Sources, and Exercises

In this chapter, we focus on the max-flow problem introduced in Example 1.3 of Section 1.2. We have a graph $(\mathcal{N}, \mathcal{A})$ with flow bounds $x_{ij} \in [b_{ij}, c_{ij}]$ for each arc (i, j), and two special nodes s and t. We want to maximize the divergence out of s over all capacity-feasible flow vectors having zero divergence for all nodes except s and t.

The max-flow problem arises in a variety of practical contexts and also as a subproblem in the context of algorithms that solve other more complex problems. For example, it can be shown that checking the existence of a feasible solution of a minimum cost flow problem, and finding a feasible solution if one exists, is essentially equivalent to a max-flow problem (see Fig. 3.1, and Exercises 3.3 and 3.4). Furthermore, a number of interesting combinatorial problems can be posed as max-flow problems (see for example Exercises 3.8-3.10).

Like the shortest path problem, the max-flow problem embodies a number of methodological ideas that are central to the more general minimum cost flow problem. In fact, whereas the shortest path problem can be viewed as a minimum cost flow problem where arc capacities play no role, the max-flow problem can be viewed as a minimum cost flow problem where arc costs play no role. In this sense, the structures of the shortest path and max-flow problems are complementary, and together provide the foundation upon which much of the algorithmic methodology of the minimum cost flow problem is built.

Central to the max-flow problem is the *max-flow/min-cut theorem*, which is one of the most celebrated theorems of network optimization. In Section 3.1, we derive this result, and we discuss some of its applications. Later, in Chapter 4, we will interpret this result as a duality theorem (see Exercise 4.4). In Section 3.2, we introduce a central algorithm for solving the max-flow problem, the Ford-Fulkerson method. This is a fairly simple method, which however can behave in interesting and surprising ways. Much research has been devoted to developing clever and efficient implementations of the Ford-Fulkerson method. We describe some of these implementations in Sections 3.2 and 3.3, and in the exercises.

3.1 THE MAX-FLOW AND MIN-CUT PROBLEMS

The key idea in the max-flow problem is very simple: a feasible flow x can be improved if we can find a path from s to t that is *unblocked* with respect to x. Pushing a positive increment of flow along such a path results in larger divergence out of s, while maintaining flow feasibility. Most (though not all) of the available max-flow algorithms are based on iterative application of this idea.

We may also ask the reverse question. If we can't find an unblocked path from s to t, is the current flow maximal? The answer is positive,

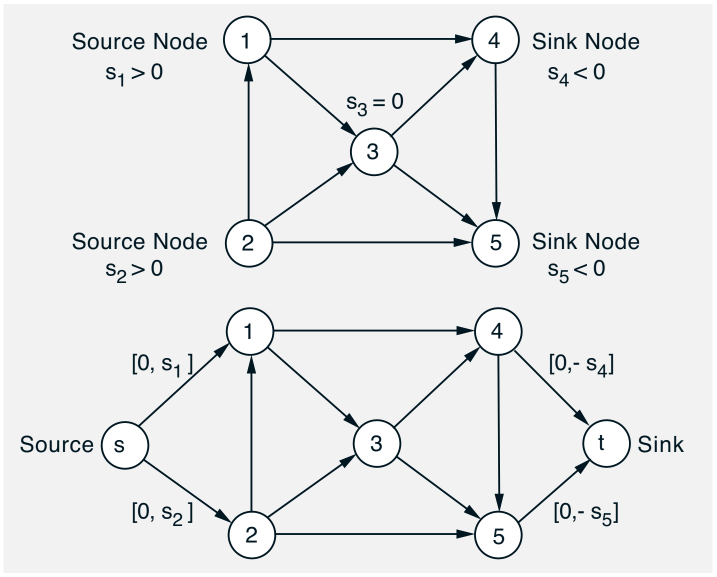


Figure 3.1: Essential equivalence of the problem of finding a feasible solution of a minimum cost flow problem and a max-flow problem. Given a set of divergences s_i satisfying $\sum_i s_i = 0$, and capacity intervals $[0, c_{ij}]$, consider the *feasibility problem* of finding a flow vector x satisfying

$$\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N}, \tag{3.1}$$

$$0 \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}. \tag{3.2}$$

Denote by $I^+ = \{i \mid s_i > 0\}$ the set of source nodes ($\{1,2\}$ in the figure) and by $I^- = \{i \mid s_i < 0\}$ the set of sink nodes ($\{4,5\}$ in the figure). If both these sets are empty, the zero vector is a feasible flow, and we are done. Otherwise, these sets are both nonempty (since $\sum_i s_i = 0$). We introduce a node s, and for all $i \in I^+$, the arcs (s,i) with flow range $[0, s_i]$. We also introduce a node t, and for all $i \in I^-$, the arcs (i,t) with flow range $[0, -s_i]$. Now consider the max-flow problem of maximizing the divergence out of s and into t, while observing the capacity constraints. Then there exists a solution to the feasibility problem of Eqs. (3.1) and (3.2), if and only if the maximum divergence out of s is equal to $\sum_{i\in I^+} s_i$. If this condition is satisfied, solutions of the feasibility problem are in one-to-one correspondence with optimal solutions of the max-flow problem.

If the capacity constraints involve lower bounds, $b_{ij} \le x_{ij} \le c_{ij}$, we may convert first the feasibility problem to one with zero lower flow bounds by a translation of variables, which replaces each variable x_{ij} with a variable $z_{ij} = x_{ij} - b_{ij}$.

Also, a max-flow problem can (in principle) be solved by an algorithm that solves the feasibility problem (we try to find a sequence of feasible flows with monotonically increasing divergence out of s, stopping with a maximum flow when no further improvement is possible). In fact, this is the main idea of the Ford-Fulkerson method, to be discussed in Section 3.2.

although the reason is not entirely obvious. For a brief justification, consider the minimum cost flow formulation of the max-flow problem, given in Example 1.3, which involves the artificial feedback arc (t, s) (see Fig. 3.2). Then, a cycle has negative cost if and only if it includes the arc (t, s), since this arc has cost -1 and is the only arc with nonzero cost. By Prop. 1.2, if a feasible flow vector x is not optimal, there must exist a simple cycle with negative cost that is unblocked with respect to x; this cycle must consist of the arc (t, s) and a path from s to t, which is unblocked with respect to x. Thus, if there is no path from s to t that is unblocked with respect to a given flow vector x, then there is no cycle of negative cost and x must be optimal.

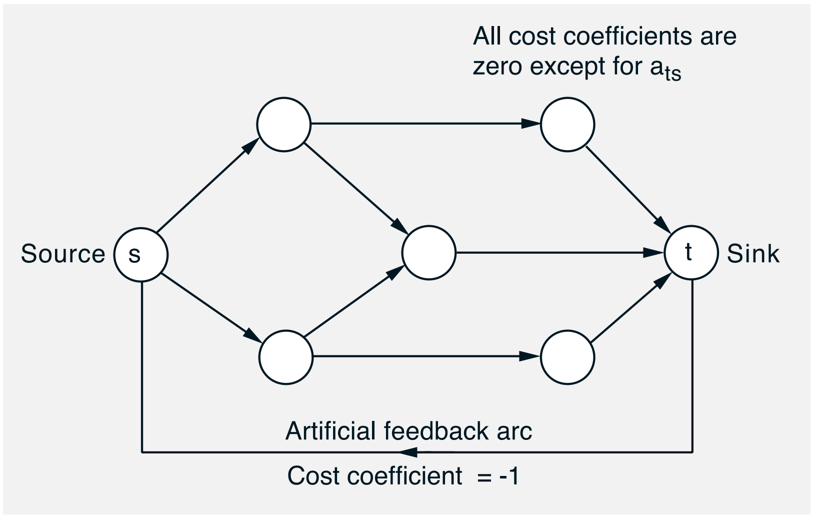


Figure 3.2: Minimum cost flow formulation of a max-flow problem, involving a feedback (t, s) arc with cost -1 and unconstrained arc flow $(-\infty < x_{ts} < \infty)$. For a nonoptimal flow x, there must exist a cycle that is unblocked with respect to x and has negative cost. Since all arcs other than the feedback arc have zero length, this cycle must contain the feedback arc. This implies that there must exist a path from s to t, which is unblocked with respect to x. Many max-flow algorithms push flow along such a path to iteratively improve an existing flow vector x.

The max-flow/min-cut theorem and the Ford-Fulkerson algorithm, to be described shortly, are based on the preceding ideas. However, rather than appealing to Prop. 1.2 (whose proof relies on the notion of a conformal decomposition), we couch the analysis of this chapter on first principles, taking advantage of the simplicity of the max-flow problem. This will also serve to develop some concepts that will be useful later. We first introduce some definitions.

3.1.1 Cuts in a Graph

A *cut* Q in a graph $(\mathcal{N}, \mathcal{A})$ is a partition of the node set $\mathcal{N}$ into two nonempty subsets, a set $\mathcal{S}$ and its complement $\mathcal{N} - \mathcal{S}$. We use the notation

$$Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}].$$

Note that the partition is ordered in the sense that the cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is distinct from the cut $[\mathcal{N} - \mathcal{S}, \mathcal{S}]$. For a cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$, we use the notation

$$Q^+ = \big\{(i,j) \in \mathcal{A} \mid i \in \mathcal{S}, j \notin \mathcal{S}\big\},$$

$$Q^- = \big\{(i,j) \in \mathcal{A} \mid i \notin \mathcal{S}, j \in \mathcal{S}\big\},$$

and we say that Q^+ and Q^- are the *sets of forward and backward arcs of the cut*, respectively. We say that the cut Q is *nonempty* if $Q^+ \cup Q^- \neq \emptyset$; otherwise we say that Q is *empty*. We say that the cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ *separates node s from node t* if $s \in \mathcal{S}$ and $t \notin \mathcal{S}$. These definitions are illustrated in Fig. 3.3.

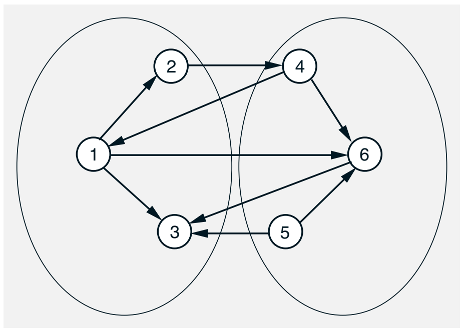


Figure 3.3: Illustration of a cut

$$Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}],$$

where $\mathcal{S} = \{1, 2, 3\}$. We have

$$Q^+ = \{(2,4), (1,6)\},$$

$$Q^- = \{(4,1), (6,3), (5,3)\}.$$

Given a flow vector x, the *flux across a nonempty cut* $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is defined to be the total net flow coming out of $\mathcal{S}$, i.e., the scalar

$$F(Q) = \sum_{(i,j) \in Q^+} x_{ij} - \sum_{(i,j) \in Q^-} x_{ij}.$$

Let us recall from Section 1.1.2 the definition of the divergence of a node i:

$$y_i = \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji}, \qquad \forall\, i \in \mathcal{N}.$$

The following calculation shows that $F(Q)$ is also equal to the sum of the divergences y_i of the nodes in $\mathcal{S}$:

$$\begin{aligned} F(Q) &= \sum_{\{(i,j)\in\mathcal{A}\mid i\in\mathcal{S},\, j\notin\mathcal{S}\}} x_{ij} - \sum_{\{(i,j)\in\mathcal{A}\mid i\notin\mathcal{S},\, j\in\mathcal{S}\}} x_{ij} \\ &= \sum_{i\in\mathcal{S}} \left(\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} \right) \qquad (3.3) \\ &= \sum_{i\in\mathcal{S}} y_i. \end{aligned}$$

(The second equality holds because the flow of an arc with both end nodes in $\mathcal{S}$ cancels out within the parentheses; it appears twice, once with a positive and once with a negative sign.)

Given lower and upper flow bounds b_{ij} and c_{ij} for each arc (i, j), the *capacity of a nonempty cut* Q is

$$C(Q) = \sum_{(i,j)\in Q^+} c_{ij} - \sum_{(i,j)\in Q^-} b_{ij}. \qquad (3.4)$$

Clearly, for any capacity-feasible flow vector x, the flux $F(Q)$ across Q is no larger than the cut capacity $C(Q)$. If $F(Q) = C(Q)$, then Q is said to be a *saturated cut with respect to* x; the flow of each forward (backward) arc of such a cut must be at its upper (lower) bound. By convention, every empty cut is also said to be saturated. The following is a simple but useful result.

Proposition 3.1: Let x be a capacity-feasible flow vector, and let s and t be two nodes. Then exactly one of the following two alternatives holds:

(1) There exists a simple path from s to t that is unblocked with respect to x.

(2) There exists a saturated cut that separates s from t.

Proof: The proof is obtained by constructing an algorithm that terminates with either a path as in (1) or a cut as in (2). This algorithm is a special case of a general method, known as *breadth-first search*, and used to find a simple path between two nodes in a graph (see Exercise 3.2). The algorithm generates a sequence of node sets $\{T_k\}$, starting with $T_0 = \{s\}$; each set T_k represents the set of nodes that can be reached from s with an unblocked path of k arcs.

Unblocked Path Search Algorithm

For $k = 0, 1, \ldots$, given T_k, terminate if either T_k is empty or $t \in T_k$; otherwise, set

$$T_{k+1} = \big\{ n \notin \cup_{i=0}^{k} T_i \mid \text{there is a node } m \in T_k, \text{ and either an arc } (m,n) \\ \text{with } x_{mn} < c_{mn}, \text{ or an arc } (n,m) \text{ with } b_{nm} < x_{nm} \big\}$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k and (m,n) or (n,m) is an arc with the property stated in the above equation, respectively.

Figure 3.4 illustrates the preceding algorithm. Since the algorithm terminates if T_k is empty, and T_k must consist of nodes not previously included in $\cup_{i=0}^{k-1} T_i$, the algorithm must eventually terminate. Let $\mathcal{S}$ be the union of the sets T_i upon termination. There are two possibilities:

(a) The final set T_k contains t, in which case, by tracing labels backward from t, a simple unblocked path P from s to t can be constructed. The forward arcs of P are of the form (m,n) with $x_{mn} < c_{mn}$ and the label of n being "(m,n)"; the backward arcs of P are of the form (n,m) with $b_{nm} < x_{nm}$ and the label of n being "(n,m)." Any cut separating s from t must contain a forward arc (m,n) of P with $x_{mn} < c_{mn}$ or a backward arc (n,m) of P with $b_{nm} < x_{nm}$, and therefore cannot be saturated. Thus, the result is proved in this case.

(b) The final set T_k is empty, in which case from the equation defining T_k, it can be seen that the cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is saturated and separates s from t. To show that there is no simple unblocked path from s to t, note that any such path must have either an arc $(m,n) \in Q^+$ with $x_{mn} < c_{mn}$ or an arc $(n,m) \in Q^-$ with $b_{nm} < x_{nm}$, which is impossible, since Q is saturated.

Q.E.D.

Exercise 3.11 provides some variations of Prop. 3.1. In particular, in place of s and t, one may use two disjoint subsets of nodes $\mathcal{N}^+$ and $\mathcal{N}^-$. Furthermore, "simple path" in alternative (1) may be replaced by "path."

3.1.2 The Max-Flow/Min-Cut Theorem

Consider now the max-flow problem, where we want to maximize the divergence out of s over all capacity-feasible flow vectors having zero divergence for all nodes other than s and t. Given any such flow vector and any cut Q separating s from t, the divergence out of s is equal to the flux across Q [cf. Eq. (3.3)], which in turn is no larger than the capacity of Q. Thus, if

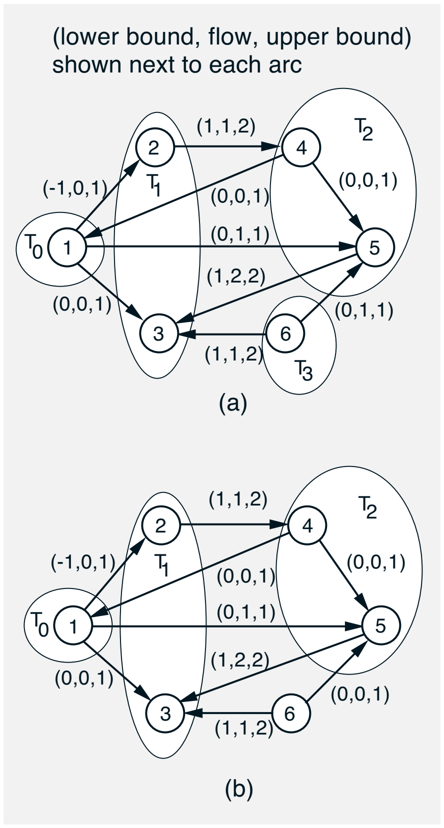


Figure 3.4: Illustration of the unblocked path search algorithm for finding an unblocked path from node 1 to node 6, or a saturated cut separating 1 from 6. The triplet (lower bound, flow, upper bound) is shown next to each arc. The figure shows the successive sets T_k generated by the algorithm. In case (a) there exists a unblocked path from 1 to 6, namely the path $(1, 3, 5, 6)$. In case (b), where the flow of arc $(6, 5)$ is at the lower bound rather than the upper bound, there is the saturated cut $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ separating 1 from 6, where $\mathcal{S} = \{1, 2, 3, 4, 5\}$ is the union of the sets T_k. Note that the algorithm works for any arc flows, and, in particular, does not require that the nodes other than the start node 1 and the end node 6 have zero divergence.

the max-flow problem is feasible, we have

$$\text{Maximum Flow} \leq \text{Capacity of } Q. \tag{3.5}$$

The following max-flow/min-cut theorem asserts that equality is attained for some Q. Part (a) of the theorem assumes the existence of an optimal solution to the max-flow problem. This assumption need not be satisfied; indeed it is possible that the max-flow problem has no feasible solution at all (consider a graph consisting of a single two-arc path from s to t, the arcs of which have disjoint feasible flow ranges). In Chapter 5, however, we will show using the theory of the simplex method (see Prop. 5.7), that the max-flow problem (and indeed every minimum cost flow problem) has an optimal solution if it has at least one feasible solution. [Alternatively, this can be shown using a fundamental result of mathematical analysis, the Weierstrass theorem, which states that a continuous function attains a maximum over a nonempty and compact set (see Appendix A and the sources given there).] If the lower flow bound is zero for every arc, the max-flow problem has at least one feasible solution, namely the zero flow vector.

Thus the theory of Chapter 5 (or the Weierstrass theorem) guarantees that the max-flow problem has an optimal solution in this case. This is stated as part (b) of the following theorem, even though its complete proof must await the developments of Chapter 5.

Proposition 3.2: (Max-Flow/Min-Cut Theorem)

(a) If x^* is an optimal solution of the max-flow problem, then the divergence out of s corresponding to x^* is equal to the minimum cut capacity over all cuts separating s from t.

(b) If all lower arc flow bounds are zero, the max-flow problem has an optimal solution, and the maximal divergence out of s is equal to the minimum cut capacity over all cuts separating s from t.

Proof: (a) Let F^* be the value of the maximum flow, that is, the divergence out of s corresponding to x^*. There cannot exist an unblocked path P from s to t with respect to x^*, since by increasing the flow of the forward arcs of P and by decreasing the flow of the backward arcs of P by a common positive increment, we would obtain a flow vector with a divergence out of s larger than F^*. Therefore, by Prop. 3.1, there must exist a cut Q, that is saturated with respect to x^* and separates s from t. The flux across Q is equal to F^* and is also equal to the capacity of Q [since Q is saturated; see Eqs. (3.3) and (3.4)]. Since we know that F^* is less or equal to the minimum cut capacity [cf. Eq. (3.5)], the result follows.

(b) See the discussion preceding the proposition. **Q.E.D.**

3.1.3 The Maximal and Minimal Saturated Cuts

Given an optimal solution x^* of the max-flow problem, there may exist several saturated cuts $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ separating s and t. We will show that out of these cuts, there exists one, called *maximal*, corresponding to the union of the sets $\mathcal{S}$. Similarly, there is a *minimal* saturated cut, corresponding to the intersection of the sets $\mathcal{S}$. (The maximal and minimal cuts coincide if and only if there is a unique saturated cut.)

Indeed, let $\overline{\mathcal{S}}$ be the union of all node sets $\mathcal{S}$ such that $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is a saturated cut separating s and t. Consider the cut

$$\overline{Q} = [\overline{\mathcal{S}}, \mathcal{N} - \overline{\mathcal{S}}].$$

Clearly $\overline{Q}$ separates s and t. If $(i, j) \in \overline{Q}^+$, then we have $x^*_{ij} = c_{ij}$ because i belongs to one of the sets $\mathcal{S}$ such that $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ is a saturated cut, and j does not belong to $\mathcal{S}$ since $j \notin \overline{\mathcal{S}}$. Thus we have $x^*_{ij} = c_{ij}$ for all $(i, j) \in \overline{Q}^+$.

Similarly, we obtain $x_{ij}^* = b_{ij}$ for all $(i,j) \in \overline{Q}^-$. Thus $\overline{Q}$ is a saturated cut separating s and t, and in view of its definition, it is the maximal such cut. By using set intersection in place of set union in the preceding argument, it is seen that we can similarly form the minimal saturated cut that separates s and t.

The maximal and minimal saturated cuts can be used to deal with infeasibility in the context of various network flow problems, as we discuss next.

3.1.4 Decomposition of Infeasible Network Problems

Consider the minimization of a separable cost function of the flow vector x,

$$\sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}),$$

subject to conservation of flow constraints

$$\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\ i \in \mathcal{N},$$

and capacity constraints

$$0 \le x_{ij} \le c_{ij}, \qquad \forall\ (i,j) \in \mathcal{A}.$$

We assume that the scalars s_i are given and satisfy $\sum_{i\in\mathcal{N}} s_i = 0$, but that the problem is infeasible, because the capacities c_{ij} are not sufficiently large to carry all the supply from the set of supply nodes

$$I^+ = \{i \mid s_i > 0\}$$

to the set of demand nodes

$$I^- = \{i \mid s_i < 0\}.$$

Then it may make sense to minimize the cost function over the set of all *maximally feasible flows*, which is the set of flow vectors x whose divergences

$$y_i = \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji}$$

satisfy

$$\begin{aligned} y_i &\ge 0 && \text{if } i \in I^+,\\ y_i &\le 0 && \text{if } i \in I^-,\\ y_i &= 0 && \text{if } i \notin I^+ \cup I^-, \end{aligned}$$

and minimize

$$\sum_{i \in \mathcal{N}} |s_i - y_i|.$$

Thus, roughly, a flow vector is maximally feasible if it is capacity-feasible, and it satisfies as much of the given demand as possible by using as much of the given supply as possible.

Note that we can find a maximally feasible flow x^* by solving the max-flow problem given in Fig. 3.1. The vector x^* defines corresponding minimal and maximal saturated cuts

$$[\mathcal{S}_{min}, \mathcal{N} - \mathcal{S}_{min}], \qquad [\mathcal{S}_{max}, \mathcal{N} - \mathcal{S}_{max}],$$

respectively, separating the supply node set P from the demand node set D. Furthermore, the flows of all arcs (i, j) that belong to these cuts are equal to x^*_{ij} for *every* maximally feasible flow vector. It can now be seen that given x^*, we can decompose the problem of minimizing the cost function over the set of maximally feasible flows into two or three feasible and independent subproblems, depending on whether $\mathcal{S}_{min} = \mathcal{S}_{max}$ or not. The node sets of these problems are $\mathcal{S}_{min}$, $\mathcal{N} - \mathcal{S}_{max}$, and $\mathcal{S}_{max} - \mathcal{S}_{min}$, (if $\mathcal{S}_{max} \neq \mathcal{S}_{min}$). The supplies for these problems are appropriately adjusted to take into account the arc flows x^*_{ij} for the arcs (i, j) of the corresponding cuts, as illustrated in Fig. 3.5.

3.2 THE FORD-FULKERSON ALGORITHM

In this section, we focus on a fundamental algorithm for solving the max-flow problem. This algorithm is of the primal cost improvement type, because it improves the primal cost (the divergence out of s) at every iteration. The idea is that, given a feasible flow vector x (i.e., one that is capacity-feasible and has zero divergence out of every node other than s and t), and a path P from s to t, which is unblocked with respect to x, we can increase the flow of all forward arcs (i, j) of P and decrease the flow of all backward arcs (i, j) of P. The maximum increment of flow change is

$$\delta = \min\big\{\{c_{ij} - x_{ij} \mid (i, j) \in P^+\}, \{x_{ij} - b_{ij} \mid (i, j) \in P^-\}\big\},$$

where P^+ is the set of forward arcs of P and P^- is the set of backward arcs of P. The resulting flow vector $\overline{x}$, given by

$$\overline{x}_{ij} = \begin{cases} x_{ij} + \delta & \text{if } (i, j) \in P^+, \\ x_{ij} - \delta & \text{if } (i, j) \in P^-, \\ x_{ij} & \text{otherwise,} \end{cases}$$

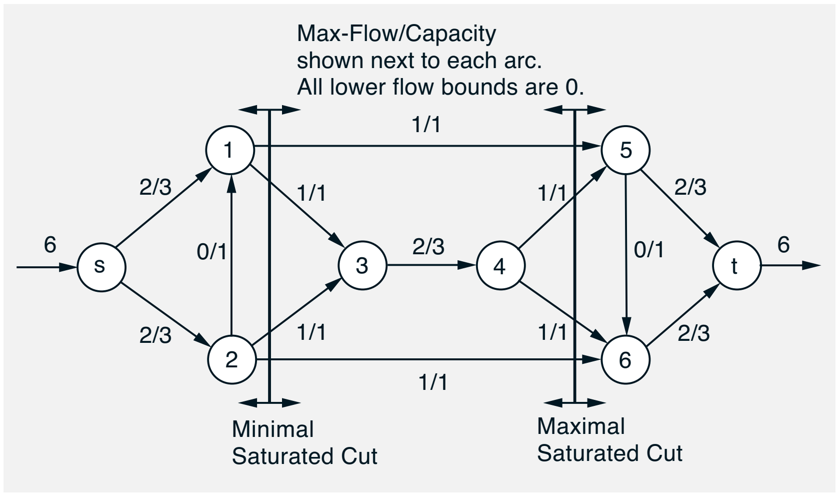


Figure 3.5: Decomposition of the problem of minimizing a separable cost function $\sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij})$ over the set of maximally feasible flow vectors into three (feasible) optimization problems. The problem here is to send 6 units of flow from node s to node t, while satisfying capacity constraints $[0, c_{ij}]$ and minimizing a cost function $\sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij})$. In this example, all arcs have capacity 1, except for arc $(3,4)$ and the incident arcs to nodes s and t, which have capacity 3. The problem is infeasible, so we consider optimization over all maximally feasible solutions. We solve the max-flow problem from s to t, and we obtain the corresponding minimal and maximal saturated cuts, as shown in the figure. Note that the flows of the arcs across these cuts are unique, although the max-flow vector is not unique.

We can now decompose the original (infeasible) optimization problem into three (feasible) optimization problems, each with the cost function $\sum_{(i,j)} f_{ij}(x_{ij})$, where the summation is over the relevant set of arcs. These problems are:

(1) The problem involving the nodes s, 1, and 2, with conservation of flow constraints

$$x_{s1} + x_{s2} = 4, \qquad -x_{21} - x_{s1} = -2, \qquad x_{21} - x_{s2} = -2.$$

(2) The problem involving the nodes 3 and 4, with conservation of flow constraint (for both nodes) $x_{34} = 2$.

(3) The problem involving the nodes 5, 6, and t, with conservation of flow constraints

$$x_{5t} + x_{56} = 2, \qquad x_{6t} - x_{56} = 2, \qquad -x_{5t} - x_{6t} = -4.$$

Note that while in this example the 2nd problem is trivial (has only one feasible solution), the 1st and 3rd problems have multiple feasible solutions.

is feasible, and it has a divergence out of s that is larger by δ than the divergence out of s corresponding to x. We refer to P as an *augmenting path*, and we refer to the operation of replacing x by $\overline{x}$ as a flow *augmentation* along P. Such an operation may also be viewed as a modification of x along the negative cost cycle consisting of P and an artificial arc (t, s) that has cost -1; see the formulation of the max-flow problem as a minimum cost flow problem in Fig. 3.2, and the discussion at the beginning of Section 3.1.

The Ford-Fulkerson algorithm starts with a feasible flow vector. If the lower flow bound is zero for all arcs, the zero flow vector can be used as a starting vector; otherwise, a preliminary phase is needed to obtain a feasible starting flow vector. This involves solving an auxiliary max-flow problem with zero lower flow bounds starting from the zero flow vector and using the Ford-Fulkerson algorithm described below (cf. Fig. 3.1 and Exercise 3.4). At each iteration the algorithm has a feasible flow vector and uses the unblocked path search method, given in the proof of Prop. 3.1, to either generate a new feasible flow vector with larger divergence out of s or terminate with a maximum flow and a minimum capacity cut.

Iteration of Ford-Fulkerson Algorithm

Use the unblocked path search method to either

(1) find a saturated cut separating s from t or

(2) find an unblocked path P with respect to x starting from s and ending at t.

In case (1), terminate the algorithm; the current flow vector solves the max-flow problem. In case (2), perform an augmentation along P and go to the next iteration.

Figure 3.6 illustrates the Ford-Fulkerson algorithm. Based on the preceding discussion, we see that with each augmentation, the Ford-Fulkerson algorithm improves the primal cost (the divergence out of s) by the augmentation increment δ. Thus, if δ is bounded below by some positive number, the algorithm can execute only a finite number of iterations and must terminate with an optimal solution. In particular, if the arc flow bounds are integer and the initial flow vector is also integer, δ is a positive integer at each iteration, and the algorithm terminates. The same is true even if the arc flow bounds and the initial flow vector are rational; by multiplication with a suitably large integer, one can scale these numbers up to integer while leaving the problem essentially unaffected.

On the other hand, if the problem data are irrational, proving termination of the Ford-Fulkerson algorithm is nontrivial. The proof (outlined in Exercise 3.12) depends on the use of the specific unblocked path search

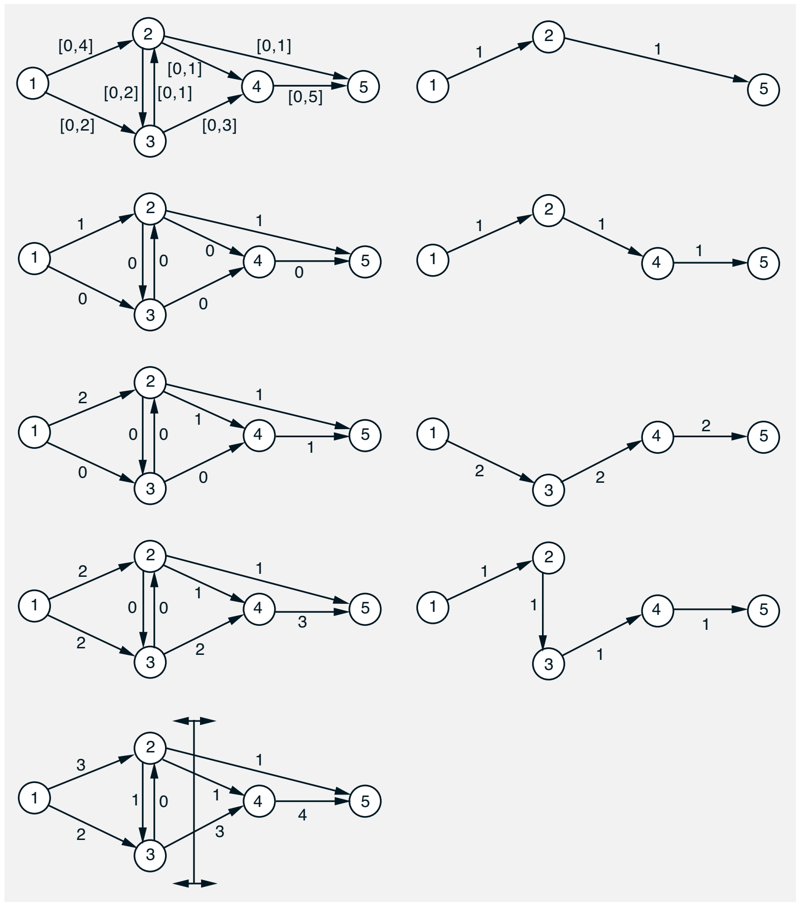


Figure 3.6: Illustration of the Ford-Fulkerson algorithm for finding a maximum flow from node $s = 1$ to node $t = 5$. The arc flow bounds are shown next to the arcs in the top left figure, and the starting flow is zero. The sequence of successive flow vectors is shown on the left, and the corresponding sequence of augmentations is shown on the right. The saturated cut obtained is $[\{1, 2, 3\}, \{4, 5\}]$. The capacity of this cut as well as the maximum flow is 5.

method of Prop. 3.1; this method (also referred to as *breadth-first search*, see Exercise 3.2) yields *augmenting paths with as few arcs as possible* (see Exercises 3.2 and 3.12). If unblocked paths are constructed using a different method, then, surprisingly, the Ford-Fulkerson algorithm need not terminate, and the generated sequence of divergences out of s may converge to a value strictly smaller than the maximum flow (for an example, see Exercise 3.7, and for a different example, see Ford and Fulkerson [1962], or Papadimitriou and Steiglitz [1982], p. 126, or Rockafellar [1984], p. 92). Even with integer problem data, if the augmenting paths are constructed using a different unblocked path search method, the Ford-Fulkerson algorithm may require a very large (pseudopolynomial) number of iterations to terminate; see Fig. 3.7.

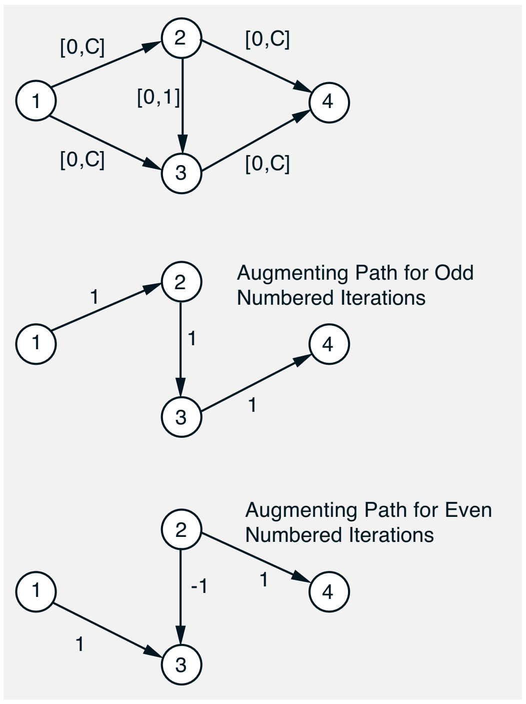

[0,C]
[0,C]
[0,1]
[0,C]
[0,C]
1
2
3
4
Augmenting Path for Odd Numbered Iterations
1
1
1
Augmenting Path for Even Numbered Iterations
-1
1
1

Figure 3.7: An example showing that if the augmenting paths used in the Ford-Fulkerson algorithm do not have a number of arcs that is as small as possible, the number of iterations may be very large. Here, C is a large integer. The maximum flow is $2C$, and can be produced after a sequence of $2C$ augmentations using the three-arc augmenting paths shown in the figure. Thus, the running time is pseudopolynomial (it is proportional to C).

If on the other hand the two-arc augmenting paths $(1, 2, 4)$ and $(1, 3, 4)$ are used, only two augmentations are needed.

Polynomial Max-Flow Algorithms

Using "shortest" augmenting paths (paths with as few arcs as possible) not only guarantees termination of the Ford-Fulkerson algorithm. It turns out that it also results in polynomial running time, as the example of Fig. 3.7 illustrates. In particular, the number of augmentations of the algorithm with shortest augmenting paths can be estimated as $O(NA)$; see Exercise 3.12. This yields an $O(NA^2)$ running time to solve the problem, since each augmentation requires $O(A)$ operations to execute the unblocked path search method and to carry out the subsequent flow update.

Much research has been devoted to developing max-flow algorithms with better than $O(NA^2)$ running time. The algorithms that we will discuss can be grouped into two main categories:

(a) Variants of the Ford-Fulkerson algorithm, which use special data structures and preprocessing calculations to generate augmenting paths efficiently. We will describe some algorithms of this type in what follows in this chapter.

(b) Algorithms that depart from the augmenting path approach, but instead move flow from the source to the sink in a less structured fashion than the Ford-Fulkerson algorithm. These algorithms, known as *preflow-push methods*, will be discussed in Section 7.3. Their underlying mechanism is related to the one of the auction algorithm described in Section 1.3.3.

The algorithms that have the best running times at present are the preflow-push methods. In particular, in Section 7.3 we will demonstrate an $O(N^3)$ running time for one of these methods, and we will describe another method with an $O(N^2 A^{1/2})$ running time. Preflow-push algorithms with even better running times exist (see the discussion in Chapter 7). It is unclear, however, whether the best preflow-push methods outperform in practice the best of the Ford-Fulkerson-like algorithms of this chapter.

In the remainder of this chapter, we will discuss efficient variants of the Ford-Fulkerson algorithm. These variants are motivated by a clear inefficiency of the unblocked path search algorithm: *it discards all the labeling information collected from the construction of each augmenting path.* Since, in a large graph, an augmentation typically has a relatively small effect on the current flow vector, each augmenting path problem is similar to the next augmenting path problem. One would thus think that the search for an augmenting path could be organized to preserve information for use in subsequent augmentations.

A prime example of an algorithm that cleverly preserves such information is the historically important algorithm of Dinic [1970], illustrated in Figure 3.8. Let us assume for simplicity that each lower arc flow bound is zero. One possible implementation of the algorithm starts with the zero flow vector and operates in phases. At the start of each phase, we have a feasible flow vector x and we construct an acyclic network, called the *layered network*, which is partitioned in layers (subsets) of nodes as follows:

Construction of the Layered Network

Layer 0 consists of just the sink node t, and layer k consists of all nodes i such that the shortest unblocked path from i to t has k arcs. Let $k(i)$ be the layer number of each node i [$k(i) = \infty$ if i does not belong to any layer].

If the source node s does not belong to any layer, there must exist a saturated cut separating s from t, so the current flow is maximal and the algorithm terminates. Otherwise, we form the layered network as follows: we delete all nodes i such that $k(i) \geq k(s)$ and their incident arcs, and we delete all remaining arcs except the arcs (i, j) such that $k(i) = k(j) + 1$ and $x_{ij} < c_{ij}$, or $k(j) = k(i) + 1$ and $x_{ij} > 0$.

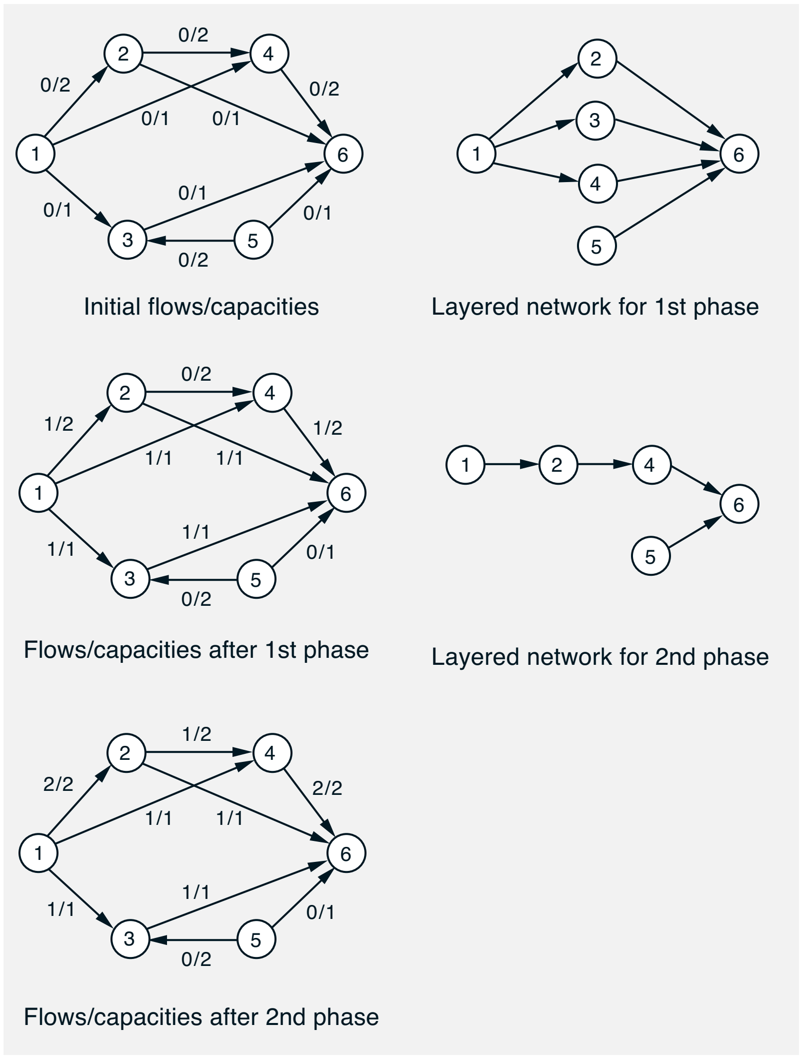

Initial flows/capacities
0/2
0/2
0/1
0/1
0/2
0/1
0/1
0/1
0/2
Layered network for 1st phase
Flows/capacities after 1st phase
1/2
0/2
1/1
1/1
1/2
1/1
1/1
0/1
0/2
Layered network for 2nd phase
Flows/capacities after 2nd phase
2/2
1/2
1/1
1/1
2/2
1/1
1/1
0/1
0/2

Figure 3.8: Illustration of Dinic's algorithm for the problem shown at the top left (node 1 is the source and node 6 is the sink).

In the first phase, there are three layers, as shown in the top right figure. There are three augmentations in the layered network ($1 \to 2 \to 6$, $1 \to 3 \to 6$, and $1 \to 4 \to 6$), and the resulting flows are shown in the middle left figure. In the second phase, there are four layers, as shown in the bottom right figure. There is only one augmenting path in the layered network ($1 \to 2 \to 4 \to 6$), and the resulting flows are shown in the bottom left figure. The algorithm then terminates because in constructing the layered network, no augmenting paths from 1 to 6 can be found.

Notice a key property of the algorithm: with each new phase, the layer number of the source node is strictly increased (from 2 to 3 in this example). This property shows that the number of phases is at most $N - 1$.

Each phase consists of successively performing augmentations using only arcs of the layered network constructed at the start of the phase, until no more augmentations can be performed.

It can be seen that with proper implementation, the layered network can be constructed in $O(A)$ time. Furthermore, the number of augmentations in each phase is at most A, since each augmentation makes at least one arc unusable for transferring flow from s to t. Given that the flow changes of each augmentation require $O(N)$ time, it follows that each phase requires $O(NA)$ time. Finally, it can be shown that with each phase, the layer number $k(s)$ of the source node s increases strictly, so that there can be at most $N-1$ phases (we leave this as Exercise 3.13 for the reader). It thus follows that the running time of the algorithm is $O(N^2A)$.

We note that the Dinic algorithm motivated a number of other max-flow algorithms with improved complexity, including an algorithm of Karzanov [1974], which has a $O(N^3)$ running time (see the sources cited at the end of the chapter). The Karzanov algorithm in turn embodied some of the ideas that were instrumental for the development of the preflow-push algorithms for max-flow, which will be discussed in Section 7.3.

3.3 PRICE-BASED AUGMENTING PATH ALGORITHMS

In this section, we develop another type of Ford-Fulkerson algorithm, which reuses information from one augmentation to the next, but does not construct shortest augmenting paths. With proper implementation, this algorithm can be shown to have an $O(N^2A)$ running time. However, there is evidence that in practice it outperforms the Dinic and the Karzanov algorithms, as well as the preflow-push algorithms of Section 7.3.

We mentioned earlier that constructing shortest augmenting paths provides some guarantee of computational efficiency in the Ford-Fulkerson algorithm. We can in fact view formally the problem of constructing such an augmenting path as a shortest path problem in a certain graph, which we will call the *reduced graph*. In particular, given a capacity-feasible flow vector x, this graph has a node set that is the same as the one of the original graph, and an arc set that is constructed from the one of the original graph by reversing the direction of some of the arcs and by duplicating some arcs and then reversing their direction. In particular, it contains:

(a) An arc (i, j) for each arc (i, j) of the original problem's graph with $x_{ij} < c_{ij}$.

(b) An arc (j, i) for each arc (i, j) of the original problem's graph with $b_{ij} < x_{ij}$.

Thus each incident arc of a node i (either outgoing or incoming) in the original graph along which flow can be pushed from i towards the opposite node, corresponds to an outgoing arc from i in the reduced graph. Furthermore, a path in the original graph is unblocked if it corresponds to a forward path of the reduced graph. Figure 3.9 illustrates the reduced graph.

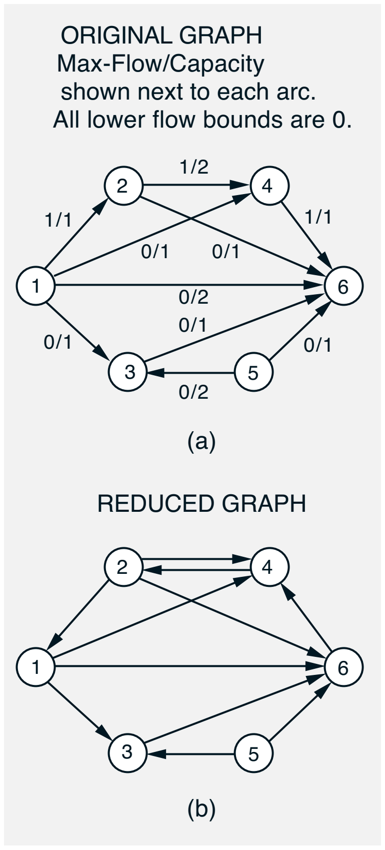


Figure 3.9: Illustration of the reduced graph corresponding to a given flow vector. Node 1 is the source, and node 6 is the sink.

Figure (a) shows the original graph, and the flow and upper flow bound next to each arc (all lower flow bounds are 0). Figure (b) shows the reduced graph. The arc (4,2) is added because the flow of arc (2,4) is strictly between the arc flow bounds. The arcs (1,2) and (4,6) are reversed because their flows are at the corresponding upper bounds.

Note that every forward path in the reduced graph, such as $(1,4,2,6)$, corresponds to an unblocked path in the original graph.

It can now be seen that, given a capacity-feasible flow vector, the problem of finding an augmenting path from s to t with a minimum number of arcs is equivalent to the problem of finding a shortest path from s to t in the corresponding reduced graph, with each arc having length 1. This suggests the simple idea of embedding one of the shortest path algorithms of Chapter 2 within the Ford-Fulkerson method. The shortest path algorithm will be used to construct the sequence of augmenting paths from s to t. Ideally, the algorithm should reuse some information from one shortest path construction to the next; we mentioned earlier that this is a key to computational efficiency.

Reusing information for a shortest path method amounts to providing some form of advanced initialization, such as label information in the context of label correcting methods or price information in the context of auction algorithms. In particular, following a shortest path augmentation, and the attendant change of the reduced graph, one would like to be able to reuse at least some of the final data of the preceding shortest path construction, to provide an advanced start for the next shortest path construction. Unfortunately, label correcting methods do not seem well suited for this purpose, because it turns out that following a change of the reduced graph due to an augmentation, many of the corresponding node labels can become unusable.

On the other hand, the auction algorithm of Section 2.6 is much better suited. The reason is that the node prices in the auction algorithm are required to satisfy the CS condition

$$p_i \leq p_j + 1 \tag{3.6}$$

for all arcs (i, j) of the reduced graph. Furthermore, upon discovery of a shortest augmenting path, there holds

$$p_i = p_j + 1$$

for all arcs (i, j) of the augmenting path. It can be seen that this equality guarantees that following a flow augmentation, the CS condition (3.6) will be satisfied for all newly created arcs of the reduced graph. As a result, following an augmentation along a shortest path found by the auction algorithm, *the node prices can be reused without modification* to start the auction algorithm for finding the next shortest augmenting path.

The preceding observations can be used to formally define a max-flow algorithm, where each augmenting path is found as a shortest path from s to t in the reduced graph using the auction algorithm as a shortest path subroutine. The initial node prices can be all equal to 0, and the prevailing prices upon discovery of a shortest augmenting path are used as the starting prices for searching for the next augmenting path. The auction algorithm maintains a path starting at s, which is contracted or extended at each iteration. The price of the terminal node of the path increases by at least 1 whenever there is a contraction. An augmentation occurs whenever the terminal node of the path is the sink node t. The overall algorithm is terminated when the price of the terminal node exceeds $N - 1$, indicating that there is no path starting at s and ending at t.

It is possible to show that, with proper implementation, the max-flow algorithm just described has an $O(N^2A)$ running time. Unfortunately, however, the practical performance of the algorithm is not very satisfactory, because the computation required by the auction/shortest path algorithm is usually much larger than what is needed to find an augmenting path. The reason is that one needs *just a path* from s to t in the reduced graph and *insisting on obtaining a shortest path may involve a substantial additional computational cost*. In what follows, we will give a price-based method that constructs a (not necessarily shortest) path from s to t. This method is similar to the auction/shortest path algorithm, but when embedded within a sequential augmenting path construction scheme, it results in a max-flow algorithm that is much faster in practice.

3.3.1 A Price-Based Path Construction Algorithm

We will describe a special method for finding a simple forward path in a directed graph $(\mathcal{N}, \mathcal{A})$ that starts at a given node s and ends at a given node t. This method will be subsequently embedded within a max-flow context to construct augmenting paths. The algorithm maintains (except upon termination) a simple forward path $P = (s, n_1, \ldots, n_k)$ and a set of integer node prices p_i, $i \in \mathcal{N}$, satisfying

$$p_i \le p_j + 1, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{3.7}$$

$$p_s < N, \qquad p_t = 0, \tag{3.8}$$

$$p_i \ge p_j, \qquad \forall\ (i,j) \in P. \tag{3.9}$$

[Note the difference with the auction/shortest path algorithm of Section 2.6, where we require that $p_i = p_j + 1$ for all arcs (i,j) of the path P, rather than $p_i \ge p_j$.]

At the start of the algorithm, we require that $P = (s)$, and that p is such that Eqs. (3.7) and (3.8) hold. The path P is modified repeatedly using the following two operations:

(a) A *contraction* of P, which deletes the last arc of P, that is, replaces the path $P = (s, n_1, \ldots, n_k)$ by the path $P = (s, n_1, \ldots, n_{k-1})$. [In the degenerate case where $P = (s)$, a contraction leaves P unchanged.]

(b) An *extension* of P, which adds to P an arc outgoing from its end node, that is, replaces the path $P = (s, n_1, \ldots, n_k)$ by a path $P = (s, n_1, \ldots, n_k, n_{k+1})$, where (n_k, n_{k+1}) is an arc.

The prices p_i may also be increased in the course of the algorithm so that, together with P, they satisfy the conditions (3.7)-(3.9). A contraction always involves a price increase of the end node n_k. An extension may or may not involve such a price increase. An extension of P is always done to a neighbor node of n_k that has minimal price. The algorithm terminates if either node t becomes the end node of P (then P is the desired path), or else $p_s \ge N$ [in view of $p_t = 0$ and $p_i \le p_j + 1$ for all arcs (i,j), as per Eqs. (3.7) and (3.8), this means that there is no forward path from s to t].

Path Construction Algorithm

Set $P = (s)$, and select p such that Eqs. (3.7) and (3.8) hold.

Step 1 (Check for contraction or extension): Let n_k be the end node of the current path P and if $n_k \neq s$, let $pred(n_k)$ be the predecessor node of n_k on P. If the set of downstream neighbors of n_k,

$$N(n_k) = \{j \mid (n_k, j) \in \mathcal{A}\},$$

is empty, set $p_{n_k} = N$ and go to Step 3. Otherwise, find a node in $N(n_k)$ with minimal price and denote it $succ(n_k)$,

$$succ(n_k) = \arg \min_{j \in N(n_k)} p_j. \tag{3.10}$$

Set

$$p_{n_k} = p_{succ(n_k)} + 1. \tag{3.11}$$

If $n_k = s$, or if

$$n_k \neq s \qquad \text{and} \qquad p_{pred(n_k)} > p_{succ(n_k)},$$

go to Step 2; otherwise go to Step 3.

Step 2 (Extend path): Extend P by node $succ(n_k)$ and the corresponding arc $\big(n_k, succ(n_k)\big)$. If $succ(n_k) = t$, terminate the algorithm; otherwise go to Step 1.

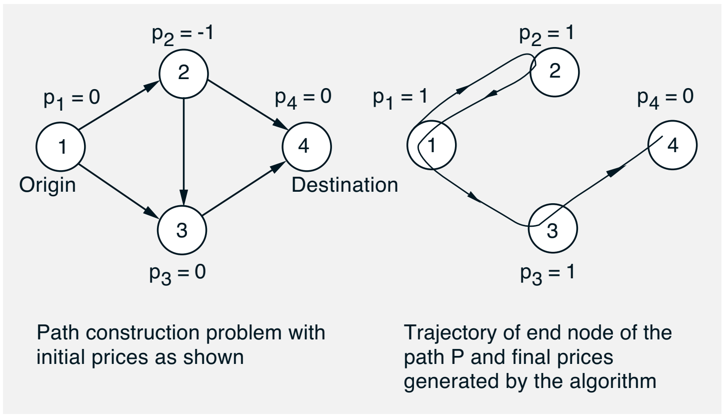
Step 3 (Contract path): If $P = (s)$ and $p_s \geq N$, terminate the algorithm; otherwise, contract P and go to Step 1.

Figure 3.10 illustrates the preceding path construction algorithm. In the special case where all initial prices are zero and there is a path from each node to t, by tracing the steps, it can be seen that the algorithm will work like depth-first search, raising to 1 the prices of the nodes of some path from s to t in a sequence of extensions with no intervening contractions. More generally, the algorithm terminates without performing any contractions if the initial prices satisfy $p_i \geq p_j$ for all arcs (i, j) and there is a path from each node to t.

Note that the algorithm does not necessarily generate a shortest path. Instead, it can be shown that it solves a special type of assignment problem by means of the auction algorithm of Section 1.3.3 (which will be further developed in Chapter 7); see Exercise 3.17.

We make the following observations:

(1) The prices remain integer throughout the algorithm [cf. Eq. (3.11)].


p2 = -1
p1 = 0
p4 = 0
p3 = 0
Origin
Destination
Path construction problem with initial prices as shown
p2 = 1
p1 = 1
p4 = 0
p3 = 1
Trajectory of end node of the path P and final prices generated by the algorithm

Iteration #	**Path P prior to iteration**	**Type of action during iteration**	**Price vector p after the iteration**
1	(1)	extension to 2	$(0,-1,0,0)$
2	$(1,2)$	contraction at 2	$(0,1,0,0)$
3	(1)	extension to 3	$(1,1,0,0)$
4	$(1,3)$	extension to 4	$(1,1,1,0)$
5	$(1,3,4)$	stop	

Figure 3.10: An example illustrating the path construction algorithm from $s = 1$ to $t = 4$, where the initial price vector is $p = (0,-1,0,0)$.

(2) The conditions (3.7)-(3.9) are satisfied each time Step 1 is entered. The proof is by induction. These conditions hold initially by assumption. Condition (3.8) is maintained by the algorithm, since termination occurs as soon as $p_s \geq N$ or t becomes the end node of P. To verify conditions (3.7) and (3.9), we note that only the price of n_k can change in Step 1, and by Eqs. (3.10) and (3.11), this price change maintains condition (3.7) for all arcs, and condition (3.9) for all arcs of P, except possibly for the arc $\big(pred(n_k), n_k\big)$ in the case of an extension with the condition

$$p_{pred(n_k)} > p_{succ(n_k)}$$

holding. In the latter case, we must have

$$p_{pred(n_k)} \geq p_{succ(n_k)} + 1$$

because the prices are integer, so by Eq. (3.11), we have

$$p_{pred(n_k)} \geq p_{n_k}$$

at the next entry to Step 1. This completes the induction.

(3) A contraction is always accompanied by a price increase. Indeed by Eq. (3.9), which was just established, upon entering Step 1 with $n_k \neq s$, we have

$$p_{n_k} \leq p_{pred(n_k)},$$

and to perform a contraction, we must have

$$p_{pred(n_k)} \leq p_{succ(n_k)}.$$

Hence

$$p_{n_k} \leq p_{succ(n_k)},$$

implying by Eq. (3.11) that $p(n_k)$ must be increased to $p_{succ(n_k)} + 1$. It can be seen, however, by example (see Fig. 3.10), that an extension may or may not be accompanied by a price increase.

(4) Upon return to Step 1 following an extension, the end node n_k satisfies [cf. Eq. (3.11)]

$$p_{pred(n_k)} = p_{n_k} + 1.$$

This, together with the condition $p_i \geq p_j$ for all $(i,j) \in P$ [cf. Eq. (3.9)], implies that the path P will not be extended to a node that already belongs to P, thereby closing a cycle. Thus P remains a simple path throughout the algorithm.

The following proposition establishes the termination properties of the algorithm.

Proposition 3.3: If there exists a forward path from s to t, the path construction algorithm terminates via Step 2 with such a path. Otherwise, the algorithm terminates via Step 3 when $p_s \geq N$.

Proof: We first note that the prices of the nodes of P are upper bounded by N in view of Eqs. (3.8) and (3.9). Next we observe that there is a price change of at least one unit with each contraction, and since the prices of the nodes of P are upper bounded by N, there can be only a finite number of contractions. Since P never contains a cycle, there can be at most $N - 1$ successive extensions without a contraction, so the algorithm must terminate. Throughout the algorithm, we have $p_t = 0$ and $p_i \leq p_j + 1$ for all arcs (i,j). Hence, if a forward path from s to t exists, we must have $p_s < N$ throughout the algorithm, including at termination, and since termination via Step 3 requires that $p_s \geq N$, it follows that the algorithm must terminate via Step 2 with a path from s to t. If a forward path from s to t does not exist, termination can only occur via Step 3, in which case we must have $p_s \geq N$. **Q.E.D.**

3.3.2 A Price-Based Max-Flow Algorithm

Let us now return to the max-flow problem. We can construct an augmenting path algorithm of the Ford-Fulkerson type that uses the path construction algorithm just presented. The algorithm consists of a sequence of augmentations, each performed using the path construction algorithm to obtain a path of the reduced graph that starts at the source node s and ends at the sink node t. As starting price vector we can use the zero vector.

An important point here is that, *following an augmentation, the price vector of the path construction algorithm can remain unchanged*. The reason is that the node prices in the path construction algorithm are required to satisfy the condition

$$p_i \leq p_j + 1 \tag{3.12}$$

for all arcs (i,j) of the reduced graph. Furthermore, upon discovery of an augmenting path P, there holds

$$p_i \geq p_j$$

for all arcs (i,j) of P. It follows that as the reduced graph changes due to the corresponding augmentation, for every newly created arc (j,i) of the reduced graph, the arc (i,j) must belong to P, so that $p_i \geq p_j$. Hence the newly created arc (j,i) of the reduced graph will also satisfy the required condition $p_j \leq p_i + 1$ [cf. Eq. (3.12)].

For a practically efficient implementation of the max-flow algorithm just described, a number of fairly complex modifications may be needed. A description of these and a favorable computational comparison with other competing methods can be found in Bertsekas [1995c], where an $O(N^2A)$ complexity bound is also shown for a suitable variant of the method.

3.4 NOTES, SOURCES, AND EXERCISES

The max-flow/min-cut theorem was independently given in Dantzig and Fulkerson [1956], Elias, Feinstein, and Shannon [1956], and Ford and Fulkerson [1956b]. The material of Section 3.1.4 on decomposition of infeasible problems is apparently new.

The proof that the Ford-Fulkerson algorithm with breadth-first search has polynomial complexity $O(NA^2)$ (Exercise 3.12) is due to Edmonds and Karp [1972]. Using the idea of a layered network, this bound was improved to $O(N^2A)$ by Dinic [1970], whose work motivated a lot of research on max-flow algorithms with improved complexity. In particular, Dinic's complexity bound was improved to $O(N^3)$ by Karzanov [1974] and by Malhotra, Kumar, and Maheshwari [1978], to $O(N^2A^{1/2})$ by Cherkasky [1977], to $O(N^{5/3}A^{2/3})$ by Galil [1980], and to $O\big(NA\log^2 N\big)$ by Galil and Naamad

[1980]. Dinic's algorithm when applied to the maximal matching problem (Exercise 3.9) can be shown to have running time $O(N^{1/2}A)$ (see Hopcroft and Karp [1973]). The survey paper by Ahuja, Magnanti, and Orlin [1989] provides a complexity-oriented account of max-flow algorithms.

The max-flow algorithm of Section 3.3 is due to Bertsekas [1995c]. This reference contains several variants of the basic method, a discussion of implementation issues, and extensive computational results that indicate a superior practical performance over competing methods, including the preflow-push algorithms of Chapter 7.

There are two important results in network optimization that deal with the existence of feasible solutions for minimum cost flow problems. The first is the *feasible distribution theorem*, due to Gale [1957] and Hoffman [1960], which is a consequence of the max-flow/min-cut theorem (Exercise 3.3). The second is the *feasible differential theorem*, due to Minty [1960], which deals with the existence of a set of prices satisfying certain constraints. This theorem is a consequence of the duality theory to be fully developed in Chapter 5, and will be given in Exercise 5.11 (see also Exercise 5.12).

E X E R C I S E S

3.1

Consider the max-flow problem of Fig. 3.11, where $s = 1$ and $t = 5$.

(a) Enumerate all cuts of the form $[\mathcal{S}, \mathcal{N} - \mathcal{S}]$ such that $1 \in \mathcal{S}$ and $5 \notin \mathcal{S}$. Calculate the capacity of each cut.

(b) Find the maximal and minimal saturated cuts.

(c) Apply the Ford-Fulkerson method to find the maximum flow and verify the max-flow/min-cut equality.

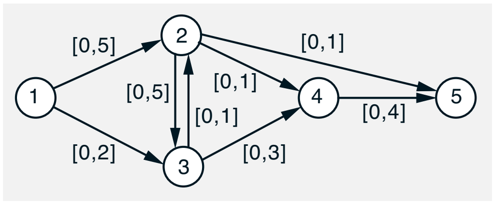


Figure 3.11: Max-flow problem for Exercise 3.1. The arc capacities are shown next to the arcs.

3.2 (Breadth-First Search)

Let i and j be two nodes of a directed graph $(\mathcal{N}, \mathcal{A})$.

(a) Consider the following algorithm, known as *breadth-first search*, for finding a path from i to j. Let $T_0 = \{i\}$. For $k = 0, 1, \ldots$, let

$$T_{k+1} = \{n \notin \cup_{p=0}^{k} T_p \mid \text{for some node } m \in T_k,\ (m,n) \text{ or } (n,m) \text{ is an arc}\},$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k such that (m,n) or (n,m) is an arc, respectively. The algorithm terminates if either (1) T_{k+1} is empty or (2) $j \in T_{k+1}$. Show that case (1) occurs if and only if there is no path from i to j. If case (2) occurs, how would you use the labels to construct a path from i to j?

(b) Show that a path found by breadth-first search has a minimum number of arcs over all paths from i to j.

(c) Modify the algorithm of part (a) so that it finds a *forward* path from i to j.

3.3 (Feasible Distribution Theorem)

Show that the minimum cost flow problem introduced in Section 1.2.1, has a feasible solution if and only if $\sum_{i\in\mathcal{N}} s_i = 0$ and for every cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ we have

$$\text{Capacity of } Q \ \geq \sum_{i\in\mathcal{S}} s_i.$$

Show also that feasibility of the problem can be determined by solving a max-flow problem with zero lower flow bounds. *Hint*: Assume first that all lower flow bounds b_{ij} are zero. Use the conversion to a max-flow problem of Fig. 3.1, and apply the max-flow/min-cut theorem. In the general case, transform the problem to one with zero lower flow bounds.

3.4 (Finding a Feasible Flow Vector)

Describe an algorithm of the Ford-Fulkerson type for checking the feasibility and finding a feasible solution of a minimum cost flow problem (cf., Section 1.2.1). If the supplies s_i and the arc flow bounds b_{ij} and c_{ij} are integer, your algorithm should be guaranteed to find an integer feasible solution (assuming at least one feasible solution exists). *Hint*: Use the conversion to a max-flow problem of Fig. 3.1.

3.5 (Integer Approximations of Feasible Solutions)

Given a graph $(\mathcal{N}, \mathcal{A})$ and a flow vector x with integer divergence, show that there exists an integer flow vector $\overline{x}$ having the same divergence vector as x and satisfying

$$|x_{ij} - \overline{x}_{ij}| < 1, \qquad \forall\ (i,j) \in \mathcal{A}.$$

Hint: For each arc (i, j), define the integer flow bounds

$$b_{ij} = \lfloor x_{ij} \rfloor, \qquad c_{ij} = \lceil x_{ij} \rceil.$$

Use the result of Exercise 3.3.

3.6

Consider a graph with arc flow range $[0, c_{ij}]$ for each arc (i, j), and let x be a capacity-feasible flow vector.

(a) Consider any subset $\mathcal{S}$ of nodes all of which have nonpositive divergence and at least one of which has negative divergence. Show that there must exist at least one arc (i, j) with $i \notin \mathcal{S}$ and $j \in \mathcal{S}$ such that $x_{ij} > 0$.

(b) Show that for each node with negative divergence there is an augmenting path that starts at that node and ends at a node with positive divergence. *Hint*: Construct such a path using an algorithm that is based on part (a).

3.7 (Ford-Fulkerson Method Counterexample)

This counterexample (from Chvatal [1983]) illustrates how the version of the Ford-Fulkerson method where augmenting paths need not have as few arcs as possible may not terminate for a problem with irrational arc flow bounds. Consider the max-flow problem shown in Fig. 3.12.

(a) Verify that an infinite sequence of augmenting paths is characterized by the table of Fig. 3.12; each augmentation increases the divergence out of the source s but the sequence of divergences converges to a value, which can be arbitrarily smaller than the maximum flow.

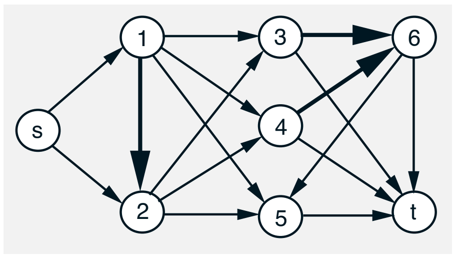
(b) Solve the problem with the Ford-Fulkerson method (where the augmenting paths involve a minimum number of arcs, as given in Section 3.2).

3.8 (Graph Connectivity – Menger's Theorem)

Let s and t be two nodes in a directed graph. Use the max-flow/min-cut theorem to show that:

(a) The maximum number of forward paths from s to t that do not share any arcs is equal to the minimum number of arcs that when removed from the graph, eliminate all forward paths from s to t.

(b) The maximum number of forward paths from s to t that do not share any nodes (other than s and t) is equal to the minimum number of nodes that when removed from the graph, eliminate all forward paths from s to t.



After Iter. #	**Augm. Path**	x_{12}	x_{36}	x_{46}	x_{65}
$6k+1$	$(s,1,2,3,6,t)$	σ	$1-\sigma^{3k+2}$	$\sigma-\sigma^{3k+1}$	0
$6k+2$	$(s,2,1,3,6,5,t)$	$\sigma-\sigma^{3k+2}$	1	$\sigma-\sigma^{3k+1}$	σ^{3k+2}
$6k+3$	$(s,1,2,4,6,t)$	σ	1	$\sigma-\sigma^{3k+3}$	σ^{3k+2}
$6k+4$	$(s,2,1,4,6,3,t)$	$\sigma-\sigma^{3k+3}$	$1-\sigma^{3k+3}$	σ	σ^{3k+2}
$6k+5$	$(s,1,2,5,6,t)$	σ	$1-\sigma^{3k+3}$	σ	σ^{3k+4}
$6k+6$	$(s,2,1,5,6,4,t)$	$\sigma-\sigma^{3k+4}$	$1-\sigma^{3k+3}$	$\sigma-\sigma^{3k+4}$	0
$6(k+1)+1$	$(s,1,2,3,6,t)$	σ	$1-\sigma^{3(k+1)+2}$	$\sigma-\sigma^{3(k+1)+1}$	0

Figure 3.12: Max-flow problem illustrating that if the augmenting paths in the Ford-Fulkerson method do not have a minimum number of arcs, then the method may not terminate. All lower arc flow bounds are zero. The upper flow bounds are larger than one, with the exception of the thick-line arcs; these are arc $(3,6)$ which has upper flow bound equal to one, and arcs $(1,2)$ and $(4,6)$ which have upper flow bound equal to $\sigma = \big(-1+\sqrt{5}\big)/2$. (Note a crucial property of σ; it satisfies $\sigma^{k+2} = \sigma^k - \sigma^{k+1}$ for all integer $k \geq 0$.) The table gives a sequence of augmentations.

3.9 (Max Matching/Min Cover Theorem (König-Egervary))

Consider a bipartite graph consisting of two sets of nodes $\mathcal{S}$ and $\mathcal{T}$ such that every arc has its start node in $\mathcal{S}$ and its end node in $\mathcal{T}$. A *matching* is a subset of arcs such that all the start nodes of the arcs are distinct and all the end nodes of the arcs are distinct. A maximal matching is a matching with a maximal number of arcs.

(a) Show that the problem of finding a maximal matching can be formulated as a max-flow problem.

(b) Define a *cover* $\mathcal{C}$ to be a subset of $\mathcal{S} \cup \mathcal{T}$ such that for each arc (i,j), either $i \in \mathcal{C}$ or $j \in \mathcal{C}$ (or both). A minimal cover is a cover with a minimal number of nodes. Show that the number of arcs in a maximal matching and the number of nodes in a minimal cover are equal. (Variants of this theorem were independently published by König [1931] and Egervary [1931].) *Hint*: Use the max-flow/min-cut theorem.

3.10 (Theorem of Distinct Representatives, Hall [1956])

Given finite sets $S_1, S_2, \ldots, S_k$, we say that the collection $\{s_1, s_2, \ldots, s_k\}$ is a system of distinct representatives if $s_i \in S_i$ for all i and $s_i \neq s_j$ for $i \neq j$. (For example, if $S_1 = \{a, b, c\}$, $S_2 = \{a, b\}$, $S_3 = \{a\}$, then $s_1 = c$, $s_2 = b$, $s_3 = a$ is a system of distinct representatives.) Show that there exists no system of distinct representatives if and only if there exists an index set $I \subset \{1, 2, \ldots, k\}$ such that the number of elements in $\cup_{i \in I} S_i$ is less than the number of elements in I. *Hint*: Consider a bipartite graph with each of the right side nodes representing an element of $\cup_{i \in I} S_i$, with each of the left side nodes representing one of the sets $S_1, S_2, \ldots S_k$, and with an arc from a left node S to a right node s if $s \in S$. Use the maximal matching/minimal cover theorem of Exercise 3.9. For additional material on this problem, see Hoffman and Kuhn [1956], and Mendelssohn and Dulmage [1958].

3.11

Prove the following generalizations of Prop. 3.1:

(a) Let x be a capacity-feasible flow vector, and let $\mathcal{N}^+$ and $\mathcal{N}^-$ be two disjoint subsets of nodes. Then exactly one of the following two alternatives holds:

  (1) There exists a simple path that starts at some node of $\mathcal{N}^+$, ends at some node of $\mathcal{N}^-$, and is unblocked with respect to x.

  (2) There exists a saturated cut $Q = [\mathcal{S}, \mathcal{N} - \mathcal{S}]$ such that $\mathcal{N}^+ \subset \mathcal{S}$ and $\mathcal{N}^- \subset \mathcal{N} - \mathcal{S}$.

(b) Show part (a) with "simple path" in alternative (1) replaced by "path". *Hint*: Use the path decomposition theorem of Exercise 1.4.

3.12 (Termination of the Ford-Fulkerson Algorithm)

Consider the Ford-Fulkerson algorithm as described in Section 3.2 (augmenting paths have as few arcs as possible). This exercise shows that the algorithm terminates and solves the max-flow problem in polynomial time, even when the problem data are irrational.

Let x^0 be the initial feasible flow vector; let x^k, $k = 1, 2, \ldots$, be the flow vector after the kth augmentation; and let P_k be the corresponding augmenting path. An arc (i, j) is said to be a k^+-*bottleneck* if (i, j) is a forward arc of P_k and $x_{ij}^k = c_{ij}$, and it is said to be a k^--*bottleneck* if (i, j) is a backward arc of P_k and $x_{ij}^k = b_{ij}$.

(a) Show that if $k < \overline{k}$ and an arc (i, j) is a k^+-bottleneck and a $\overline{k}^+$-bottleneck, then for some m with $k < m < \overline{k}$, the arc (i, j) is a backward arc of P_m. Similarly, if an arc (i, j) is a k^--bottleneck and a $\overline{k}^-$-bottleneck, then for some m with $k < m < \overline{k}$, the arc (i, j) is a forward arc of P_m.

(b) Show that P_k is a path with a minimal number of arcs over all augmenting paths with respect to x^{k-1}. (This property depends on the implementation of the unblocked path search as a breadth-first search.)

(c) For any path P that is unblocked with respect to x^k, let $n_k(P)$ be the number of arcs of P, let $a_k^+(i)$ be the minimum of $n_k(P)$ over all unblocked P from s to i, and let $a_k^-(i)$ be the minimum of $n_k(P)$ over all unblocked P from i to t. Show that for all i and k we have

$$a_k^+(i) \le a_{k+1}^+(i), \qquad a_k^-(i) \le a_{k+1}^-(i).$$

(d) Show that if $k < \overline{k}$ and arc (i,j) is both a k^+-bottleneck and a $\overline{k}^+$-bottleneck, or is both a k^--bottleneck and a $\overline{k}^-$-bottleneck, then $a_k^+(t) < a_{\overline{k}}^+(t)$.

(e) Show that the algorithm terminates after $O(NA)$ augmentations, for an $O(NA^2)$ running time.

3.13 (Layered Network Algorithm)

Consider the algorithm described near the end of Section 3.2, which uses phases and augmentations through a layered network.

(a) Provide an algorithm for constructing the layered network of each phase in $O(A)$ time.

(b) Show that the number of augmentations in each phase is at most A, and provide an implementation whereby these augmentations require $O(NA)$ total time.

(c) Show that with each phase, the layer number $k(s)$ of the source node s increases strictly, so that there can be at most $N-1$ phases.

(d) Show that with the implementations of (a) and (b), the running time of the algorithm is $O(N^2A)$.

3.14 ($O(N^{2/3}A)$ Complexity for Unit Capacity Graphs)

Consider the max-flow problem in the special case where the arc flow range is [0,1] for all arcs.

(a) Show that each path from the source to the sink that is unblocked with respect to the zero flow has at most $2N/\sqrt{M}$ arcs, where M is the value of the maximum flow. *Hint*: Let N_k be the number of nodes i such that the shortest unblocked path from s to i has k arcs. Argue that $k(k+1) \ge M$.

(b) Show that the running time of the layered network algorithm (cf. Fig. 3.8) is reduced to $O(N^{2/3}A)$. *Hint*: Argue that each arc of the layered network can be part of at most one augmenting path in a given phase, so the augmentations of each phase require $O(A)$ computation. Use part (a) to show that the number of phases is $O(N^{2/3})$.

3.15

(a) Solve the problem of Exercise 3.1 using the layered network algorithm (cf. Fig. 3.8).

(b) Construct an example of a max-flow problem where the layered network algorithm requires $N - 1$ phases.

3.16

Solve the problem of Exercise 3.1 using the max-flow algorithm of Section 3.3.2.

3.17 (Relation of Path Construction and Assignment)

The purpose of this exercise (from Bertsekas [1995c]) is to show the connection of the path construction algorithm of Section 3.3.1 with the assignment auction algorithm of Section 1.3.3.

(a) Show that the path construction problem can be converted into the problem of finding a solution of a certain assignment problem with all arc values equal to 0, as shown by example in Fig. 3.13. In particular, a forward path of a directed graph $\mathcal{G}$ that starts at node s and ends at node t corresponds to a feasible solution of the assignment problem, and conversely.

(b) Show how to relate the node prices in the path construction algorithm with the object prices of the assignment problem, so that if we apply the auction algorithm with $\epsilon = 1$, the sequence of generated prices and assignments corresponds to the sequence of generated prices and paths by the path construction algorithm.

3.18 (Decomposition of Infeasible Assignment Problems)

Apply the decomposition approach of Section 3.1.4 to an infeasible $n \times n$ assignment problem. Show that the set of persons can be partitioned in three disjoint subsets I_1, I_2, and I_3, and that the set of objects can be partitioned in three disjoint subsets J_1, J_2, and J_3 with the following properties (cf. Fig. 3.14):

(1) I_1, J_1, I_2, and J_2 are all nonempty, while I_3 and J_3 may be empty.

(2) There is no pair $(i, j) \in \mathcal{A}$ such that $i \notin I_1$ and $j \in J_1$, or $i \in I_2$ and $j \notin J_2$.

(3) If I_3 and J_3 are nonempty, then all pairs $(i, j) \in \mathcal{A}$ with $i \in I_3$ are such that $j \in J_3$.

Identify the three component problems of the decomposition in terms of the sets I_1, J_1, I_2, J_2, I_3, and J_3. Show that two of these problems are feasible asymmetric assignment problems (the numbers of persons and objects are unequal), while the third is a feasible symmetric assignment problem (the numbers of persons and objects are equal).

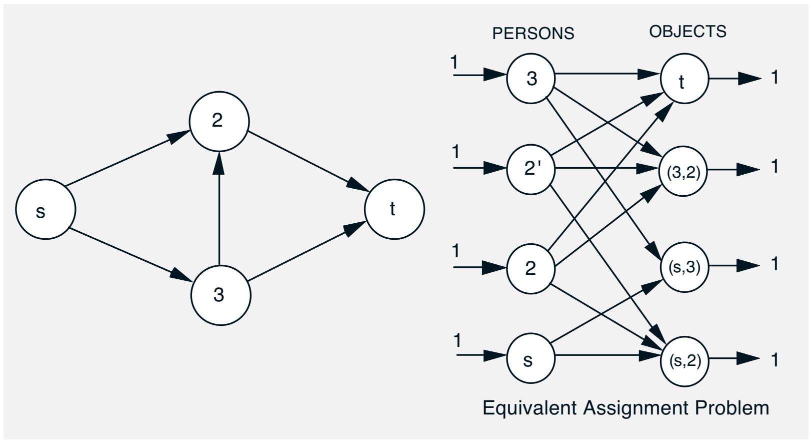


Figure 3.13: Converting the path construction problem into an equivalent feasibility problem of assigning "persons" to "objects." Each arc (i,j) of the graph $\mathcal{G}$, with $i \neq t$, is replaced by an object labeled (i,j). Each node $i \neq t$ is replaced by $R(i)$ persons, where $R(i)$ is the number of arcs of $\mathcal{G}$ that are incoming to node i (for example node 2 is replaced by the two persons 2 and $2'$). Finally, there is one person corresponding to node s and one object corresponding to node t. For every arc (i,j) of $\mathcal{G}$, with $i \neq t$, there are $R(i) + R(j)$ incoming arcs from the persons corresponding to i and j. For every arc (i,t) of $\mathcal{G}$, there are $R(i)$ incoming arcs from the persons corresponding to i. Each path that starts at s and ends at t can be associated with a feasible assignment. Conversely, given a feasible assignment, one can construct an alternating path (a sequence of alternatively assigned and unassigned pairs) starting at s and ending at t, which defines a path from s to t.

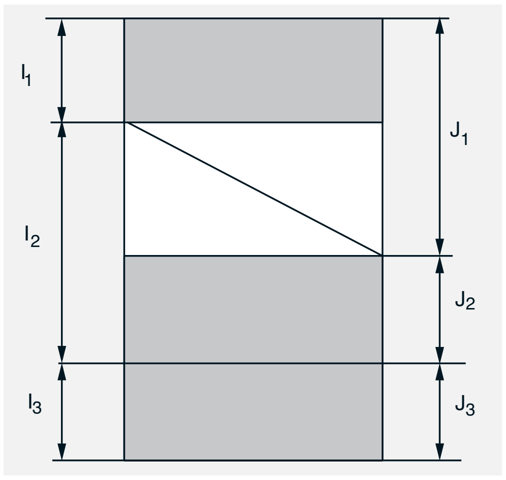


Figure 3.14: Decomposition of an infeasible assignment problem (cf. Exercise 3.18).

3.19 (Perfect Bipartite Matchings)

Consider the problem of matching n persons with n objects on a one-to-one basis (cf. Exercises 1.21 and 3.9). For each person i there is a given set of objects $A(i)$ that can be matched with i. A matching is a subset of pairs (i,j) with $j \in A(i)$, such that there is at most one pair for each person and each object. A perfect matching is one that consists of n pairs, i.e., one where every person is matched with a distinct object.

(a) Assume that there exists a perfect matching. Consider an imperfect matching $S = \big\{(i,j) \mid i \in I\big\}$, where I is a set of $m < n$ distinct persons, and let $J = \big\{j \mid \text{there exists } i \in I \text{ with } (i,j) \in S\big\}$. Show that given any $i \notin I$, there exists a sequence $\{i, j_1, i_1, j_2, i_2, \ldots, j_k, i_k, j\}$ such that $j \notin J$, the pairs $(i_1, j_1), \ldots, (i_k, j_k)$ belong to S, and $j_1 \in A(i)$, $j_2 \in A(i_1), \ldots, j_k \in A(i_{k-1})$, $j \in A(i_k)$. *Hint*: Use a max-flow formulation, and try to find an augmenting path in a suitable graph.

(b) Show that there exists a perfect matching if and only if there is no subset $I \subset \{1, \ldots, n\}$ such that the set $\cup_{i \in I} A(i)$ has fewer elements than I.

(c) (König's Theorem on Perfect Matchings) Assume that all the sets $A(i)$, $i = 1, \ldots, n$, and all the sets $B(j) = \big\{i \mid j = A(i)\big\}$, $j = 1, \ldots, n$, contain the same number of elements. Show that there exists a perfect matching.

3.20

Consider a feasible max-flow problem. Show that if the upper flow bound of each arc is increased by $\alpha > 0$, then the value of the maximum flow is increased by no more than αA, where A is the number of arcs.

3.21

A town has m dating agencies that match men and women. Agency i has a list of men and a list of women, and may match a maximum of c_i man/woman pairs from its lists. A person may be in the list of several agencies but may be matched with at most one other person. Formulate the problem of maximizing the number of matched pairs as a max-flow problem.

3.22

Consider an $n \times n$ chessboard and let A and B be two given squares.

(a) Consider the problem of finding the maximal number of knight paths that start at A, end at B, and do not overlap, in the sense that they do not share a square other than A and B. Formulate the problem as a max-flow problem.

(b) Solve the problem of part (a) using the max-flow algorithm of Section 3.3.2 for the case where $n = 8$, and the squares A and B are two opposite corners of the board.

3.23

Consider the problem of placing n queens on a chessboard of dimensions $n \times n$ so that there is no pair of queens that attack each other.

(a) Formulate the problem of finding a solution as a max-flow problem.

(b) Formulate the problem of counting the number of distinct solutions as a max-flow problem.

3.24 (Min-Flow Problem)

Consider the "opposite" to the max-flow problem, which is to minimize the divergence out of s over all capacity-feasible flow vectors having zero divergence for all nodes other than s and t.

(a) Show how to solve this problem by first finding a feasible solution, and by then using a max-flow algorithm.

(b) Derive an analog to the max-flow/min-cut theorem.

10

Network Problems with Integer Constraints

Contents

10.1. Formulation of Integer-Constrained Problems

10.2. Branch-and-Bound

10.3. Lagrangian Relaxation
- 10.3.1. Subgradients of the Dual Function
- 10.3.2. Subgradient Methods
- 10.3.3. Cutting Plane Methods
- 10.3.4. Decomposition and Multicommodity Flows

10.4. Local Search Methods
- 10.4.1. Genetic Algorithms
- 10.4.2. Tabu Search
- 10.4.3. Simulated Annealing

10.5. Rollout Algorithms

10.6. Notes, Sources, and Exercises

In this chapter, we focus again on the general nonlinear network problem of Chapter 8:

$$\begin{aligned} &\text{minimize} \quad f(x) \\ &\text{subject to} \quad x \in F, \end{aligned}$$

where x is a flow vector in a given directed graph $(\mathcal{N}, \mathcal{A})$, the feasible set F is

$$F = \left\{ x \in X \;\middle|\; \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji} = s_i,\ \forall\, i \in \mathcal{N} \right\},$$

and $f : F \mapsto \Re$ is a given real-valued function. Here s_i are given supply scalars and X is a given subset of flow vectors. We concentrate on the case where the feasible set F is discrete because the set X embodies some integer constraints and possibly some side constraints.

As we noted in Chapter 8, one may solve approximately problems with integer constraints and side constraints through some heuristic that neglects in one way or another the integer constraints. In particular, one may solve the problem as a "continuous" network flow problem and use some ad hoc method to round the fractional solution to integer. Alternatively, one may discard the complicating side constraints, obtain an integer solution of the resulting network problem, and use some heuristic to correct this solution for feasibility of the violated side constraints.

Unfortunately, there are many problems where heuristic methods of this type are inadequate, and they cannot be relied upon to produce a satisfactory solution. In such cases, one needs to strengthen the heuristics with more systematic procedures that can provide some assurance of an improved solution.

In this chapter we first describe a few examples of integer-constrained network problems, and we then focus on various systematic solution methods. In particular, in Section 10.2, we discuss the branch-and-bound method, which is in principle capable of producing an *exactly* optimal solution to an integer-constrained problem. This method relies on upper and lower bound estimates of the optimal cost of various problems that are derived from the given problem. Usually, the upper bounds are obtained with various heuristics, while the lower bounds are obtained through integer constraint relaxation or through the use of duality. A popular method for obtaining lower bounds, the Lagrangian relaxation method, is introduced in Section 10.3. This method requires the optimization of nondifferentiable functions, and two of the major algorithms that can be used for this purpose, subgradient and cutting plane methods, are discussed in Section 10.3.

Unfortunately, the branch-and-bound method is too time-consuming for exact optimal solution, so in many practical problems it can only be used as an approximation scheme. There are alternative possibilities, which do not offer the theoretical guarantees of branch-and-bound, but are much faster in practice. Two possibilities of this type, local search methods and rollout algorithms, are discussed in Sections 10.4 and 10.5, respectively.

10.1 FORMULATION OF INTEGER-CONSTRAINED PROBLEMS

There is a very large variety of integer-constrained network flow problems. Furthermore, small changes in the problem formulation can often make a significant difference in the character of the solution. As a result, it is not easy to provide a taxonomy of the major problems of interest. It is helpful, however, to study in some detail a few representative examples that can serve as paradigms when dealing with other problems that have similar structure. We have already discussed in Section 8.4 an example, the constrained shortest path problem. In this section, we provide some additional illustrative examples of broad classes of integer-constrained problems. In the exercises, we discuss several variants of these problems.

Example 10.1. Traveling Salesman Problem

An important model for scheduling a sequence of operations is the classical traveling salesman problem. This is perhaps the most studied of all combinatorial optimization problems. In addition to its use as a practical model, it has served as a testbed for a large variety of formal and heuristic approaches in discrete optimization.

In a colloquial description of the problem, a salesman wants to find a minimum mileage/cost tour that visits each of N given cities exactly once and returns to the city he started form. We associate a node with each city $i = 1, \ldots, N$, and we introduce an arc (i, j) with traversal cost a_{ij} for each ordered pair of nodes i and j. Note that we assume that the graph is *complete*; that is, there exists an arc for each ordered pair of nodes. There is no loss of generality in doing so because we can assign a very high cost a_{ij} to an arc (i, j) that is precluded from participation in the solution. We allow the possibility that $a_{ij} \neq a_{ji}$. Problems where $a_{ij} = a_{ji}$ for all i and j are sometimes called *symmetric* or *undirected* traveling salesman problems, because the direction of traversal of a given arc does not matter.

A *tour* (also called a *Hamiltonian cycle*; see Section 1.1) is defined to be a simple forward cycle that contains all the nodes of the graph. Equivalently, a tour is a connected subgraph that consists of N arcs, such that there is exactly one incoming and one outgoing arc for each node $i = 1, \ldots, N$. If we define the cost of a subgraph T to be the sum of the traversal costs of its arcs,

$$\sum_{(i,j) \in T} a_{ij},$$

the traveling salesman problem is to find a tour of minimum cost.

We formulate this problem as a network flow problem with node set $\mathcal{N} = \{1,\ldots,N\}$ and arc set $\mathcal{A} = \big\{(i,j) \mid i,j = 1,\ldots,N,\ i \neq j\big\}$, and with side constraints and 0-1 integer constraints:

$$\begin{aligned}
&\text{minimize} && \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\
&\text{subject to} && \sum_{\substack{j=1,\ldots,N\\ j\neq i}} x_{ij} = 1, \qquad i = 1,\ldots,N, \\
& && \sum_{\substack{i=1,\ldots,N\\ i\neq j}} x_{ij} = 1, \qquad j = 1,\ldots,N, \\
& && x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j)\in\mathcal{A},
\end{aligned}$$

$$\text{the subgraph with node-arc set } \big(\mathcal{N}, \{(i,j) \mid x_{ij} = 1\}\big) \text{ is connected.} \qquad (10.1)$$

Note that, given the 0-1 constraints on the arc flows and the conservation of flow equations, the last constraint can be expressed through the set of side constraints

$$\sum_{i\in S,\, j\notin S} (x_{ij} + x_{ji}) \geq 2, \quad \forall \text{ nonempty proper subsets } S \text{ of nodes.}$$

If these constraints were not present, the problem would be an ordinary assignment problem. Unfortunately, however, these constraints are essential, since without them, there would be feasible solutions involving multiple disconnected cycles, as illustrated in Fig. 10.1.

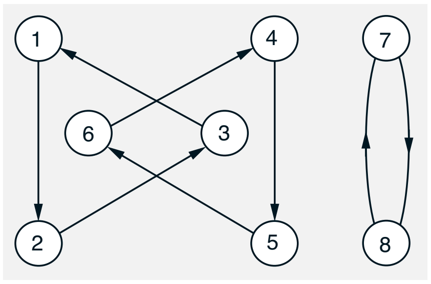


Figure 10.1: Example of an infeasible solution of a traveling salesman problem where all the constraints are satisfied except for the connectivity constraint (10.1). This solution may have been obtained by solving an $N \times N$ assignment problem and consists of multiple cycles [(1,2,3), (4,5,6), and (7,8) in the figure]. The arcs of the cycles correspond to the assigned pairs (i,j) in the assignment problem.

A simple approach for solving the traveling salesman problem is the *nearest neighbor* heuristic. We start from a path consisting of just a single node i_1 and at each iteration, we enlarge the path with a node that does not close a cycle and minimizes the cost of the enlargement. In particular, after k iterations, we have a forward path $\{i_1,\ldots,i_k\}$ consisting of distinct nodes, and at the next iteration, we add an arc (i_k, i_{k+1}) that minimizes $a_{i_k i}$ over all arcs (i_k, i) with $i \neq i_1,\ldots,i_k$. After $N-1$ iterations, all nodes are included in the path, which is then converted to a tour by adding the final arc (i_N, i_1).

Given a tour, one may try to improve its cost by using some method that changes the tour incrementally. In particular, a popular method for the symmetric case ($a_{ij} = a_{ji}$ for all i and j) is the *k-OPT heuristic*, which creates a new tour by exchanging k arcs of the current tour with another k arcs that do not belong to the tour (see Fig. 10.2). The k arcs are chosen to optimize the cost of the new tour with $O(N^k)$ computation. The method stops when no improvement of the current tour is possible through a k-interchange.

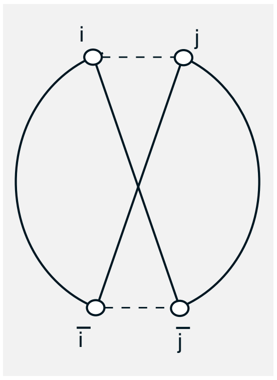


Figure 10.2: Illustration of the 2-OPT heuristic for improving a tour of the symmetric traveling salesman problem. The arcs (i,j) and $(\bar{i},\bar{j})$ are interchanged with the arcs $(i,\bar{j})$ and $(\bar{i},j)$. The choice of (i,j) and $(\bar{i},\bar{j})$ is optimized over all pairs of nonadjacent arcs of the tour.

Another possibility for constructing an initial tour is the following two-step method:

(1) Discard the side constraints (10.1), and from the resulting assignment problem, obtain a solution consisting of a collection of subtours such as the ones shown in Fig. 10.1. More generally, use some method to obtain a “reasonable” collection of subtours such that each node lies on exactly one subtour.

(2) Use some heuristic to create a tour by combining subtours. For example, any two subtours T and $\overline{T}$ can be merged into a single subtour by selecting a node $i \in T$ and a node $\bar{i} \in \overline{T}$, adding the arc $(i,\bar{i})$, deleting the unique outgoing arc (i,j) of i on the subtour T and the unique incoming arc $(\bar{j},\bar{i})$ of $\bar{j}$ on the subtour $\overline{T}$, and finally adding the arc $(\bar{j},j)$, as shown in Fig. 10.3. The pair of nodes i and $\bar{i}$ can be chosen to minimize the cost of the created subtour. This optimization requires $O(mn)$ computation, where m and n are the numbers of nodes in T and $\overline{T}$, respectively.

Still another alternative for constructing an initial tour, is to start with some spanning tree and to gradually convert it into a tour. There are quite a few heuristics based on this idea; see e.g., the book by Nemhauser and Wolsey [1988], the survey by Junger, Reinelt, and Rinaldi [1995], and the references quoted there. Unfortunately, there are no heuristics with practically useful performance guarantees for the general traveling salesman problem (Sahni and Gonzalez [1976], and Johnson and Papadimitriou [1985] make this point precise). The situation is better, however, for some special types of symmetric

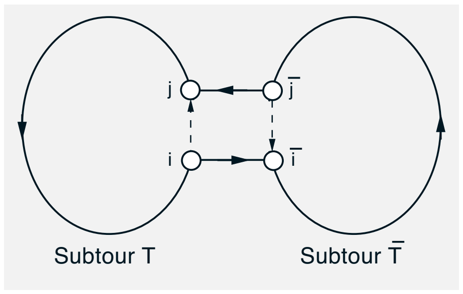


Figure 10.3: Merging two subtours T and $\overline{T}$ into a single subtour by selecting two nodes $i \in T$ and $\bar{i} \in \overline{T}$, and adding and deleting the appropriate arcs of T and $\overline{T}$.

problems where the arc costs satisfy the relation

$$a_{ij} \leq a_{ik} + a_{kj}, \qquad \text{for all nodes } i, j, k.$$

known as the *triangle inequality* (see Exercises 10.7-10.8).

Example 10.2. Fixed Charge Problems

A fixed charge problem is a minimum cost flow problem where there is an extra cost b_{ij} for each arc flow x_{ij} that is positive (in addition to the usual cost $a_{ij}x_{ij}$). Thus b_{ij} may be viewed as a "purchase cost" for acquiring the arc (i, j) and using it to carry flow.

An example of a fixed charge problem is the *facility location problem*, where we must select a subset of locations from a given candidate set, and place in each of these locations a "facility" that will serve the needs of certain "clients." There is a 0-1 decision variable associated with selecting any given location for facility placement, at a given cost. Once these variables are chosen, an assignment (or transportation) problem must be solved to optimally match clients with facilities. Mathematically, we assume that there are m clients and n locations. By $x_{ij} = 1$ (or $x_{ij} = 0$) we indicate that client i is assigned to location j at a cost a_{ij} (or is not assigned, respectively). We also introduce a 0-1 integer variable y_j to indicate (with $y_j = 1$) that a facility is placed at location j at a cost b_j. The problem is

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{j=1}^{n} b_j y_j \\
\text{subject to} \quad & \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad i = 1, \ldots, m, \\
& \sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} \leq y_j c_j, \qquad j = 1, \ldots, n, \\
& x_{ij} = 0 \text{ or } 1, \qquad \forall\, (i,j) \in \mathcal{A}, \\
& y_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n,
\end{aligned}$$

where c_j is the maximum number of customers that can be served by a facility at location j.

We can formulate this problem as a network flow problem with side constraints and integer constraints. In particular, we can view x_{ij} as the arc flows of the graph of a transportation problem (with inequality constraints). We can also view y_j as the arc flows of an artificial graph that is disconnected from the transportation graph, but is coupled to it through the side constraints $\sum_i x_{ij} \leq y_j c_j$ (see Fig. 10.4). This formulation does not necessarily facilitate the algorithmic solution of the problem, but serves to illustrate the generality of our framework for network problems with side constraints.

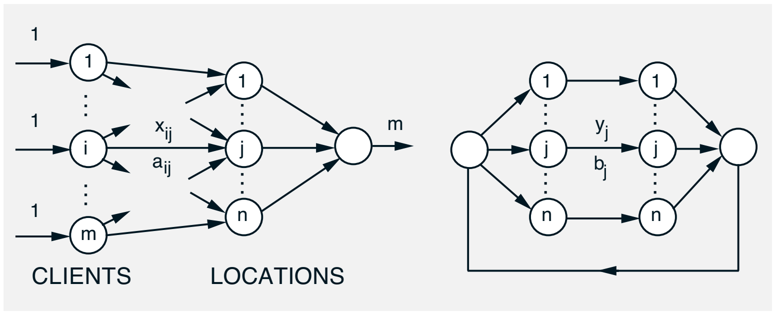


Figure 10.4: Formulation of the facility location problem as a network flow problem with side constraints and 0-1 integer constraints. There are two disconnected subgraphs: the first is a transportation-like graph that involves the flow variables x_{ij} and the second is an artificial graph that involves the flow variables y_j. The arc flows of the two subgraphs are coupled through the side constraints $\sum_i x_{ij} \leq y_j c_j$.

Example 10.3. Optimal Tree Problems

There are many network applications where one needs to construct an optimal tree subject to some constraints. For example, in data networks, a spanning tree is often used to broadcast information from some central source to all the nodes. In this context, it makes sense to assign a cost or weight a_{ij} to each arc (communication link) (i, j) and try to find a spanning tree that has minimum total weight (minimum sum of arc weights). This is the *minimum weight spanning tree problem*, which we have briefly discussed in Chapter 2 (see Exercise 2.30).

We can formulate this problem as an integer-constrained problem in several ways. For example, let x_{ij} be a 0-1 integer variable indicating whether arc (i, j) belongs to the spanning tree. Then the problem can be written as

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

$$\text{subject to} \quad \sum_{(i,j)\in\mathcal{A}} x_{ij} = N - 1,$$

$$\sum_{i\in S,\, j\notin S} (x_{ij} + x_{ji}) \geq 1, \quad \forall \text{ nonempty proper subsets } S \text{ of nodes,}$$

$$x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j) \in \mathcal{A}.$$

The first two constraints guarantee that the graph defined by the set $\{(i,j) \mid x_{ij} = 1\}$ has $N-1$ arcs and is connected, so it is a spanning tree.

In Exercise 2.30, we discussed how the minimum weight spanning tree problem can be solved with a *greedy* algorithm. An example is the *Prim-Dijkstra algorithm*, which builds an optimal spanning tree by generating a sequence of subtrees. It starts with a subtree consisting of a single node and it iteratively adds to the current subtree an incident arc that has minimum weight over all incident arcs that do not close a cycle. We indicated in Exercise 2.30 that this algorithm can be implemented so that it has an $O(N^2)$ running time. This is remarkable, because except for the minimum cost flow problems discussed in Chapters 2-7, very few other types of network optimization problems can be solved with a polynomial-time algorithm.

There are a number of variations of the minimum weight spanning tree problem. Here are some examples:

(a) There is a constraint on the number of tree arcs that are incident to a single given node. This is known as the *degree constrained minimum weight spanning tree problem.* It is possible to solve this problem using a polynomial version of the greedy algorithm (see Exercise 10.10). On the other hand, if there is a degree constraint on *every* node, the problem turns out to be much harder. For example, suppose that the degree of each node is constrained to be at most 2. Then a spanning tree subject to this constraint must be a path that goes through each node exactly once, so the problem is essentially equivalent to a symmetric traveling salesman problem (see Exercise 10.6).

(b) The *capacitated spanning tree problem.* Here the arcs of the tree are to be used for routing specified supplies from given supply nodes to given demand nodes. The tree specifies the routes that will carry the flow from the supply points to the demand points, and hence also specifies the corresponding arc flows. We require that the tree is selected so that the flow of each arc does not exceed a given capacity constraint. This is an integer-constrained problem, which is not polynomially solvable. However, there are some practical heuristic algorithms, such as an algorithm due to Esau and Williams [1966] (see Fig. 10.5).

(c) The *Steiner tree problem*, where the requirement that all nodes must be included in the tree is relaxed. Instead, we are given a subset S of the nodes, and we want to find a tree that includes the subset S and has minimum total weight. [J. Steiner (1796-1863), "the greatest geometer since Apollonius," posed the problem of finding the shortest tree spanning a given set of points on the plane.] An important application of the Steiner tree problem arises in broadcasting information over a communication network from a special node to a selected subset S of nodes. This broadcasting is most efficiently done over a Steiner tree, where the cost of each arc corresponds to the cost of communication over that arc. The Steiner tree problem also turns out to be a difficult

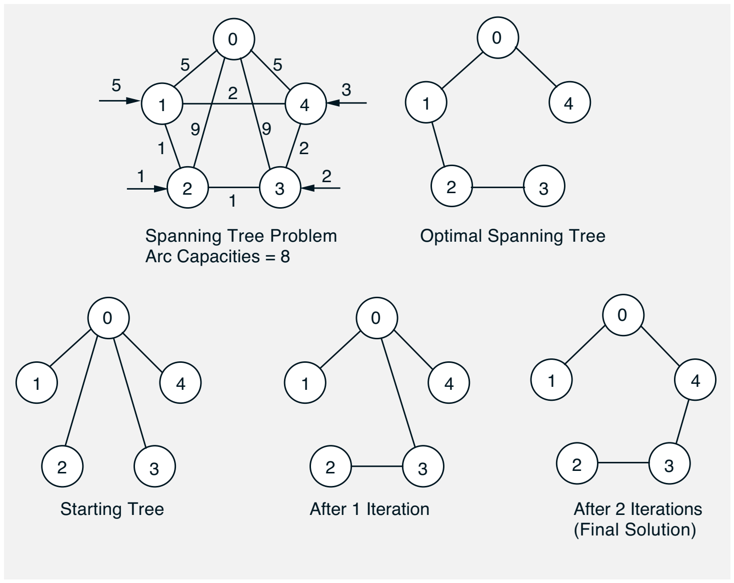


Figure 10.5: The Esau-Williams heuristic for solving a capacitated minimum weight spanning tree problem. Each arc (i, j) has a cost (or weight) a_{ij} and a capacity c_{ij}. The problem is symmetric, so that $a_{ij} = a_{ji}$ and $c_{ij} = c_{ji}$. We assume that the graph is complete [if some arcs (i, j) do not exist, we introduce them artificially with a very large cost and infinite capacity]. There is a special concentrator node 0, and for every other node $i = 1, \ldots, N$, there is a supply $s_i \geq 0$ that must be transferred to node 0 along the arcs of the spanning tree without violating the arc capacity constraints. The Esau-Williams algorithm generates a sequence of feasible spanning trees, each having a lower cost than its predecessor, by using an arc exchange heuristic. In particular, we start with a spanning tree where the concentrator node 0 is directly connected with each of the N other nodes, as in the bottom left figure [we assume that the arcs $(i, 0)$ can carry at least the supply of node i, that is, $c_{i0} \geq s_i$]. At each successive iteration, an arc $(i, 0)$ is deleted from the current spanning tree, and another arc (i, j) is added, so that:

(1) No cycle is formed.

(2) The capacity constraints of all the arcs of the new spanning tree are satisfied.

(3) The saving $a_{i0} - a_{ij}$ in cost obtained by exchanging arcs $(i, 0)$ and (i, j) is positive and is maximized over all nodes i and and j for which (1) and (2) above are satisfied.

The figure illustrates the algorithm, for the problem shown at the top left, where the cost of each arc is shown next to each arc, the capacity of each arc is 8, and the supplies of the nodes $i > 0$ are shown next to the arrows. The algorithm terminates after two iterations with the tree shown, which has a total cost of 13. Termination occurs because when arc $(1, 0)$ or $(4, 0)$ is removed and an arc that is not incident to node 0 is added, some arc capacity is violated. The optimal spanning tree has cost equal to 12.

integer-constrained problem, for which, however, effective heuristics are available (see Exercise 10.11). Note that there are degree-constrained and capacitated versions of the problem, as in (a) and (b) above.

Example 10.4. Matching Problems

A matching problem involves dividing a collection of objects into pairs. There may be some constraints regarding the objects that can be paired, and there is a benefit or value associated with matching each of the eligible pairs. The objective is to find a matching of maximal total value. We have already studied extensively special cases of matching, namely the assignment problems of Chapter 7, which are also called *bipartite matching problems*. These are matching problems where the objects are partitioned in two groups, and pairs must involve only one element from each group. Matching problems where there is no such partition are called *nonbipartite*.

To pose a matching problem as a network flow problem, we introduce a graph $(\mathcal{N}, \mathcal{A})$ that has a node for each object, and an arc (i, j) of value a_{ij} connecting any two objects i and j that can be paired. The orientation of this arc does not matter [alternatively, we may introduce both arcs (i, j) and (j, i), and assign to them equal values]. We consider a flow variable x_{ij} for each arc (i, j), where x_{ij} is 1 or 0 depending on whether objects i and j are matched or not, respectively. The objective is to maximize

$$\sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

subject to the constraints

$$\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} \leq 1, \qquad \forall\ i \in \mathcal{N}, \tag{10.2}$$

$$x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j) \in \mathcal{A}.$$

The constraint (10.2) expresses the requirement that an object can be matched with at most one other object. In a variant of the problem, it is specified that the matching should be *perfect*; that is, every object should be matched with some other object. In this case, the constraint (10.2) should be changed to

$$\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} = 1, \qquad \forall\ i \in \mathcal{N}. \tag{10.3}$$

The special case where $a_{ij} = 1$ for all arcs (i, j) is the *maximum cardinality matching problem*, i.e., finding a matching with a maximum number of matched pairs.

It is possible to view nonbipartite matching as an optimal network flow problem of the assignment type with integer constraints and with the side constraints defined by Eq. (10.2) or Eq. (10.3) (see Exercise 10.15). We would thus expect that the problem is a difficult one, and that it is not polynomially solvable (cf. the discussion of Section 8.4). However, this is not so. It turns out that nonbipartite matching has an interesting and intricate structure, which is quite unique among combinatorial and network optimization problems. In particular, nonbipartite matching problems can be solved with polynomial-time algorithms. These algorithms share some key structures with their bipartite counterparts, such as augmenting paths, but they generally become simpler and run faster when specialized to bipartite matching. One such algorithm, due to Edmonds [1965] can be implemented so that it has $O(N^3)$ running time. Furthermore, nonbipartite matching can be formulated as a linear program *without* integer constraints, and admits an analysis based on linear programming duality. We refer to the literature cited at the end of the chapter for an account.

Example 10.5. Vehicle Routing Problems

In vehicle routing problems, there is a fleet of vehicles that must pick up a number of "customers" (e.g., persons, packages, objects, etc.) from various nodes in a transportation network and deliver them at some other nodes using the network arcs. The objective is to minimize total cost subject to a variety of constraints. The cost here may include, among other things, transportation cost, and penalties for tardiness of pickup and delivery. The constraints may include vehicle capacity, and pickup and delivery time restrictions.

Vehicle routing problems are among the hardest integer programming problems because they tend to have a large number of integer variables, and also because they involve both a resource allocation and a scheduling aspect. In particular, they combine the difficult combinatorial aspects of two problems that we have already discussed:

(a) The generalized assignment problem discussed in Section 8.5 (determine which vehicles will service which customers).

(b) The traveling salesman problem discussed in Example 10.1 (determine the sequence of customer pickups and deliveries by a given vehicle). In fact, the traveling salesman problem may itself be viewed as a simple version of the vehicle routing problem, involving a single vehicle of unlimited capacity, N customers that must be picked up in some unspecified order, and a travel cost a_{ij} from customer i to customer j.

For a common type of vehicle routing problem, suppose that there are K vehicles (denoted $1, \ldots, K$) with corresponding capacities $c_1, \ldots, c_K$, which make deliveries to N customers (nodes $1, \ldots, N$) starting from a central depot (node 0). The delivery to customer i is of given size d_i, and the cost of traveling from node i to node j is denoted by a_{ij}. The problem is to find the route of each vehicle (a cycle of nodes starting from node 0 and returning to 0), that satisfies the customer delivery constraints, and the vehicle capacity constraints.

There are several heuristic approaches for solving this problem, some of which bear similarity to the heuristic approaches for solving the traveling salesman problem. For example, one may start with some set of routes, which may be infeasible because their number may exceed the number of vehicles K. One may then try to work towards feasibility by combining routes in a way that satisfies the vehicle capacity constraints, while keeping the cost as small as possible. Alternatively, one may start with a solution of a K-traveling salesmen problem (see Exercise 10.9), corresponding to the K vehicles, and then try to improve on this solution by interchanging customers between routes, while trying to satisfy the capacity constraints. These heuristics often work well, but generally they offer no guarantee of good performance, and may occasionally result in a solution that is far from optimal.

An alternative possibility, which is ultimately also based on heuristics, is to formulate the problem mathematically in a way that emphasizes its connections to both the generalized assignment problem and the traveling salesman problem. In particular, we introduce the integer variables

$$y_{ik} = \begin{cases} 1 & \text{if node } i \text{ is visited by vehicle } k, \\ 0 & \text{otherwise,} \end{cases}$$

and the vectors $y_k = (y_{1k}, \ldots, y_{Nk})$. For each $k = 1, \ldots, K$, let $f_k(y_k)$ denote the optimal cost of a traveling salesman problem involving the set of nodes

$$N_k(y_k) = \{i \mid y_{ik} = 1\}.$$

We can pose the problem as

$$\begin{aligned} \text{minimize} \quad & \sum_{k=1}^{K} f_k(y_k) \\ \text{subject to} \quad & \sum_{k=1}^{K} y_{ik} = \begin{cases} K & \text{if } i = 0, \\ 1 & \text{if } i = 1, \ldots, N, \end{cases} \\ & \sum_{i=0}^{N} d_i y_{ik} \le c_k, \qquad k = 1, \ldots, K, \\ & y_{ik} = 0 \text{ or } 1, \qquad i = 0, \ldots, N, \ k = 1, \ldots, N, \end{aligned}$$

which is a generalized assignment problem (see Section 8.5).

The difficulty with the generalized assignment formulation is that the functions f_k are generally unknown. It is possible, however, to try to approximate heuristically these functions with some linear functions of the form

$$\tilde{f}_k(y_k) = \sum_{i=0}^{N} w_{ik} y_{ik},$$

solve the corresponding generalized assignment problems for the vectors y_k, and then solve the corresponding traveling salesman problems. The weights w_{ik} can be determined in some heuristic way. For example, first specify a "seed" customer i_k to be picked up by vehicle k, and then set

$$w_{ik} = a_{0i} + a_{ii_k} - a_{0i_k},$$

which is the incremental cost of inserting customer i into the route $0 \mapsto i_k \mapsto 0$. The seed customers specify the general direction of the route taken by vehicle k, and the weight w_{ik} represents the approximate cost for picking up customer i along the way. One may select the seed customers using one of a number of heuristics, for which we refer to the literature cited at the end of the chapter.

There are several extensions and more complex variants of the preceding vehicle routing problems. For example:

(a) Some of the customers may have a "time window," in the sense that they may be served only within a given time interval. Furthermore, the total time duration of a route may be constrained.

(b) There may be multiple depots, and each vehicle may be restricted to start from a given subset of the depots.

(c) Delivery to some of the customers may not be required. Instead there may be a penalty for nondelivery or for tardiness of delivery (in the case where there are time windows).

(d) There may be precedence constraints, requiring that some of the customers be served before some others.

With additional side constraints of the type described above, the problem can become very complex. Nonetheless, with a combination of heuristics and the more formal approaches to be described in this chapter, some measure of success has been obtained in solving practical vehicle routing problems.

Example 10.6. Arc Routing Problems

Arc routing problems are similar to vehicle routing problems, except that the emphasis regarding cost and constraints is placed on arc traversals rather than node visits. Here each arc (i, j) has a cost a_{ij}, and we want to find a set of arcs that satisfy certain constraints and have minimum sum of costs. For example, a classical arc routing problem is the *Chinese postman problem*, where we want to find a cycle that traverses every arc of a graph, and has minimum sum of arc costs; here traversals in either direction and multiple traversals are allowed.† The costs of all arcs must be assumed nonnegative here in order to guarantee that the problem has an optimal solution (otherwise cycles of arbitrarily small cost would be possible by crossing back and forth an arc of negative cost).

An interesting related question is whether there exists an *Euler cycle* in the given graph, i.e., a cycle that contains every arc exactly once, with arc traversals in either direction allowed (such a cycle, if it exists, solves the Chinese postman problem since the arc costs are assumed nonnegative). This

† An analogy here is made with a postman who must traverse each arc of the road network of some town (in at least one direction), while walking the minimum possible distance. The problem was first posed by the Chinese mathematician Kwan Mei-Ko [1962].

question was posed by Euler in connection with the famous Königsberg bridge problem (see Fig. 10.6). The solution is simple: *there exists an Euler cycle if and only if the graph is connected and every node has even degree* (in an Euler cycle, the number of entrances to a node must be equal to the number of exits, so the number of incident arcs to each node must be even; for a proof of the converse, see Exercise 1.5). It turns out that even when there are nodes of odd degree, a solution to the Chinese postman problem can be obtained by constructing an Euler cycle in an expanded graph that involves some additional arcs. These arcs can be obtained by solving a nonbipartite matching problem involving the nodes of odd degree (see Exercise 10.17). Thus, since the matching problem can be solved in polynomial time as noted in Example 10.4, the Chinese postman problem can also be solved in polynomial time (see also Edmonds and Johnson [1973], who explored the relation between matching and the Chinese postman problem).

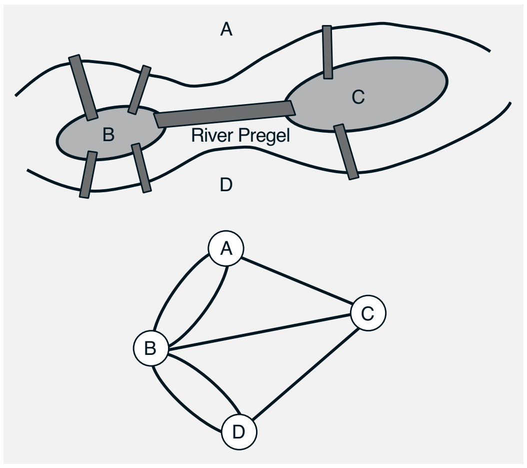


Figure 10.6: The Königsberg bridge problem, generally considered to mark the origin of graph theory. Euler attributed this problem to the citizens of Königsberg, an old port town that lies north of Warsaw on the Baltic sea (it is now called Kaliningrad). The problem, addressed by Euler in 1736, is whether it is possible to cross each of the seven bridges of the river Pregel in Königsberg exactly once, and return to the starting point. In the graph representation of the problem, shown in the figure, each bridge is associated with an arc, and each node is associated with a land area that is incident to several bridges. The question amounts to asking whether an Euler cycle exists. The answer is negative since there are nodes with odd degree.

There is also a "directed" version of the Chinese postman problem, where we want to find a *forward* cycle that traverses every arc of a graph (possibly multiple times), and has minimum sum of arc costs. It can be seen that this problem has a feasible solution if and only if the graph is strongly connected, and that it has an optimal solution if in addition all forward cycles have nonnegative cost. The problem is related to the construction of *forward* Euler cycles, in roughly the same way as the undirected Chinese postman problem was related above to the construction of an (undirected) Euler cycle. Exercise 1.8 states the basic result about the existence of a forward Euler cycle: such a cycle exists if and only if the number of incoming arcs to each node is equal to the number of its outgoing arcs. A forward Euler cycle, if it exists, is also a solution to the directed Chinese postman problem. More generally, it turns out that a solution to the directed Chinese postman problem (assuming one exists) can be obtained by finding a directed Euler cycle in an associated graph obtained by solving a certain minimum cost flow problem (see Exercise 10.17).

By introducing different constraints, one may obtain a large variety of arc routing problems. For example, a variant of the Chinese postman problem is to find a cycle of minimum cost that traverses only a given subset of the arcs. This is known as the *rural postman problem*. Other variants are characterized by arc time-windows and arc precedence constraints, similar to vehicle routing problem variants discussed earlier. In fact, it is always possible to convert an arc routing problem to a "node routing problem," where the constraints are placed on some of the nodes rather than on the arcs. This can be done by replacing each arc (i,j) with two arcs (i,k_{ij}) and (k_{ij},j) separated by an artificial middle node k_{ij}. Traversal of an arc (i,j) then becomes equivalent to visiting the artificial node k_{ij}. However, this transformation often masks important characteristics of the problem. For example it would be awkward to pose the question of existence of an Euler cycle as a node routing problem.

Example 10.7. Multidimensional Assignment Problems

In the assignment problems we have considered so far, we group the nodes of the graph in pairs. Multidimensional assignment problems involve the grouping of the nodes in subsets with more than two elements, such as triplets or quadruplets of nodes. For an example of a 3-dimensional assignment problem, suppose that the performance of a job j requires a machine m and a worker w, and that there is a given value a_{jmw} corresponding to the triplet (j,m,w). Given a set of jobs J, a set of machines M, and a set of workers W, we want to find a collection of job/machine/worker triplets that has maximum total value.

To pose this problem mathematically, we introduce 0-1 integer variables

$$x_{jmw} = \begin{cases} 1 & \text{if job } j \text{ is performed at machine } m \text{ by worker } w, \\ 0 & \text{otherwise,} \end{cases}$$

and we maximize

$$\sum_{j\in J}\sum_{m\in M}\sum_{w\in W} a_{jmw}x_{jmw}$$

subject to standard assignment constraints. In particular, if the numbers of jobs, machines, and workers are all equal, and all jobs must be assigned, we have the constraints

$$\sum_{m\in M}\sum_{w\in W} x_{jmw} = 1, \qquad \forall\, j \in J,$$

$$\sum_{j\in J}\sum_{w\in W} x_{jmw} = 1, \qquad \forall\, m \in M,$$

$$\sum_{j\in J}\sum_{m\in M} x_{jmw} = 1, \qquad \forall\, w \in W.$$

In alternative formulations, some of these constraints may involve inequalities.

An important and particularly favorable special case of the problem arises when the values a_{jmw} have the *separable* form

$$a_{jmw} = \beta_{jm} + \gamma_{mw},$$

where β_{jm} and γ_{mw} are given scalars. In this case, there is no coupling between jobs and workers, and the problem can be solved by solving two decoupled (2-dimensional) assignment problems: one involving the pairing of jobs and machines, with the β_{jm} as values, and the other involving the pairing of machines and workers, with the γ_{mw} as values. In general, however, the 3-dimensional assignment problem is a difficult integer programming problem, for which there is no known polynomial algorithm.

A simple heuristic approach is based on relaxing each of the constraints in turn. In particular, suppose that the constraint on the workers is neglected first. It can then be seen that the problem takes the 2-dimensional assignment form

$$\begin{aligned} &\text{maximize} && \sum_{j\in J}\sum_{m\in M} b_{jm} y_{jm} \\ &\text{subject to} && \sum_{m\in M} y_{jm} = 1, \qquad \forall\, j \in J, \\ &&& \sum_{j\in J} y_{jm} = 1, \qquad \forall\, m \in M, \\ &&& y_{jm} = 0 \text{ or } 1, \qquad \forall\, j \in J,\, m \in M, \end{aligned}$$

where

$$b_{jm} = \max_{w\in W} a_{jmw}, \tag{10.4}$$

and $y_{jm} = 1$ indicates that job j must be performed at machine m. For each $j \in J$, let j_m be the job assigned to machine m, according to the solution of this problem. We can now optimally assign machines m to workers w, using as assignment values

$$c_{mw} = a_{j_m mw},$$

and obtain a 3-dimensional assignment $\{(j_m, m, w_m) \mid m \in M\}$. It can be seen that this approach amounts to *enforced separation*, whereby we replace the values a_{jmw} with the separable approximations $b_{jm} + c_{mw}$. In fact, it can be shown that if the problem is *ϵ-separable*, in the sense that for some (possibly unknown) $\overline{\beta}_{jm}$ and $\overline{\gamma}_{mw}$, and some $\epsilon \geq 0$, we have

$$|\overline{\beta}_{jm} + \overline{\gamma}_{mw} - a_{jmw}| \leq \epsilon, \qquad \forall\, j \in J,\, m \in M,\, w \in W,$$

then the assignment $\{(j_m, m, w_m) \mid m \in M\}$ obtained using the preceding enforced separation approach achieves the optimal value of the problem within $4n\epsilon$, where n is the cardinality of the sets J, M, and W (see Exercise 10.31).

The enforced separation approach is simple and can be generalized to multidimensional assignment problems of dimension more than 3. However, it often results in significant loss of optimality. A potential improvement is to introduce some corrections to the values b_{jm} that reflect some dependence on the values of workers. For example, we can use instead of the values b_{jm} of Eq. (10.4), the modified values

$$\hat{b}_{jm} = \max_{w \in W} \{a_{jmw} - \mu_w\},$$

where μ_w is a nonnegative scalar that can be viewed as a *wage* to be paid to worker w. This allows the possibility of adjusting the scalars μ_w to some "optimal" values. Methods for doing this will be discussed in Section 10.3 in the context of the Lagrangian relaxation method, where we will view μ_w as a Lagrange multiplier corresponding to the constraint $\sum_{j \in J} \sum_{m \in M} x_{jmw} = 1$.

There are several extensions of the multidimensional assignment problem. For example, we may have transportation constraints, where multiple jobs can be performed on the same machine, and/or multiple machines can be operated by a single worker. In this case, our preceding discussion of the enforced separation heuristic applies similarly. We may also have generalized assignment constraints such as

$$\sum_{j \in J} \sum_{w \in W} g_{jmw} x_{jmw} \le 1, \qquad \forall\, m \in M,$$

where g_{jmw} represents the portion of machine m needed to perform job j by worker w. In this case, the enforced separation heuristic results in difficult integer-constrained generalized assignment problems, which we may have to solve heuristically. Alternatively, we may use the more formal methodology of the next two sections.

10.2 BRANCH-AND-BOUND

The branch-and-bound method implicitly enumerates all the feasible solutions, using calculations where the integer constraints of the problem are relaxed. The method can be very time-consuming, but is in principle capable of yielding an exactly optimal solution.

To describe the branch-and-bound method, consider the general discrete optimization problem

$$\begin{aligned} &\text{minimize} \quad f(x) \\ &\text{subject to} \quad x \in F, \end{aligned}$$

where the feasible set F is a *finite* set. The branch-and-bound algorithm uses an acyclic graph known as the *branch-and-bound tree*, which corresponds to a progressively finer partition of F. In particular, the nodes of this graph correspond to a collection $\mathcal{F}$ of subsets of F, which is such that:

1. $F \in \mathcal{F}$ (i.e., the set of all solutions is a node).
2. If x is a feasible solution, then $\{x\} \in \mathcal{F}$ (i.e., each solution viewed as a singleton set is a node).
3. If a set $Y \in \mathcal{F}$ contains more than one solution $x \in F$, then there exist disjoint sets $Y_1, \ldots, Y_n \in \mathcal{F}$ such that

$$\bigcup_{i=1}^{n} Y_i = Y.$$

   The set Y is called the *parent* of $Y_1, \ldots, Y_n$, and the sets $Y_1, \ldots, Y_n$ are called the *children* or *descendants* of Y.
4. Each set in $\mathcal{F}$ other than F has a parent.

The collection of sets $\mathcal{F}$ defines the branch-and-bound tree as in Fig. 10.7. In particular, this tree has the set of all feasible solutions F as its root node and the singleton solutions $\{x\}$, $x \in F$, as terminal nodes. The arcs of the graph are those that connect parents Y and their children Y_i.

The key assumption in the branch-and-bound method is that for every nonterminal node Y, there is an algorithm that calculates:

(a) A lower bound $\underline{f}_Y$ to the minimum cost over Y

$$\underline{f}_Y \le \min_{x \in Y} f(x).$$

(b) A feasible solution $\overline{x} \in Y$, whose cost $f(\overline{x})$ can serve as an upper bound to the optimal cost of the original problem $\min_{x \in F} f(x)$.

The main idea of the branch-and-bound algorithm is to save computation by discarding the nodes/subsets of the tree that have no chance of containing an optimal solution. In particular, the algorithm selects nodes Y from the branch-and-bound tree, and checks whether the lower bound $\underline{f}_Y$ exceeds the best available upper bound [the minimal cost $f(\overline{x})$ over all feasible solutions $\overline{x}$ found so far]. If this is so, we know that Y cannot contain an optimal solution, so all its descendant nodes in the tree need not be considered further.

To organize the search through the tree, the algorithm maintains a node list called OPEN, and also maintains a scalar called UPPER, which is equal to the minimal cost over feasible solutions found so far. Initially, OPEN contains just F, and UPPER is equal to ∞ or to the cost $f(\overline{x})$ of some feasible solution $\overline{x} \in F$.

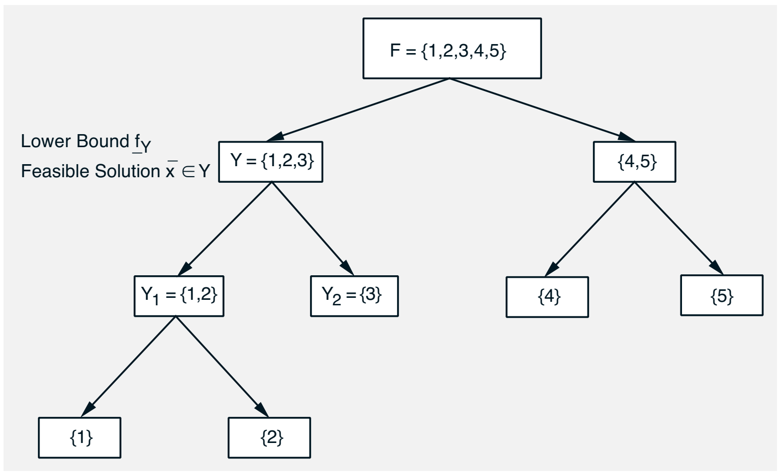

F = {1,2,3,4,5}
Lower Bound $\underline{f}_Y$
Feasible Solution $\bar{x} \in Y$
Y = {1,2,3}
{4,5}
Y1 = {1,2}
Y2 = {3}
{4}
{5}
{1}
{2}

Figure 10.7: Illustration of a branch-and-bound tree. Each node Y (a subset of the feasible set F), except those consisting of a single solution, is partitioned into several other nodes (subsets) $Y_1, \ldots, Y_n$. The original feasible set is divided repeatedly into subsets until no more division is possible. For each node/subset Y of the tree, one may compute a lower bound $\underline{f}_Y$ to the optimal cost of the corresponding restricted subproblem $\min_{x \in Y} f(x)$, and a feasible solution $\overline{x} \in Y$, whose cost can serve as an upper bound to the optimal cost $\min_{x \in F} f(x)$ of the original problem. The idea is to use these bounds to economize computation by eliminating nodes of the tree that cannot contain an optimal solution.

Branch-and-Bound Algorithm

Step 1: Remove a node Y from OPEN. For each child Y_j of Y, do the following: Find the lower bound $\underline{f}_{Y_j}$ and a feasible solution $\overline{x} \in Y_j$. If

$$\underline{f}_{Y_j} < \text{ UPPER},$$

place Y_j in OPEN. If in addition

$$f(\overline{x}) < \text{ UPPER},$$

set

$$\text{UPPER } = f(\overline{x})$$

and mark $\overline{x}$ as the best solution found so far.

Step 2: (Termination Test) If OPEN is nonempty, go to step 1. Otherwise, terminate; the best solution found so far is optimal.

A node Y_j that is not placed in OPEN in Step 1 is said to be *fathomed*. Such a node cannot contain a better solution than the best solution found so far, since the corresponding lower bound $\underline{f}_{Y_j}$ is not smaller than UPPER. Therefore nothing is lost when we drop this node from further consideration and forego the examination of its descendants. Regardless of how many nodes are fathomed, the branch-and-bound algorithm is guaranteed to examine either explicitly or implicitly (through fathoming) all the terminal nodes, which are the singleton solutions. As a result, it will terminate with an optimal solution.

Note that a small (near-optimal) value of UPPER and tight lower bounds $\underline{f}_{Y_j}$ contribute to the quick fathoming of large portions of the branch-and-bound tree, and an early termination of the algorithm, with either an optimal solution or a solution that is within some given tolerance of being optimal. In fact, a popular variant, aimed at accelerating the branch-and-bound algorithm, is to fix an $\epsilon > 0$, and replace the test

$$\underline{f}_{Yj} < \text{ UPPER}$$

with

$$\underline{f}_{Yj} < \text{ UPPER} - \epsilon$$

in Step 1. This variant may terminate much faster, while the best solution obtained upon termination is guaranteed to be within ϵ of optimality.

Other variations of branch-and-bound relate to the method for selecting a node from OPEN in Step 1. For example, a possible strategy is to choose the node with minimal lower bound; alternatively, one may choose the node containing the best solution found so far. In fact it is neither practical nor necessary to generate a priori the branch-and-bound tree. Instead, one may adaptively decide on the order and the manner in which the nodes are partitioned into descendants based on the progress of the algorithm.

Branch-and-bound typically uses "continuous" network optimization problems (without integer constraints) to obtain lower bounds to the optimal costs of the restricted problems $\min_{x \in Y} f(x)$ and to construct corresponding feasible solutions. For example, suppose that our original problem has a convex cost function, and a feasible set F that consists of convex set constraints and side constraints, *plus the additional constraint that all the arc flows must be 0 or 1*. Then a restricted subset Y may specify that the flows of some given subset of arcs are fixed at 0 or at 1, while the remaining arc flows may take either the value 0 or the value 1. A lower bound to the restricted optimal cost $\min_{x \in Y} f(x)$ is then obtained by relaxing the 0-1 constraint on the latter arc flows, thereby allowing them to take any value in the interval $[0, 1]$ and resulting in a convex network problem with side constraints. Thus the solution by branch-and-bound of a network problem with convex cost and side constraints *plus* additional integer constraints requires the solution of many convex network problems with side constraints but *without* integer constraints.

Example 10.8. Facility Location Problems

Let us consider the facility location problem introduced in Example 10.2, which involves m clients and n locations. By $x_{ij} = 1$ (or $x_{ij} = 0$) we indicate that client i is assigned to location j at a cost a_{ij} (or is not assigned, respectively). We also introduce a 0-1 integer variable y_j to indicate (with $y_j = 1$) that a facility is placed at location j at a cost b_j. The problem is

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{j=1}^{n} b_j y_j \\
\text{subject to} \quad & \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad i = 1,\ldots,m, \\
& \sum_{\{i\mid(i,j)\in\mathcal{A}\}} x_{ij} \le y_j c_j, \qquad j = 1,\ldots,n, \\
& x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j)\in\mathcal{A}, \\
& y_j = 0 \text{ or } 1, \qquad j = 1,\ldots,n,
\end{aligned}$$

where c_j is the maximum number of customers that can be served by a facility at location j.

The solution of the problem by branch-and-bound involves the partition of the feasible set F into subsets. The choice of subsets is somewhat arbitrary, but it is convenient to select subsets of the form

$$F(J_0, J_1) = \big\{(x,y)\in F \mid y_j = 0,\ \forall\ j\in J_0,\ y_j = 1,\ \forall\ j\in J_1\big\},$$

where J_0 and J_1 are disjoint subsets of the index set $\{1,\ldots,n\}$ of facility locations. Thus, $F(J_0, J_1)$ is the subset of feasible solutions such that:

a facility is placed at the locations in J_1,

no facility is placed at the locations in J_0,

a facility may or may not be placed at the remaining locations.

For each node/subset $F(J_0, J_1)$, we may obtain a lower bound and a feasible solution by solving the linear program where all integer constraints are relaxed except for the variables y_j, $j\in J_0\cup J_1$, which have been fixed at either 0 or 1:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{j=1}^{n} b_j y_j \\
\text{subject to} \quad & \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad i = 1,\ldots,m,
\end{aligned}$$

$$\sum_{\{i \mid (i,j)\in\mathcal{A}\}} x_{ij} \le y_j c_j, \qquad j = 1,\ldots,n,$$
$$x_{ij} \in [0,1], \qquad \forall\ (i,j)\in\mathcal{A},$$
$$y_j \in [0,1], \qquad \forall\ j \notin J_0 \cup J_1,$$
$$y_j = 0, \quad \forall\ j\in J_0, \qquad y_j = 1, \quad \forall\ j \in J_1.$$

As an illustration, let us work out the example shown in Figure 10.8, which involves 3 clients and 2 locations. The facility capacities at the two locations are $c_1 = c_2 = 3$. The cost coefficients a_{ij} and b_j are shown next to the corresponding arcs. The optimal solution corresponds to $y_1 = 0$ and $y_2 = 1$, that is, placing a facility only in location 2 and serving all the clients at that facility. The corresponding optimal cost is

$$f^* = 5.$$

Let us apply the branch-and-bound algorithm using the tree shown in Fig. 10.8. We first consider the top node $\big(J_0 = \varnothing, J_1 = \varnothing\big)$, where neither y_1 nor y_2 is fixed at 0 or 1. The lower bound $\underline{f}_Y$ is obtained by solving the (relaxed) linear program

$$\begin{aligned}
\text{minimize}\quad & (2x_{11}+x_{12}) + (2x_{21}+x_{22}) + (x_{31}+2x_{32}) + 3y_1 + y_2\\
\text{subject to}\quad & x_{11}+x_{12} = 1, \qquad x_{21}+x_{22} = 1, \qquad x_{31}+x_{32} = 1,\\
& x_{11}+x_{21}+x_{31} \le 3y_1, \qquad x_{12}+x_{22}+x_{32} \le 3y_2,\\
& 0 \le x_{ij} \le 1, \qquad \forall\ (i,j)\in\mathcal{A},\\
& 0 \le y_1 \le 1, \qquad 0 \le y_2 \le 1.
\end{aligned}$$

The optimal solution of this program is

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) = (1,2),(2,2),(3,1),\\ 0 & \text{otherwise,}\end{cases}$$

$$y_1 = 1/3, \qquad y_2 = 2/3,$$

and the corresponding optimal cost (lower bound) is

$$\underline{f}_Y = 4.66.$$

A feasible solution of the original problem is obtained by rounding the fractional values of y_1 and y_2 to

$$\overline{y}_1 = 1, \qquad \overline{y}_2 = 1,$$

and the associated cost is 7. Thus, we set

$$\text{UPPER} = 7,$$

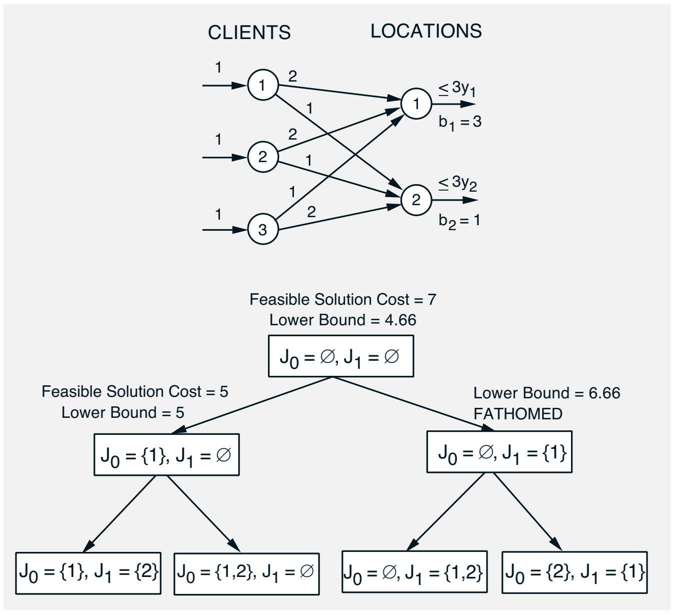


Figure 10.8: Branch-and-bound solution of a facility location problem with 3 clients and 2 locations. The facility capacities at the two locations are $c_1 = c_2 = 3$. The cost coefficients a_{ij} and b_j are shown next to the corresponding arcs. The relaxed problem for the top node $\big(J_0 = \varnothing, J_1 = \varnothing\big)$, corresponding to relaxing all the integer constraints, is solved first, obtaining the lower and upper bounds shown. Then the relaxed problem corresponding to the left node $\big(J_0 = \{1\}, J_1 = \varnothing\big)$ is solved, obtaining the lower and upper bounds shown. Finally, the relaxed problem corresponding to the right node $\big(J_0 = \varnothing, J_1 = \{1\}\big)$ is solved, obtaining a lower bound that is higher than the current value of UPPER. As a result this node can be fathomed, and its descendants need not be considered further.

and we place in OPEN the two descendants $\big(J_0 = \{1\}, J_1 = \varnothing\big)$ and $\big(J_0 = \varnothing, J_1 = \{1\}\big)$, corresponding to fixing y_1 at 0 and at 1, respectively.

We proceed with the left branch of the branch-and-bound tree, and consider the node $\big(J_0 = \{1\}, J_1 = \varnothing\big)$, corresponding to fixing y_1 as well as the corresponding flows x_{11}, x_{21}, and x_{31} to 0. The associated (relaxed) linear program is

$$\begin{aligned}
\text{minimize}\quad & x_{12} + x_{22} + 2x_{32} + y_2 \\
\text{subject to}\quad & x_{12} = 1, \qquad x_{22} = 1, \qquad x_{32} = 1, \\
& x_{12} + x_{22} + x_{32} \le 3y_2, \\
& 0 \le x_{12} \le 1, \quad 0 \le x_{22} \le 1, \quad 0 \le x_{32} \le 1, \\
& 0 \le y_2 \le 1.
\end{aligned}$$

The optimal solution (in fact the only feasible solution) of this program is

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) = (1,2), (2,2), (3,2), \\ 0 & \text{otherwise}, \end{cases}$$

$$y_2 = 1,$$

and the corresponding optimal cost (lower bound) is

$$\underline{f}_Y = 5.$$

The optimal solution of the relaxed problem is integer, and its cost, 5, is lower than the current value of UPPER, so we set

$$\text{UPPER} = 5.$$

The two descendants, $\big(J_0 = \{1\}, J_1 = \{2\}\big)$ and $\big(J_0 = \{1,2\}, J_1 = \varnothing\big)$, corresponding to fixing y_2 at 1 and at 0, respectively, are placed in OPEN.

We proceed with the right branch of the branch-and-bound tree, and consider the node $\big(J_0 = \varnothing, J_1 = \{1\}\big)$, corresponding to fixing y_1 to 1. The associated (relaxed) linear program is

$$\begin{aligned} \text{minimize} \quad & (2x_{11} + x_{12}) + (2x_{21} + x_{22}) + (x_{31} + 2x_{32}) + 3 + y_2 \\ \text{subject to} \quad & x_{11} + x_{12} = 1, \qquad x_{21} + x_{22} = 1, \qquad x_{31} + x_{32} = 1, \\ & x_{11} + x_{21} + x_{31} \le 3, \qquad x_{12} + x_{22} + x_{32} \le 3y_2, \\ & 0 \le x_{ij} \le 1, \qquad \forall\ (i,j) \in \mathcal{A}, \\ & 0 \le y_2 \le 1. \end{aligned}$$

The optimal solution of this program is

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) = (1,2), (2,2), (3,1), \\ 0 & \text{otherwise}, \end{cases}$$

$$y_2 = 2/3,$$

and the corresponding optimal cost (lower bound) is

$$\underline{f}_Y = 6.66.$$

This is larger than the current value of UPPER, so the node can be fathomed, and its two descendants are not placed in OPEN.

We conclude that one of the two descendants of the left node, $\big(J_0 = \{1\}, J_1 = \{2\}\big)$ and $\big(J_0 = \{1,2\}, J_1 = \varnothing\big)$ (the only nodes in OPEN), contains the optimal solution. We can proceed to solve the relaxed linear programs corresponding to these two nodes, and obtain the optimal solution. However, there is also a shortcut here: since these are the only two remaining nodes and the upper bound corresponding to these nodes coincides with the lower bound, we can conclude that the lower bound is equal to the optimal cost and the corresponding integer solution $(y_1 = 0, y_2 = 1)$ is optimal.

Generally, for the success of the branch-and-bound approach it is important that the lower bounds are as tight as possible, because this facilitates the fathoming of nodes, and leads to fewer restricted problem solutions. On the other hand, the tightness of the bounds strongly depends on how the problem is formulated as an integer programming problem. There may be several possible formulations, some of which are "stronger" than others in the sense that they provide better bounds within the branch-and-bound context. As an illustration, consider the following example.

Example 10.9. Facility Location – Alternative Formulation

Consider the following alternative formulation of the preceding facility location problem

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} + \sum_{j=1}^{n} b_j y_j \\
\text{subject to} \quad & \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad i = 1,\ldots,m, \\
& \sum_{\{i\mid(i,j)\in\mathcal{A}\}} x_{ij} \le c_j, \qquad j = 1,\ldots,n, \\
& x_{ij} \le y_j, \qquad \forall\ (i,j)\in\mathcal{A}, \\
& x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j)\in\mathcal{A}, \\
& y_j = 0 \text{ or } 1, \qquad j = 1,\ldots,n.
\end{aligned}$$

This formulation involves a lot more constraints, but is in fact superior to the one given earlier (cf. Example 10.8). The reason is that, if we relax the 0-1 constraints on x_{ij} and y_j, the side constraints $\sum_i x_{ij} \le y_j c_j$ of Example 10.8 are implied by the constraints $\sum_i x_{ij} \le c_j$ and $x_{ij} \le y_j$ of the present example. As a result, the lower bounds obtained by relaxing some of the 0-1 constraints are tighter in the alternative formulation just given, thereby enhancing the effectiveness of the branch-and-bound method. In fact, it can be verified that for the example of Fig. 10.8, by relaxing the 0-1 constraints in the stronger formulation of the present example, we obtain the correct optimal integer solution at the very first node of the branch-and-bound tree.

An important conclusion from the preceding example is that *it is possible to accelerate the branch-and-bound solution of a problem by adding more side constraints*. Even if these constraints do not affect the set of feasible integer solutions, they can improve the lower bounds obtained by relaxing the 0-1 constraints. Basically, when the integer constraints are relaxed, one obtains a superset of the feasible set of integer solutions, so with more side constraints, the corresponding superset becomes smaller and approximates better the true feasible set (see Fig. 10.9). It is thus very important to select a problem formulation such that when the integer constraints are relaxed, the feasible set is as small as possible.

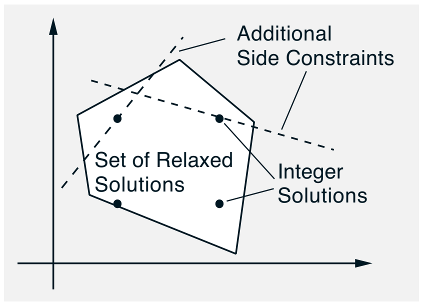


Figure 10.9: Illustration of the effect of additional side constraints. They do not affect the set of feasible integer solutions, but they reduce the set of "relaxed solutions," that is, those x that satisfy all the constraints except for the integer constraints. This results in improved lower bounds and a faster branch-and-bound solution.

We note that the subject of characterizing the feasible set of an integer programming problem, and approximating it tightly with a polyhedral set has received extensive attention. In particular, there is a lot of theory and accumulated practical knowledge on characterizing the feasible set in specific problem contexts; see the references cited at the end of the chapter. A further discussion of branch-and-bound is beyond our scope. We refer to sources on linear and combinatorial optimization, such as Zoutendijk [1976], Papadimitriou and Steiglitz [1982], Schrijver [1986], Nemhauser and Wolsey [1988], Bertsimas and Tsitsiklis [1997], Cook, Cunningham, Pulleyblank, and Schrijver [1998], which also describe many applications.

10.3 LAGRANGIAN RELAXATION

In this section, we consider an important approach for obtaining lower bounds to use in the branch-and-bound method. Let us consider the case of the network optimization problem with linear cost function, linear side constraints, and integer constraints on the arc flows:

$$\begin{aligned} &\text{minimize} \quad a'x \\ &\text{subject to} \quad \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\, i \in \mathcal{N}, \\ &\qquad\qquad\quad c_t'x \le d_t, \qquad t = 1,\ldots,r, \\ &\qquad\qquad\quad x_{ij} \in X_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}, \end{aligned}$$

where a and c_t are given vectors, d_t are given scalars, and each X_{ij} is a *finite* subset of contiguous integers (i.e., the convex hull of X_{ij} contains all the integers in X_{ij}, as for example in the cases $X_{ij} = \{0,1\}$ or $X_{ij} = \{1,2,3,4\}$). We assume that *the supplies s_i are integer*, so that if the side constraints $c_t'x \le d_t$ were not present, the problem would become a minimum cost flow problem that has integer optimal solutions, according to the theory developed in Chapter 5. Note that for this it is not necessary that the arc cost coefficients a_{ij} (the components of the vectors a) be integer.

In the Lagrangian relaxation approach, we eliminate the side constraints $c_t'x \leq d_t$ by adding to the cost function the terms $\mu_t(c_t'x - d_t)$, thereby forming the Lagrangian function

$$L(x, \mu) = a'x + \sum_{t=1}^{r} \mu_t(c_t'x - d_t),$$

where $\mu = (\mu_1, \ldots, \mu_r)$ is a vector of nonnegative scalars. Each μ_t may be viewed as a penalty per unit violation of the corresponding side constraint $c_t'x \leq d_t$, and may also be viewed as a Lagrange multiplier.

A key idea of Lagrangian relaxation is that regardless of the choice of μ, *the minimization of the Lagrangian $L(x, \mu)$ over the set of remaining constraints*

$$\tilde{F} = \{x \mid x_{ij} \in X_{ij},\ x \text{ satisfies the conservation of flow constraints}\},$$

yields a lower bound to the optimal cost of the original problem (cf. the weak duality property, discussed in Section 8.7). To see this, note that we have

$$\begin{aligned} \min_{x \in \tilde{F}} L(x, \mu) &= \min_{x \in \tilde{F}} \left\{ a'x + \sum_{t=1}^{r} \mu_t(c_t'x - d_t) \right\} \\ &\leq \min_{x \in \tilde{F},\, c_t'x - d_t \leq 0,\, t=1,\ldots,r} \left\{ a'x + \sum_{t=1}^{r} \mu_t(c_t'x - d_t) \right\} \\ &\leq \min_{x \in \tilde{F},\, c_t'x - d_t \leq 0,\, t=1,\ldots,r} a'x, \end{aligned}$$

where the first inequality follows because the minimum of the Lagrangian in the next-to-last expression is taken over a subset of $\tilde{F}$, and the last inequality follows using the nonnegativity of μ_t. The lower bound $\min_{x \in \tilde{F}} L(x, \mu)$ can in turn be used in the branch-and-bound procedure discussed earlier.

Since in the context of branch-and-bound, it is important to use as tight a lower bound as possible, we are motivated to search for an optimal lower bound through adjustment of the vector μ. To this end, we form the following dual function (cf. Section 8.7)

$$q(\mu) = \min_{x \in \tilde{F}} L(x, \mu),$$

and we consider the dual problem

$$\begin{aligned} &\text{maximize} \quad q(\mu) \\ &\text{subject to} \quad \mu_t \geq 0, \quad t = 1, \ldots, r. \end{aligned}$$

Solution of this problem yields the tightest lower bound to the optimal cost of the original problem.

Example 10.10. Constrained Shortest Path Problem

As an example of the use of Lagrangian relaxation, consider the constrained shortest path problem discussed in Example 8.6 of Section 8.4. Here, we want to find a simple forward path P from an origin node s to a destination node t that minimizes the path length

$$\sum_{(i,j)\in P} a_{ij},$$

subject to the following side constraints on P:

$$\sum_{(i,j)\in P} c_{ij}^k \le d^k, \qquad k = 1,\ldots,K.$$

As discussed in Section 8.4, we can formulate this as the following network flow problem with integer constraints and side constraints:

$$\begin{aligned}
&\text{minimize} && \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\
&\text{subject to} && \sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1 & \text{if } i = s, \\ -1 & \text{if } i = t, \\ 0 & \text{otherwise,} \end{cases} \\
&&& x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j)\in\mathcal{A}, \\
&&& \sum_{(i,j)\in\mathcal{A}} c_{ij}^k x_{ij} \le d^k, \qquad k = 1,\ldots,K.
\end{aligned} \tag{10.7}$$

Here, a path P from s to t is optimal if and only if the flow vector x defined by

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ belongs to } P, \\ 0 & \text{otherwise,} \end{cases}$$

is an optimal solution of the problem (10.7).

To apply Lagrangian relaxation, we eliminate the side constraints, and we form the corresponding Lagrangian function assigning a nonnegative multiplier μ^k to the kth constraint. Minimization of the Lagrangian now becomes a shortest path problem with respect to corrected arc lengths $\hat{a}_{ij}$ given by

$$\hat{a}_{ij} = a_{ij} + \sum_{k=1}^{K} \mu^k c_{ij}^k.$$

(We assume here that there are no negative length cycles with respect to the arc lengths $\hat{a}_{ij}$; this will be so if all the a_{ij} and c_{ij}^k are nonnegative.) We then obtain μ^* that solves the dual problem $\max_{\mu\ge 0} q(\mu)$ and we obtain a corresponding optimal cost/lower bound. We can then use μ^* to obtain a feasible solution (a path that satisfies the side constraints) as discussed in Example 8.6.

The preceding example illustrates an important advantage of Lagrangian relaxation, as applied to integer-constrained network problems: it eliminates the side constraints *simultaneously* with the integer constraints. In particular, *minimizing* $L(x,\mu)$ *over the set*

$$\tilde{F} = \{x \mid x_{ij} \in X_{ij},\ x \text{ satisfies the conservation of flow constraints}\}$$

is a (linear) minimum cost flow problem that can be solved using the methodology of Chapters 2-7: the Lagrangian $L(x,\mu)$ is linear in x and the integer constraints do not matter, and can be replaced by the interval constraints $x_{ij} \in \hat{X}_{ij}$, where $\hat{X}_{ij}$ is the convex hull of the set X_{ij}. This should be contrasted with the integer constraint relaxation approach, where we eliminate just the integer constraints, while leaving the side constraints unaffected (see the facility location problem that we solved using branch-and-bound in Example 10.8). As a result, the minimum cost flow methodology of Chapters 2-7 does not apply when there are side constraints and the integer constraint relaxation approach is used. This is the main reason for the widespread use of Lagrangian relaxation in combination with branch-and-bound.

Actually, in Lagrangian relaxation it is not mandatory to eliminate just the side constraints. One may eliminate the conservation of flow constraints, in addition to or in place of the side constraints. (The multipliers corresponding to the conservation of flow constraints should be unconstrained in the dual problem, because the conservation of flow is expressed in terms of equality constraints; cf. the discussion in Section 8.7.) One still obtains a lower bound to the optimal cost of the original problem, because of the weak duality property (cf. Section 8.7). However, the minimization of the Lagrangian is not a minimum cost flow problem anymore. Nonetheless, by choosing properly the constraints to eliminate and by taking advantage of the special structure of the problem, the minimization of the Lagrangian over the remaining set of constraints may be relatively simple. The following is an illustrative example.

Example 10.11. Traveling Salesman Problem

Consider the traveling salesman problem of Example 10.1. Here, we want to find a minimum cost tour in a complete graph where the cost of arc (i,j) is denoted a_{ij}. We formulate this as the following network problem with side constraints and 0-1 integer constraints:

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

$$\text{subject to} \quad \sum_{\substack{j=1,\ldots,N\\ j\neq i}} x_{ij} = 1, \qquad i = 1,\ldots,N, \tag{10.8}$$

$$\sum_{\substack{i=1,\ldots,N\\ i\neq j}} x_{ij} = 1, \qquad j = 1,\ldots,N, \tag{10.9}$$

$$x_{ij} = 0 \text{ or } 1, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{10.10}$$

$$\text{the subgraph with node-arc set } \big(\mathcal{N}, \{(i,j) \mid x_{ij} = 1\}\big) \text{ is connected.} \tag{10.11}$$

We may express the connectivity constraint (10.11) in several different ways, leading to different Lagrangian relaxation and branch-and-bound algorithms. One of the most successful formulations is based on the notion of a *1-tree*, which consists of a tree that spans nodes $2,\ldots,N$, *plus* two arcs that are incident to node 1. Equivalently, a 1-tree is a connected subgraph that contains a single cycle passing through node 1 (see Fig. 10.10). Note that if the conservation of flow constraints (10.8) and (10.9), and the integer constraints (10.10) are satisfied, then the connectivity constraint (10.11) is equivalent to the constraint that the subgraph $\big(\mathcal{N}, \{(i,j) \mid x_{ij} = 1\}\big)$ is a 1-tree.

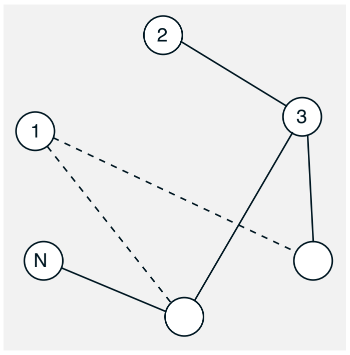


Figure 10.10: Illustration of a 1-tree. It consists of a tree that spans nodes $2,\ldots,N$, *plus* two arcs that are incident to node 1.

Let X_1 be the set of all x with $0-1$ components, and such that the subgraph $\big(\mathcal{N}, \{(i,j) \mid x_{ij} = 1\}\big)$ is a 1-tree. Let us consider a Lagrangian relaxation approach based on elimination of the conservation of flow equations. Assigning multipliers u_i and v_j to the constraints (10.8) and (10.9), respectively, the Lagrangian function is

$$L(x,u,v) = \sum_{i,j,i\neq j} (a_{ij} + u_i + v_j)x_{ij} - \sum_{i=1}^{N} u_i - \sum_{j=1}^{N} v_j.$$

The minimization of the Lagrangian is over all 1-trees, leading to the problem

$$\min_{x\in X_1} \left\{ \sum_{i,j,i\neq j} (a_{ij} + u_i + v_j)x_{ij} \right\}.$$

If we view $a_{ij} + u_i + v_j$ as a *modified cost* of arc (i,j), this minimization is quite easy. It is equivalent to obtaining a tree of minimum modified cost that spans the nodes $2, \ldots, N$, and then adding two arcs that are incident to node 1 and have minimum modified cost. The minimum cost spanning tree problem can be easily solved using the Prim-Dijkstra algorithm (see Exercise 2.30).

Unfortunately, the Lagrangian relaxation method has several weaknesses:

(a) Even if we find an optimal μ, we still have only a lower bound to the optimal cost of the original problem.

(b) The minimization of $L(x, \mu)$ over the set

$$\tilde{F} = \{x \mid x_{ij} \in X_{ij},\ x \text{ satisfies the conservation of flow constraints}\},$$

may yield an x that violates some of the side constraints $c_t'x - d_t \leq 0$, so it may be necessary to adjust this x for feasibility using some heuristic.

(c) The maximization of $q(\mu)$ over $\mu \geq 0$ may be quite nontrivial for a number of reasons, including the fact that q is typically nondifferentiable.

In what follows in this section, we will discuss the algorithmic methodology for solving the dual problem, including the subgradient and cutting plane methods, which have enjoyed a great deal of popularity. These methods have also been used widely in connection with various decomposition schemes for large-scale problems with special structure. For further discussion, we refer to the nonlinear programming literature (see for example Lasdon [1970], Auslender [1976], Shapiro [1979], Shor [1985], Poljak [1987], Hiriart-Urruty and Lemarechal [1993], and Bertsekas [1995b]).

10.3.1 Subgradients of the Dual Function

Let us consider the algorithmic solution of the dual problem

$$\begin{aligned} &\text{maximize} \ \ q(\mu) \\ &\text{subject to} \ \ \mu_t \geq 0,\ t = 1, \ldots, r. \end{aligned}$$

The dual function is

$$q(\mu) = \min_{x \in \tilde{F}} L(x, \mu),$$

where

$$\tilde{F} = \{x \mid x_{ij} \in X_{ij},\ x \text{ satisfies the conservation of flow constraints}\},$$

and $L(x, \mu)$ is the Lagrangian function

$$L(x, \mu) = a'x + \sum_{t=1}^{r} \mu_t (c_t'x - d_t).$$

Recall here that the set $\tilde{F}$ is finite, because we have assumed that each X_{ij} is a finite set of contiguous integers.

We note that for a fixed $x \in \tilde{F}$, the Lagrangian $L(x,\mu)$ is a linear function of μ. Thus, because the set $\tilde{F}$ is finite, the dual function q is the minimum of a finite number of linear functions of μ – there is one such function for each $x \in \tilde{F}$. For conceptual simplification, we may write q in the following generic form:

$$q(\mu) = \min_{i \in I}\{\alpha_i'\mu + \beta_i\}, \tag{10.12}$$

where I is some finite index set, and α_i and β_i are suitable vectors and scalars, respectively (see Fig. 10.11).

Of particular interest for our purposes are the "slopes" of q at various vectors μ, i.e., the vectors α_{i_μ}, where $i_\mu \in I$ is an index attaining the minimum of $\alpha_i'\mu + \beta_i$ over $i \in I$ [cf. Eq. (10.12)]. If i_μ is the *unique* index attaining the minimum, then q is differentiable (in fact linear) at μ, and its gradient is a_{i_μ}. If there are multiple indices i attaining the minimum, then q is nondifferentiable at μ (see Fig. 10.11). To deal with such differentiabilities, we generalize the notion of a gradient. In particular, we define a *subgradient* of q at a given $\mu \geq 0$ to be any vector g such that

$$q(\nu) \leq q(\mu) + (\nu - \mu)'g, \qquad \forall\ \nu \geq 0, \tag{10.13}$$

(see Fig. 10.11). The right-hand side of the above inequality provides a linear approximation to the dual function q using the function value $q(\mu)$ at the given μ and the corresponding subgradient g. The approximation is exact at the vector μ, and is an overestimate at other vectors ν. Some further properties of subgradients are summarized in Appendix A.

We now consider the calculation of subgradients of the dual function. For any μ, let x_μ minimize the Lagrangian $L(x,\mu)$ over $x \in \tilde{F}$,

$$x_\mu = \arg\min_{x \in \tilde{F}} L(x,\mu).$$

Let us show that *the vector $g(x_\mu)$ that has components*

$$g_t(x_\mu) = c_t'x_\mu - d_t, \qquad t = 1,\ldots,r,$$

is a subgradient of q at μ. To see this, we use the definition of L, q, and x_μ to write for all $\nu \geq 0$,

$$\begin{aligned} q(\nu) &= \min_{x \in \tilde{F}} L(x,\nu) \\ &\leq L(x_\mu,\nu) \\ &= a'x_\mu + \nu'g(x_\mu) \\ &= a'x_\mu + \mu'g(x_\mu) + (\nu-\mu)'g(x_\mu) \\ &= q(\mu) + (\nu-\mu)'g(x_\mu), \end{aligned}$$

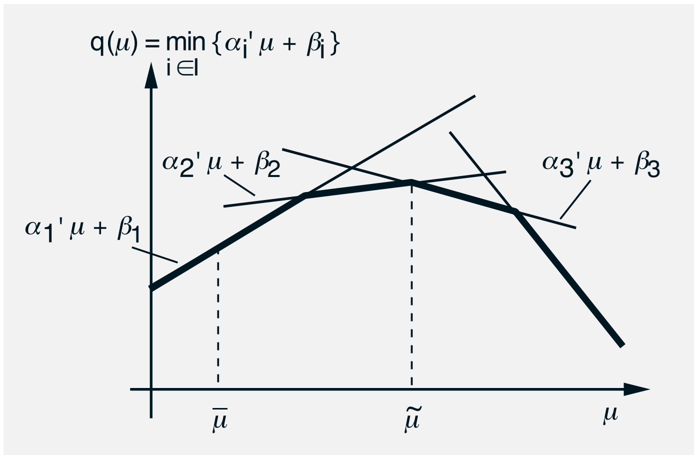

$q(\mu) = \min_{i \in I} \{\alpha_i' \mu + \beta_i\}$
$\alpha_2' \mu + \beta_2$
$\alpha_3' \mu + \beta_3$
$\alpha_1' \mu + \beta_1$
$\overline{\mu}$
$\tilde{\mu}$
μ

Figure 10.11: Illustration of the dual function q and its subgradients. The generic form of q is

$$q(\mu) = \min_{i \in I} \{\alpha_i' \mu + \beta_i\},$$

where I is some finite index set, and α_i and β_i are suitable vectors and scalars, respectively. Given μ, and an index $i_\mu \in I$ attaining the minimum in the above equation, the vector α_{i_μ} is a subgradient at μ. Furthermore, any subgradient at μ is a convex combination of vectors a_{i_μ} such that $i_\mu \in I$ and i_μ attains the minimum of $\alpha_i'\mu + \beta_i$ over $i \in I$. For example, at the vector $\overline{\mu}$ shown in the figure, there is a unique subgradient, the vector α_1. At the vector $\tilde{\mu}$ shown in the figure, the set of subgradients is the line segment connecting the vectors α_2 and α_3.

so the subgradient inequality (10.13) is satisfied for $g = g(x_\mu)$. Thus, for a given μ, *the evaluation of $q(\mu)$, which requires finding a minimizer x_μ of $L(x, \mu)$ over $\tilde{F}$, yields as a byproduct the subgradient $g(x_\mu)$.* This convenience in calculating subgradients is particularly important for the algorithms that we discuss in what follows in this section.

10.3.2 Subgradient Methods

We now turn to algorithms that use subgradients for solving the dual problem. The *subgradient method* consists of the iteration

$$\mu^{k+1} = \left[\mu^k + s^k g^k\right]^+, \tag{10.14}$$

where g^k is any subgradient of q at μ^k, s^k is a positive scalar stepsize, and $[y]^+$ is the operation that sets to 0 all the negative components of the vector y. Thus the iteration (10.14) can be written as

$$\mu_t^{k+1} = \max\{0, \mu_t^k + s^k g_t^k\}, \qquad t = 1, \ldots, r.$$

The simplest way to calculate the subgradient g^k is to find an x_{μ^k} that minimizes $L(x, \mu^k)$ over $x \in \tilde{F}$, and to set

$$g^k = g(x_{\mu^k}),$$

where for every x, $g(x)$ is the r-dimensional vector with components

$$g_t(x) = c_t' x - d_t, \qquad t = 1, \ldots, r.$$

An important fact about the subgradient method is that the new iterate may not improve the dual cost for all values of the stepsize s^k; that is, we may have

$$q\big([\mu^k + s^k g^k]^+\big) < q(\mu^k), \qquad \forall\ s^k > 0$$

(see Fig. 10.12). What makes the subgradient method work is that for sufficiently small stepsize s^k, *the distance of the current iterate to the optimal solution set is reduced*, as illustrated in Fig. 10.12, and as shown in the following proposition.

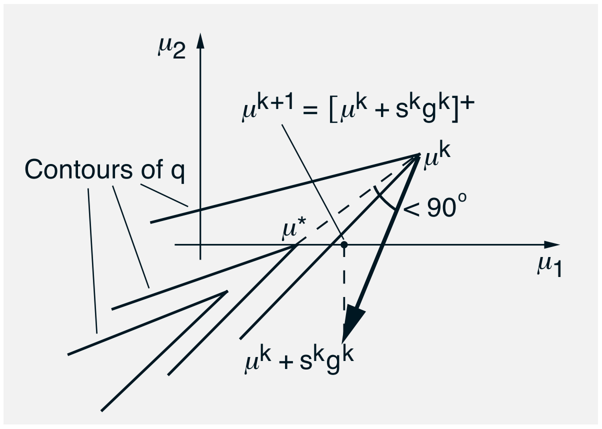


Figure 10.12: Illustration of how it may not be possible to improve the dual function by using the subgradient iteration $\mu^{k+1} = [\mu^k + s^k g^k]^+$, regardless of the value of the stepsize s^k. However, the distance to any optimal solution μ^* is reduced using a subgradient iteration with a sufficiently small stepsize. The crucial fact, which follows from the definition of a subgradient, is that the angle between the subgradient g^k and the vector $\mu^* - \mu^k$ is less than 90 degrees. As a result, for s^k small enough, the vector $\mu^k + s^k g^k$ is closer to μ^* than μ^k. Furthermore, the vector $[\mu^k + s^k g^k]^+$ is closer to μ^* than $\mu^k + s^k g^k$ is.

Proposition 10.1: If μ^k is not optimal, then for any dual optimal solution μ^*, we have

$$\|\mu^{k+1} - \mu^*\| < \|\mu^k - \mu^*\|,$$

for all stepsizes s^k such that

$$0 < s^k < \frac{2\big(q(\mu^*) - q(\mu^k)\big)}{\|g^k\|^2}. \tag{10.15}$$

Proof: We have

$$\|\mu^k + s^k g^k - \mu^*\|^2 = \|\mu^k - \mu^*\|^2 - 2s^k(\mu^* - \mu^k)'g^k + (s^k)^2\|g^k\|^2,$$

and by using the subgradient inequality (10.13),

$$(\mu^* - \mu^k)'g^k \geq q(\mu^*) - q(\mu^k).$$

Combining the last two relations, we obtain

$$\|\mu^k + s^k g^k - \mu^*\|^2 \leq \|\mu^k - \mu^*\|^2 - 2s^k\big(q(\mu^*) - q(\mu^k)\big) + (s^k)^2\|g^k\|^2.$$

We can now verify that for the range of stepsizes of Eq. (10.15) the sum of the last two terms in the above relation is negative. In particular, with a straightforward calculation, we can write this relation as

$$\|\mu^k + s^k g^k - \mu^*\|^2 \leq \|\mu^k - \mu^*\|^2 - \frac{\gamma^k(2 - \gamma^k)\big(q(\mu^*) - q(\mu^k)\big)^2}{\|g^k\|^2}, \tag{10.16}$$

where

$$\gamma^k = \frac{s^k\|g^k\|^2}{q(\mu^*) - q(\mu^k)}.$$

If the stepsize s^k satisfies Eq. (10.15), then $0 < \gamma^k < 2$, so Eq. (10.16) yields

$$\|\mu^k + s^k g^k - \mu^*\| < \|\mu^k - \mu^*\|.$$

We now observe that since $\mu^* \geq 0$, we have

$$\big\|\big[\mu^k + s^k g^k\big]^+ - \mu^*\big\| \leq \|\mu^k + s^k g^k - \mu^*\|,$$

and from the last two inequalities, we obtain $\|\mu^{k+1} - \mu^*\| < \|\mu^k - \mu^*\|$. **Q.E.D.**

The inequality (10.16) can also be used to establish convergence and rate of convergence results for the subgradient method with stepsize rules satisfying

$$0 < s^k < \frac{2\big(q(\mu^*) - q(\mu^k)\big)}{\|g^k\|^2}$$

[cf. Eq. (10.15)]. Unfortunately, however, unless we know the dual optimal cost $q(\mu^*)$, which is rare, the range of stepsizes (10.15) is unknown. In practice, a frequently used stepsize formula is

$$s^k = \frac{\alpha^k\big(q^k - q(\mu^k)\big)}{\|g^k\|^2}, \tag{10.17}$$

where q^k is an approximation to the optimal dual cost and

$$0 < \alpha^k < 2.$$

Note that we can estimate the optimal dual cost from below with the best current dual cost

$$\hat{q}^k = \max_{0 \le i \le k} q(\mu^i).$$

As an overestimate of the optimal dual cost, we can use the cost $f(\bar{x})$ of any primal feasible solution $\bar{x}$; in many circumstances, primal feasible solutions are naturally obtained in the course of the algorithm. Finally, the special structure of many problems can be exploited to yield improved bounds to the optimal dual cost.

Here are two common ways to choose α^k and q^k in the stepsize formula (10.17):

(a) q^k is the best known upper bound to the optimal dual cost at the kth iteration and α^k is a number, which is initially equal to one and is decreased by a certain factor (say, two) every few (say, five or ten) iterations. An alternative formula for α^k is

$$\alpha^k = \frac{m}{k+m},$$

where m is a positive integer.

(b) $\alpha^k = 1$ for all k and q^k is given by

$$q^k = \big(1 + \beta(k)\big)\hat{q}^k, \tag{10.18}$$

where $\hat{q}^k$ is the best current dual cost $\hat{q}^k = \max_{0 \le i \le k} q(\mu^i)$. Furthermore, $\beta(k)$ is a number greater than zero, which is increased by a certain factor if the previous iteration was a "success," that is, if it improved the best current dual cost, and is decreased by some other factor otherwise. This method requires that $\hat{q}^k > 0$. Also, if upper bounds $\tilde{q}^k$ to the optimal dual cost are available as discussed earlier, then a natural improvement to Eq. (10.18) is

$$q^k = \min\bigl\{\tilde{q}^k, \bigl(1+\beta(k)\bigr)\hat{q}^k\bigr\}.$$

For a convergence analysis of the subgradient method and its variants, we refer to the literature cited at the end of the chapter (see also Exercises 10.36-10.38). However, the convergence properties of the schemes most often preferred in practice, including the ones given above, are neither solid nor well understood. It is easy to find problems where the subgradient method works very poorly. On the other hand, the method is simple and works well for many types of problems, yielding good approximate solutions within a few tens or hundreds of iterations. Also, frequently a good primal feasible solution can be obtained using effective heuristics, even with a fairly poor dual solution.

10.3.3 Cutting Plane Methods

Consider again the dual problem

$$\begin{aligned}&\text{maximize} \;\; q(\mu)\\ &\text{subject to} \;\; \mu \ge 0.\end{aligned}$$

The cutting plane method, at the kth iteration, replaces the dual function q by a polyhedral approximation Q^k, constructed using the vectors μ^i and corresponding subgradients g^i, $i = 0, 1, \ldots, k-1$, obtained so far. It then solves the problem

$$\begin{aligned}&\text{maximize} \;\; Q^k(\mu)\\ &\text{subject to} \;\; \mu \ge 0.\end{aligned}$$

In particular, for $k = 1, 2, \ldots$, Q^k is given by

$$Q^k(\mu) = \min_{i=0,\ldots,k-1} \bigl\{q(\mu^i) + (\mu - \mu^i)'g^i\bigr\}, \tag{10.19}$$

and the kth iterate is generated by

$$\mu^k = \arg\max_{\mu \ge 0} Q^k(\mu). \tag{10.20}$$

As in the case of subgradient methods, the simplest way to calculate the subgradient g^i is to find an x_{μ^i} that minimizes $L(x, \mu^i)$ over $x \in \tilde{F}$, and to set

$$g^i = g(x^i_\mu),$$

where for every x, $g(x)$ is the r-dimensional vector with components

$$g_t(x) = c_t' x - d_t, \qquad t = 1, \ldots, r.$$

Note that the approximation $Q^k(\mu)$ is an *overestimate* of the dual function q,

$$q(\mu) \leq Q^k(\mu), \qquad \mu \geq 0, \tag{10.21}$$

since, in view of the definition of a subgradient [cf. Eq. (10.13)], each of the linear terms in the right-hand side of Eq. (10.19) is an overestimate of $q(\mu)$.

We assume that, for all k, it is possible to find a maximum μ^k of Q^k over $\mu \geq 0$. To ensure this, the method has to be suitably initialized; for example by selecting a sufficiently large number of vectors μ, and by computing corresponding subgradients, to form an initial approximation that is bounded from above over the set $\{\mu \mid \mu \geq 0\}$. Thus, in this variant, we start the method at some iteration $\overline{k} > 0$, with the vectors $\mu^0, \ldots, \mu^{\overline{k}-1}$ suitably selected so that $Q^{\overline{k}}(\mu)$ is bounded from above over $\mu \geq 0$. Alternatively, we may maximize Q^k over a suitable bounded polyhedral set that is known to contain an optimal dual solution, instead of maximizing over $\mu \geq 0$. We note that given the iterate μ^k, the method produces both the exact and the approximate dual values $q(\mu^k)$ and $Q^k(\mu^k)$. It can be seen, using Eqs. (10.20) and (10.21), that the optimal dual cost is bracketed between these two values:

$$q(\mu^k) \leq \max_{\mu \geq 0} q(\mu) \leq Q^k(\mu^k). \tag{10.22}$$

Thus, in particular, the equality

$$q(\mu^k) = Q^k(\mu^k) \tag{10.23}$$

guarantees the optimality of the vector μ^k. It turns out that because the dual function is piecewise linear, and consequently only a finite number of subgradients can be generated, the optimality criterion (10.23) is satisfied in a finite number of iterations, and the method terminates. This is shown in the following proposition and is illustrated in Fig. 10.13.

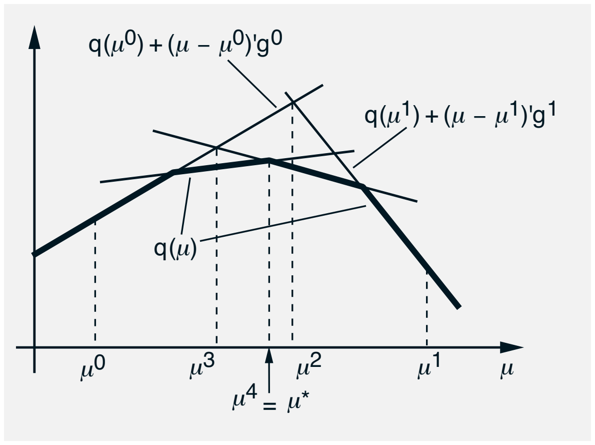
Proposition 10.2: The cutting plane method terminates finitely; that is, for some k, μ^k is a dual optimal solution and the termination criterion (10.23) is satisfied.

Proof: For notational convenience, let us write the dual function in the polyhedral form

$$q(\mu) = \min_{i \in I}\{\alpha_i'\mu + \beta_i\},$$

where I is some finite index set and α_i, β_i, $i \in I$, are suitable vectors and scalars, respectively. Let i^k be an index attaining the minimum in the equation

$$q(\mu^k) = \min_{i \in I}\{\alpha_i'\mu^k + \beta_i\},$$



$q(\mu^0) + (\mu - \mu^0)'g^0$
$q(\mu^1) + (\mu - \mu^1)'g^1$
$q(\mu)$
μ^0
μ^3
μ^2
μ^1
μ
$\mu^4 = \mu^*$

Figure 10.13: Illustration of the cutting plane method. With each new iterate μ^i, a new hyperplane $q(\mu^i)+(\mu-\mu^i)'g^i$ is added to the polyhedral approximation of the dual function. The method converges finitely, since if μ^k is not optimal, a new cutting plane will be added at the corresponding iteration, and there can be only a finite number of cutting planes.

so that α_{i^k} is a subgradient at μ^k. If the termination criterion (10.23) is not satisfied at μ^k, we must have

$$\alpha'_{i^k}\mu^k + \beta_{i^k} = q(\mu^k) < Q^k(\mu^k).$$

Since

$$Q^k(\mu^k) = \min_{0\le m\le k-1}\{\alpha'_{i^m}\mu^k + \beta_{i^m}\},$$

it follows that the pair $(\alpha_{i^k}, \beta_{i^k})$ is not equal to any of the preceding pairs $(\alpha_{i^0}, \beta_{i^0}), \ldots, (\alpha_{i^{k-1}}, \beta_{i^{k-1}})$. Since the index set I is finite, it follows that there can be only a finite number of iterations for which the termination criterion (10.23) is not satisfied. **Q.E.D.**

Despite its finite convergence property, the cutting plane method may converge slowly, and in practice one may have to stop it short of finding an optimal solution [the error bounds (10.22) may be used for this purpose]. An additional drawback of the method is that it can take large steps away from the optimum even when it is close to (or even at) the optimum. This phenomenon is referred to as *instability*, and has another undesirable effect, namely, that μ^{k-1} may not be a good starting point for the algorithm that minimizes $Q^k(\mu)$. A way to limit the effects of this phenomenon is to add to the polyhedral function approximation a quadratic term that penalizes large deviations from the current point. In this method, μ^k is obtained as

$$\mu^k = \arg\max_{\mu\ge 0}\left\{Q^k(\mu) - \frac{1}{2c^k}\|\mu - \mu^{k-1}\|^2\right\},$$

where $\{c^k\}$ is a positive nondecreasing scalar parameter sequence. This is known as the *proximal cutting plane algorithm*, and is related to the proximal minimization method discussed in Section 8.8.5. It can be shown that this variant of the cutting plane method also terminates finitely thanks to the polyhedral nature of q.

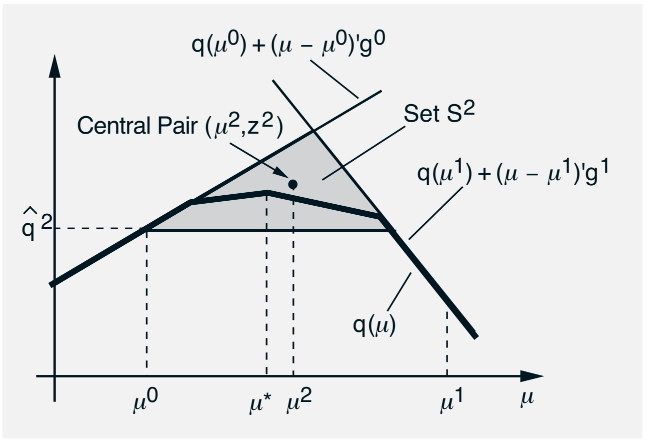
Another interesting variant of the cutting plane method, known as the *central cutting plane method*, maintains the polyhedral approximation $Q^k(\mu)$ to the dual function q, but generates the next vector μ^k by using a somewhat different mechanism. In particular, instead of maximizing Q^k, the method obtains μ^k by finding a "central pair" (μ^k, z^k) within the subset

$$S^k = \big\{(\mu, z) \mid \mu \geq 0,\ \hat{q}^k \leq q(\mu),\ \hat{q}^k \leq z \leq Q^k(\mu)\big\},$$

where $\hat{q}^k$ is the best lower bound to the optimal dual cost that has been found so far,

$$\hat{q}^k = \max_{i=0,\ldots,k-1} q(\mu^i).$$

The set S^k is illustrated in Fig. 10.14.



$q(\mu^0) + (\mu - \mu^0)'g^0$
Central Pair (μ^2, z^2)
Set S^2
$q(\mu^1) + (\mu - \mu^1)'g^1$
$\hat{q}^2$
$q(\mu)$
μ^0
μ^*
μ^2
μ^1
μ

Figure 10.14: Illustration of the set

$$S^k = \big\{(\mu, z) \mid \mu \geq 0,\ \hat{q}^k \leq q(\mu),\ \hat{q}^k \leq z \leq Q^k(\mu)\big\},$$

for $k = 2$, in the central cutting plane method.

There are several possible methods for finding the central pair (μ^k, z^k). Roughly, the idea is that the central pair should be "somewhere in the middle" of S^k. For example, consider the case where S^k is polyhedral with nonempty interior. Then (μ^k, z^k) could be the *analytic center* of S^k, where for any polyhedron

$$P = \{y \mid a_p' y \leq c_p,\ p = 1, \ldots, m\}$$

with nonempty interior, its analytic center is the unique maximizer of $\sum_{p=1}^{m} \ln(c_p - a_p' y)$ over $y \in P$. Another possibility is the *ball center* of S, that is, the center of the largest inscribed sphere in S. Assuming that the polyhedron P given above has nonempty interior, its ball center can be obtained by solving the following problem with optimization variables (y, σ):

$$\begin{aligned} &\text{maximize} \ \ \sigma \\ &\text{subject to} \ \ a_p'(y+d) \le c_p, \quad \forall \ \|d\| \le \sigma, \ p = 1, \ldots, m. \end{aligned}$$

It can be seen that this problem is equivalent to the linear program

$$\begin{aligned} &\text{maximize} \ \ \sigma \\ &\text{subject to} \ \ a_p' y + \|a_p\|\sigma \le c_p, \quad p = 1, \ldots, m. \end{aligned}$$

While the central cutting plane methods are not guaranteed to terminate finitely, their convergence properties are satisfactory. Furthermore, the methods have benefited from advances in the implementation of interior point methods; see the references cited at the end of the chapter.

10.3.4 Decomposition and Multicommodity Flows

Lagrangian relaxation is particularly convenient when by eliminating the side constraints, we obtain a network problem that decomposes into several independent subproblems. A typical example arises in multicommodity flow problems where we want to minimize

$$\sum_{m=1}^{M} \sum_{(i,j)\in\mathcal{A}} a_{ij}(m) x_{ij}(m) \tag{10.24}$$

subject to the conservation of flow constraints

$$\sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij}(m) - \sum_{\{j \mid (j,i)\in\mathcal{A}\}} x_{ji}(m) = s_i(m), \qquad \forall \ i \in \mathcal{N}, \ m = 1, \ldots, M, \tag{10.25}$$

the set constraints

$$x_{ij}(m) \in X_{ij}(m), \qquad \forall \ m = 1, \ldots, M, \ (i,j) \in \mathcal{A}, \tag{10.26}$$

and the side constraints

$$\sum_{m=1}^{M} A(m) x(m) \le b. \tag{10.27}$$

Here $s_i(m)$ are given supply integers for the mth commodity, $A(m)$ are given matrices, b is a given vector, and $x(m)$ is the flow vector of the mth commodity, with components $x_{ij}(m)$, $(i,j) \in \mathcal{A}$. Furthermore, each $X_{ij}(m)$ is a *finite* subset of contiguous integers.

The dual function is obtained by relaxing the side constraints (10.27), and by minimizing the corresponding Lagrangian function. This minimization separates into m independent minimizations, one per commodity:

$$q(\mu) = -\mu' b + \sum_{m=1}^{M} \min_{x(m)\in F(m)} \big(a(m) + A(m)'\mu\big)' x(m), \tag{10.28}$$

where $a(m)$ is the vector with components $a_{ij}(m)$, $(i,j) \in \mathcal{A}$, and

$$F(m) = \big\{x(m) \text{ satisfying Eq. (10.25)} \mid x_{ij}(m) \in X_{ij}(m),\ \forall\ (i,j) \in \mathcal{A}\big\}.$$

An important observation here is that each of the minimization subproblems above is a minimum cost flow problem that can be solved using the methods of Chapters 2-7. Furthermore, if $x^\mu(m)$ solves the mth subproblem, the vector

$$g^\mu = \sum_{m=1}^{M} A(m) x^\mu(m) - b \tag{10.29}$$

is a subgradient of q at μ.

Let us now discuss the computational solution of the dual problem $\max_{\mu\geq 0} q(\mu)$. The application of the subgradient method is straightforward, so we concentrate on the cutting plane method, which leads to a method known as *Dantzig-Wolfe decomposition.* This method consists of the iteration

$$\mu^k = \arg\max_{\mu\geq 0} Q^k(\mu),$$

where $Q^k(\mu)$ is the piecewise linear approximation of the dual function based on the preceding function values $q(\mu^0), \ldots, q(\mu^{k-1})$, and the corresponding subgradients $g^0, \ldots, g^{k-1}$:

$$Q^k(\mu) = \min\big\{q(\mu^0) + (\mu-\mu^0)'g^0, \ldots, q(\mu^{k-1}) + (\mu-\mu^{k-1})'g^{k-1}\big\}.$$

Consider now the cutting plane subproblem $\max_{\mu\geq 0} Q^k(\mu)$. By introducing an auxiliary variable v, we can write this problem as

$$\begin{aligned} &\text{maximize} && v \\ &\text{subject to} && q(\mu^i) + (\mu-\mu^i)'g^i \geq v, \quad i = 0, \ldots, k-1, \\ &&& \mu \geq 0. \end{aligned} \tag{10.30}$$

This is a linear program in the variables v and μ. We can form its dual problem by assigning a Lagrange multiplier ξ^i to each of the constraints

$q(\mu^i) + (\mu - \mu^i)'g^i \geq v$. After some calculation, this dual problem can be verified to have the form

$$\begin{aligned} &\text{minimize} \quad \sum_{i=0}^{k-1} \xi^i \big(q(\mu^i) - \mu^{i\prime} g^i\big) \\ &\text{subject to} \quad \sum_{i=0}^{k-1} \xi^i = 1, \qquad \sum_{i=0}^{k-1} \xi^i g^i \leq 0, \\ &\qquad\qquad \xi^i \geq 0, \quad i = 0, \ldots, k-1. \end{aligned} \tag{10.31}$$

Using Eqs. (10.28) and (10.29), we have

$$q(\mu^i) = -\mu^{i\prime} b + \sum_{m=1}^{M} \big(a(m) + A(m)'\mu^i\big)' x^{\mu^i}(m),$$

$$g^i = \sum_{m=1}^{M} A(m) x^{\mu^i}(m) - b,$$

so the problem (10.31) can be written as

$$\begin{aligned} &\text{minimize} \quad \sum_{m=1}^{M} a(m)' \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \\ &\text{subject to} \quad \sum_{i=0}^{k-1} \xi^i = 1, \qquad \sum_{m=1}^{M} A(m) \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \leq b, \\ &\qquad\qquad \xi^i \geq 0, \quad i = 0, \ldots, k-1. \end{aligned} \tag{10.32}$$

The preceding problem is called the *master problem*. It is the dual of the cutting plane subproblem $\max_{\mu \geq 0} Q^k(\mu)$, which in turn approximates the dual problem $\max_{\mu \geq 0} q(\mu)$; in short, it is the *dual of the approximate dual*. We may view this problem as an approximate version of the primal problem where the commodity flow vectors $x(m)$ are constrained to lie in the convex hull of the already generated vectors $x^{\mu^i}(m)$, $i = 0, \ldots, k-1$, rather than in their original constraint set. It can be shown, using linear programming theory, that if the problem (10.30) has an optimal solution [i.e., enough vectors μ^i are available so that the maximum of $Q^k(\mu)$ over $\mu \geq 0$ is attained], then the master problem also has an optimal solution.

Suppose now that we solve the master problem (10.32) using a method that yields a Lagrange multiplier vector, call it μ^k, corresponding to the constraints

$$\sum_{m=1}^{M} A(m) \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \leq b.$$

(Standard linear programming methods, such as the simplex method, can be used for this.) Then, the dual of the master problem [which is the cutting plane subproblem $\max_{\mu \geq 0} Q^k(\mu)$] is solved by the Lagrange multiplier μ^k. Therefore, μ^k is the next iterate of the cutting plane method.

We can now piece together the typical cutting plane iteration.

Cutting Plane – Dantzig-Wolfe Decomposition Iteration

Step 1: Given $\mu^0, \ldots, \mu^{k-1}$, and the commodity flow vectors $x^{\mu^i}(m)$ for $m = 1, \ldots, M$ and $i = 0, \ldots, k-1$, solve the master problem

$$\begin{aligned} &\text{minimize} \quad \sum_{m=1}^{M} a(m)' \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \\ &\text{subject to} \quad \sum_{i=0}^{k-1} \xi^i = 1, \qquad \sum_{m=1}^{M} A(m) \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \leq b, \\ &\qquad\qquad\quad \xi^i \geq 0, \quad i = 0, \ldots, k-1. \end{aligned}$$

and obtain μ^k, which is a Lagrange multiplier vector of the constraints

$$\sum_{m=1}^{M} A(m) \sum_{i=0}^{k-1} \xi^i x^{\mu^i}(m) \leq b.$$

Step 2: For each $m = 1, \ldots, M$, obtain a solution $x^{\mu^k}(m)$ of the minimum cost flow problem

$$\min_{x(m) \in F(m)} \big(a(m) + A(m)'\mu^k\big)' x(m).$$

Step 3: Use $x^{\mu^k}(m)$ to modify the master problem by adding one more variable ξ^k and go to the next iteration.

Decomposition by Right-Hand Side Allocation

There is an alternative decomposition approach for solving the multicommodity flow problem with side constraints (10.24)-(10.27). In this approach, we introduce auxiliary variables $y(m)$, $m = 1, \ldots, M$, and we write the problem as

$$\begin{aligned}&\text{minimize} \quad \sum_{m=1}^{M} a(m)'x(m)\\ &\text{subject to} \;\; x(m) \in F(m), \qquad m = 1, \ldots, M,\\ &\qquad\qquad \sum_{m=1}^{M} y(m) = b, \qquad A(m)x(m) \le y(m), \quad m = 1, \ldots, M.\end{aligned}$$

Equivalently, we can write the problem as

$$\begin{aligned}&\text{minimize} \quad \sum_{m=1}^{M} \min_{\substack{x(m)\in F(m)\\ A(m)x(m)=y(m)}} a(m)'x(m)\\ &\text{subject to} \quad \sum_{m=1}^{M} y(m) = b, \qquad y(m) \in Y(m), \quad m = 1, \ldots, M,\end{aligned} \tag{10.33}$$

where $Y(m)$ is the set of all vectors $y(m)$ for which the inner minimization problem

$$\begin{aligned}&\text{minimize} \;\; a(m)'x(m)\\ &\text{subject to} \;\; x(m) \in F(m), \qquad A(m)x(m) \le y(m)\end{aligned} \tag{10.34}$$

has at least one feasible solution.

Let us define

$$p_m\big(y(m)\big) = \min_{\substack{x(m)\in F(m)\\ A(m)x(m)\le y(m)}} a(m)'x(m).$$

Then, problem (10.33) can be written as

$$\begin{aligned}&\text{minimize} \quad \sum_{m=1}^{M} p_m\big(y(m)\big)\\ &\text{subject to} \quad \sum_{m=1}^{M} y(m) = b, \qquad y(m) \in Y(m), \quad m = 1, \ldots, M.\end{aligned}$$

This problem, called the *master problem*, may be solved with nondifferentiable optimization methods, and in particular with the subgradient and the cutting plane methods. Note, however, that the commodity problems (10.34) involve the side constraints $A(m)x(m) \le y(m)$, and need not be of the minimum cost flow type, except in special cases. We refer to the literature cited at the end of the chapter for further details.

10.4 LOCAL SEARCH METHODS

Local search methods are a broad and important class of heuristics for discrete optimization. They apply to the general problem of minimizing a function $f(x)$ over a *finite* set F of (feasible) solutions. In principle, one may solve the problem by *global enumeration* of the entire set F of solutions (this is what branch-and-bound does). A local search method tries to economize on computation by using *local enumeration*, based on the notion of a *neighborhood* $N(x)$ of a solution x, which is a (usually very small) subset of F, containing solutions that are "close" to x in some sense.

In particular, given a solution x, the method selects among the solutions in the neighborhood $N(x)$ a successor solution $\overline{x}$, according to some rule. The process is then repeated with $\overline{x}$ replacing x (or stops when some termination criterion is met). Thus a local search method is characterized by:

(a) The method for choosing a starting solution.

(b) The definition of the neighborhood $N(x)$ of a solution x.

(c) The rule for selecting a successor solution from within $N(x)$.

(d) The termination criterion.

For an example of a local search method, consider the k-OPT heuristic for the traveling salesman problem that we discussed in Example 10.1. Here the starting tour is obtained by using some method, based for example on subtour elimination or a minimum weight spanning tree, as discussed in Example 10.1. The neighborhood of a tour T is defined as the set $N(T)$ of all tours obtained from T by exchanging k arcs that belong to T with another k arcs that do not belong to T. The rule for selecting a successor tour is based on cost improvement; that is, the tour selected from $N(T)$ has minimum cost over all tours in $N(T)$ that have smaller cost than T. Finally, the algorithm terminates when no tour in $N(T)$ has smaller cost than T. Another example of a local search method is provided by the Esau-Williams heuristic of Fig. 10.5.

The definition of a neighborhood often involves intricate calculations and suboptimizations that aim to bring to consideration promising neighbors. Here is an example, due to Kernighan and Lin [1970]:

Example 10.12. (Uniform Graph Partitioning)

Consider a graph $(\mathcal{N}, \mathcal{A})$ with $2n$ nodes, and a cost a_{ij} for each arc (i, j). We want to find a partition of $\mathcal{N}$ into two subsets $\mathcal{N}_1$ and $\mathcal{N}_2$, each with n nodes, so that the total cost of the arcs connecting $\mathcal{N}_1$ and $\mathcal{N}_2$,

$$\sum_{(i,j),\, i\in\mathcal{N}_1,\, j\in\mathcal{N}_2} a_{ij} + \sum_{(i,j),\, i\in\mathcal{N}_2,\, j\in\mathcal{N}_1} a_{ij},$$

is minimized.

Here a natural neighborhood of a partition $(\mathcal{N}_1, \mathcal{N}_2)$ is the *k-exchange neighborhood*. This is the set of all partitions obtained by selecting a fixed number k of pairs of nodes (i, j) with $i \in \mathcal{N}_1$ and $j \in \mathcal{N}_2$, and interchanging them, that is, moving i into $\mathcal{N}_2$ and j into $\mathcal{N}_1$. The corresponding local search algorithm moves from a given solution to its minimum cost neighbor, and terminates when no neighbor with smaller cost can be obtained. Unfortunately, the amount of work needed to generate a k-exchange neighborhood increases exponentially with k [there are $\binom{m}{k}$ different ways to select k objects out of m]. One may thus consider a *variable depth neighborhood* that involves multiple successive k-exchanges with small k. As an example, for $k = 1$ we obtain the following algorithm:

Given the starting partition $(\mathcal{N}_1, \mathcal{N}_2)$, consider all pairs (i, j) with $i \in \mathcal{N}_1$ and $j \in \mathcal{N}_2$, and let $c(i, j)$ be the cost change that results from moving i into $\mathcal{N}_2$ and j into $\mathcal{N}_1$. If $(\bar{i}, \bar{j})$ is the pair that minimizes $c(i, j)$, move $\bar{i}$ into $\mathcal{N}_1$ and $\bar{j}$ into $\mathcal{N}_2$, and let $c_1 = c(\bar{i}, \bar{j})$. Repeat this process a fixed number M of times, obtaining a sequence $c_2, c_3, \ldots, c_M$ of minimal cost changes resulting from the sequence of exchanges. Then find

$$\overline{m} = \arg \min_{m=1,\ldots,M} \sum_{l=1}^{m} c_l,$$

and accept as the next partition the one involving the first $\overline{m}$ exchanges.

This type of algorithm avoids the exponential running time of k-exchange neighborhoods, while still considering neighbors differing by as many as M node pairs.

While the definition of neighborhood is often problem-dependent, some general classes of procedures for generating neighborhoods have been developed. One such class is *genetic algorithms*, to be discussed shortly. In some cases, neighborhoods are dynamically changing, and they may depend not only on the current solution, but also on several past solutions. The method of *tabu search*, to be discussed shortly, falls in this category.

The criterion for selecting a solution from within a neighborhood is usually the cost of the solution, but sometimes a more complex criterion based on various problem characteristics and/or constraint violation considerations is adopted. An important possibility, which is the basis for the *simulated annealing method*, to be discussed shortly, is to use a random mechanism for selecting the successor solution within a neighborhood.

Finally, regarding the termination criterion, many local search methods are *cost improving*, and stop when an improved solution cannot be found within the current neighborhood. This means that these methods stop at a *local minimum*, that is, a solution that is no worse than all other solutions within its neighborhood. Unfortunately, for many problems, a local minimum may be far from optimal, particularly if the neighborhood used is relatively small. Thus, for a cost improving method, there is a basic tradeoff between using a large neighborhood to diminish the difficulty with local minima, and paying the cost of increased computation per iteration. Note that there is an important advantage to a cost improving method: *it can never repeat the same solution*, so that in view of the finiteness of the feasible set F, it will always terminate with a local minimum.

An alternative type of neighbor selection and termination criterion, used by simulated annealing and tabu search, is to allow successor solutions to have worse cost than their predecessors, but to also provide mechanisms that ensure the future generation of improved solutions with substantial likelihood. The advantage of accepting solutions of worse cost is that stopping at a local minimum becomes less of a difficulty. For example, the method of simulated annealing, cannot be trapped at a local minimum, as we will see shortly. Unfortunately, methods that do not enforce cost improvement run the danger of cycling through repetition of the same solution. It is therefore essential in these methods to provide a mechanism by virtue of which cycling is either precluded, or becomes highly unlikely.

As a final remark, we note an important advantage of local search methods. While they offer no solid guarantee of finding an optimal or near-optimal solution, they offer the promise of substantial improvement over any heuristic that can be used to generate the starting solution. Unfortunately, however, one can seldom be sure that this promise will be fulfilled in a given practical problem.

10.4.1 Genetic Algorithms

These are local search methods where the neighborhood generation mechanism is inspired by real-life processes of genetics and evolution. In particular, the current solution is modified by "splicing" and "mutation" to obtain neighboring solutions. A typical example is provided by problems of scheduling, such as the traveling salesman problem. The neighborhood of a schedule T may be a collection of other schedules obtained by modifying some contiguous portion of T in some way, while keeping the remainder of the schedule T intact. Alternatively, the neighborhood of a schedule may be obtained by interchanging the position of a few tasks, as in the k-OPT traveling salesman heuristic.

In a variation of this approach, a pool of solutions may be maintained. Some of these solutions may be modified, while some pairs of these solutions may be combined to form new solutions. These solutions, are added to the pool if some criterion, typically based on cost improvement, is met, and some of the solutions of the existing pool may be dropped. In this way, it is argued, the pool is "evolving" in a Darwinian way through a "survival of the fittest" process.

A specific example implementation of this approach operates in phases. At the beginning of a phase, we have a *population* X consisting of n feasible solutions $x_1, \ldots, x_n$. The phase proceeds as follows:

Typical Phase of a Genetic Algorithm

Step 1: (Local Search) Starting from each solution x_i of the current population X, apply a local search algorithm up to obtaining a local minimum $\overline{x}_i$. Let $\overline{X} = \{\overline{x}_1, \dots, \overline{x}_n\}$.

Step 2: (Mutation) Select at random a subset of elements of $\overline{X}$, and modify each element according to some (problem dependent) mechanism, to obtain another feasible solution.

Step 3: (Recombination) Select at random a subset of pairs of elements of $\overline{X}$, and produce from each pair a feasible solution according to some (problem dependent) mechanism.

Step 4: (Survivor Selection) Let $\tilde{X}$ be the set of feasible solutions obtained from the mutation and recombination Steps 3 and 4. Out of the population $\overline{X} \cup \tilde{X}$, select a subset of n elements according to some criterion. Use this subset to start the next phase.

Mutation allows speculative variations of the local minima at hand, while recombination (also called *crossover*) aims to combine attributes of a pair of local minima. The processes of mutation and recombination are usually performed with the aid of some data structure that is used to represent a solution, such as for example a string of bits. There is a very large number of variants of genetic algorithm approaches. Typically, these approaches are problem-dependent and require a lot of trial-and-error. However, genetic algorithms are quite easy to implement, and have achieved considerable popularity. We refer to the literature cited at the end of the chapter for more details.

10.4.2 Tabu Search

Tabu search aims to avoid getting trapped at a poor local minimum, by accepting on occasion a worse or even infeasible solution from within the current neighborhood. Since cost improvement is not enforced, tabu search runs the danger of cycling, i.e., repeating the same sequence of solutions indefinitely. To alleviate this problem, tabu search keeps track of recently obtained solutions in a "forbidden" (tabu) list. Solutions in the tabu list cannot be regenerated, thereby avoiding cycling, at least in the short run. In a more sophisticated variation of this strategy, the tabu list contains attributes of recently obtained solutions rather than the solutions themselves. Solutions with attributes in the tabu list are forbidden from being generated (except under particularly favorable circumstances, under which the tabu list is overridden).

Tabu search is also based on an elaborate web of implementation heuristics that have been developed through experience with a large number of practical problems. These heuristics regulate the size of the current neighborhood, the criterion of selecting a new solution from the current neighborhood, the criterion for termination, etc. These heuristics may also involve selective memory storage of previously generated solutions or their attributes, penalization of the constraints with (possibly time-varying) penalty parameters, and multiple tabu lists. We refer to the literature cited at the end of the chapter for further details.

10.4.3 Simulated Annealing

Simulated annealing is similar to tabu search in that it occasionally allows solutions of inferior cost to be generated. It differs from tabu search in the manner in which it avoids cycling. Instead of checking deterministically the preceding solutions for cycling, it simply randomizes its selection of the next solution. In doing so, it not only avoids cycling, but also provides some theoretical guarantee of escaping from local minima and eventually finding a *global* minimum.

Being able to find a global minimum is not really exciting in itself. For example, under fairly general conditions, one can do so by using unsophisticated *random search* methods, such as for example a method where feasible solutions are sampled at random. However, simulated annealing randomizes the choice of the successor solution from within the current neighborhood in a way that gives preference to solutions of smaller cost, and in doing so, it aims to find a global minimum faster than simple-minded random search methods.

In particular, given a solution x, we select by random sampling a candidate solution $\overline{x}$ from the neighborhood $N(x)$. The sampling probabilities are positive for all members of $N(x)$, but are otherwise unspecified. The solution $\overline{x}$ is accepted if it is cost improving, that is

$$f(\overline{x}) < f(x).$$

Otherwise, $\overline{x}$ is accepted with probability

$$e^{-\big(f(\overline{x})-f(x)\big)/T},$$

where T is some positive constant, and is rejected with the complementary probability.

The constant T regulates the likelihood of accepting solutions of worse cost. It is called the *temperature* of the process (the name is inspired by a certain physical analogy that will not be discussed here). The likelihood of accepting a solution $\overline{x}$ of worse cost than x decreases as its cost increases. Furthermore, when T is large (or small), the probability of accepting a worse solution is close to 1 (or close to 0, respectively). In practice, it is typical to start with a large T, allowing a better chance of escaping from local minima, and then to reduce T gradually to enhance the selectivity of the method towards improved solutions.

Contrary to genetic algorithms and tabu search, which offer no general theoretical guarantees of good performance, simulated annealing is supported by solid theory. In particular, under fairly general conditions, it can be shown that a global minimum will be eventually visited (with probability 1), and that with gradual reduction of the temperature T, the search process will be confined with high likelihood to solutions that are globally optimal.

For an illustrative analysis, assume that T is kept constant and let p_{xy} be the probability that when the current solution is x, the next solution sampled is y. Consider the special case where $p_{xy} = p_{yx}$ for all feasible solutions x and y, and assume that the Markov chain defined by the probabilities p_{xy} is *irreducible*, in the sense that there is positive probability to go from any x to any y, with one or more samples. Then it can be shown (see Exercise 10.20) that the steady-state probability of a solution $\overline{x}$ is

$$\frac{e^{-f(\overline{x})/T}}{\sum_{x \in F} e^{-f(x)/T}}.$$

Essentially, this says that for very small T and far into the future, the current solution is almost always optimal.

When the condition $p_{xy} = p_{yx}$ does not hold, one cannot obtain a closed-form expression for the steady-state probabilities of the various solutions. However, as long as the underlying Markov chain is irreducible, the behavior is qualitatively similar: the steady-state probability of nonoptimal solutions diminishes to 0 as T approaches 0. There is also related analysis for the case where the temperature parameter T is time-varying and converges to 0; see the references cited at the end of the chapter.

The results outlined above should be viewed with a grain of salt. In practice, *speed of convergence* is as important as eventual convergence to the optimum, and solving a given problem by simulated annealing can be very slow. A nice aspect of the method is that it depends very little on the structure of the problem being solved, and this enhances its value for relatively unstructured problems that are not well-understood. For other problems, where there exists a lot of accumulated insight and experience, simulated annealing is usually inferior to other local search approaches.

10.5 ROLLOUT ALGORITHMS

The branch-and-bound algorithm is guaranteed to find an optimal flow vector, but it may require the solution of a very large number of subproblems. Basically, the algorithm amounts to an exhaustive search of the entire branch-and-bound tree. An alternative is to consider faster methods that are based on intelligent but nonexhaustive search of the tree. In this section, we develop one such method, the *rollout algorithm*, which, in its simplest version, sequentially constructs a suboptimal flow vector by fixing the arc flows, a few arcs at a time. The rollout algorithm can be combined with most heuristics, including the local search methods of the preceding section, and is capable of magnifying their effectiveness.

Let us consider the minimization of a function f of a flow vector x over a feasible set F, and let us assume that F is *finite* (presumably because of some integer constraints on the arc flows). Define a *partial solution* to be a collection of arc flows $\{x_{ij} \mid (i,j) \in S\}$, corresponding to some proper subset of arcs $S \subset \mathcal{A}$. Such a collection is distinguished from a flow vector ($S = \mathcal{A}$), which is also referred to as a *complete solution.*

The rollout algorithm generates a sequence of partial solutions, culminating with a complete solution. For this purpose, it employs some problem-dependent heuristic algorithm, called the *base heuristic*. This algorithm, given a partial solution

$$P = \big\{x_{ij} \mid (i,j) \in S\big\},$$

produces a *complementary solution*

$$\overline{P} = \big\{x_{ij} \mid (i,j) \notin S\big\},$$

and a corresponding (complete) flow vector

$$x = \big\{x_{ij} \mid (i,j) \in \mathcal{A}\big\} = P \cup \overline{P}.$$

The cost of this flow vector is denoted by

$$H(P) = \begin{cases} f(x) & \text{if } x \in F, \\ \infty & \text{otherwise,} \end{cases}$$

and is called the *heuristic cost of the partial solution P*. If P is a complete solution, which is feasible, i.e., a flow vector $x \in F$, by convention the heuristic cost of P is the true cost $f(x)$. There are no restrictions on the nature of the base heuristic; a typical example is an integer rounding heuristic applied to the solution of some related linear or convex network problem, which may be obtained by relaxing/neglecting the integer constraints.

The rollout algorithm starts with some partial solution, or with the empty set of arcs, $S = \emptyset$. It enlarges a partial solution iteratively, with a few arc flows at a time. The algorithm terminates when a complete solution is obtained. At the start of the typical iteration, we have a current partial solution

$$P = \big\{x_{ij} \mid (i,j) \in S\big\},$$

and at the end of the iteration, we augment this solution with some more arc flows. The steps of the iteration are as follows:

Iteration of the Rollout Algorithm

Step 1: Select a subset T of arcs that are not in S according to some criterion. (The arc selection method is usually based on some heuristic preliminary optimization, and is problem-dependent.)

Step 2: Consider the collection F_T of all possible values of the arc flows $y = \{y_{ij} \mid (i,j) \in T\}$, and apply the base heuristic to compute the heuristic cost $H(P_y^+)$ of the augmented partial solution

$$P_y^+ = \big\{\{x_{ij} \mid (i,j) \in S\}, \{y_{ij} \mid (i,j) \in T\}\big\}$$

for each $y \in F_T$.

Step 3: Choose from the set F_T the arc flows $\overline{y} = \{\overline{y}_{ij} \mid (i,j) \in T\}$ that minimize the heuristic cost $H(P_y^+)$; that is, find

$$\overline{y} = \arg \min_{y \in F_T} H(P_y^+). \tag{10.35}$$

Step 4: Augment the current partial solution $\{x_{ij} \mid (i,j) \in S\}$ with the arc flows $\{\overline{y}_{ij} \mid (i,j) \in T\}$ thus obtained, and proceed with the next iteration.

As an example of this algorithm, let us consider the traveling salesman problem, and let us use as base heuristic the nearest neighbor method, whereby we start from some simple path and at each iteration, we add a node that does not close a cycle and minimizes the cost of the enlarged path. The rollout algorithm operates as follows: After k iterations, we have a path $\{i_1, \ldots, i_k\}$ consisting of distinct nodes. At the next iteration, we run the nearest neighbor heuristic starting from each of the paths $\{i_1, \ldots, i_k, i\}$ with $i \neq i_1, \ldots, i_k$, and obtain a corresponding tour. We then select as next node i_{k+1} of the path the node i that corresponds to the best tour thus obtained. Here, the set of arcs used to augment the current partial solution in the rollout algorithm is

$$T = \big\{(i_k, i) \mid i \neq i_1, \ldots, i_k\big\},$$

and at the kth iteration the flows of all of these arcs are set to 0, except for arc (i_k, i_{k+1}) whose flow is set to 1.

Note that a rollout algorithm requires considerably more computation than the base heuristic. For example, in the case where the subset T in Step

1 consists of a single arc, the rollout algorithm requires $O(mn)$ applications of the base heuristic, where

m is the number of arcs, and

n is a bound on the number of possible values of each arc flow.

Nonetheless the computational requirements of the rollout algorithm may be quite manageable. In particular, if the arc flows are restricted to be 0 or 1, and the base heuristic has polynomial running time, so does the corresponding rollout algorithm.

An important question is whether, given an initial partial solution, the rollout algorithm performs at least as well as its base heuristic when started from that solution. This can be guaranteed if the base heuristic is *sequentially consistent*. By this we mean that the heuristic has the following property:

Suppose that starting from a partial solution

$$P = \{x_{ij} \mid (i,j) \in S\},$$

the heuristic produces the complementary solution

$$\overline{P} = \{x_{ij} \mid (i,j) \notin S\}.$$

Then starting from the partial solution

$$P^+ = \{x_{ij} \mid (i,j) \in S \cup T\},$$

the heuristic produces a complementary solution

$$\overline{P}^+ = \{x_{ij} \mid (i,j) \notin S \cup T\},$$

which coincides with $\overline{P}$ on the arcs $(i,j) \notin S \cup T$.

As an example, it can be seen that the nearest neighbor heuristic for the traveling salesman problem, discussed earlier, is sequentially consistent. This is a manifestation of a more general property: many common base heuristics of the greedy type are by nature sequentially consistent (see Exercise 10.21). It may be verified, based on Eq. (10.35), that a sequentially consistent rollout algorithm keeps generating the same solution $P \cup \overline{P}$, up to the point where by examining the alternatives in Eq. (10.35) and by calculating their heuristic costs, it discovers a better solution. As a result, sequential consistency guarantees that the costs of the successive solutions $P \cup \overline{P}$ produced by the rollout algorithm are monotonically nonincreasing; that is, we have

$$H(P^+) \leq H(P)$$

at every iteration. Thus, the cost $f(x_t)$ of the solution x_t produced upon termination of the rollout algorithm is at least as small as the cost $f(x_0)$

of the initial solution x_0 produced by the base heuristic. For further elaboration of the sequential consistency property, we refer to the paper by Bertsekas, Tsitsiklis, and Wu [1997], which also discusses some underlying connections with the policy iteration method of dynamic programming.

A condition that is more general than sequential consistency is that the algorithm be *sequentially improving*, in the sense that at each iteration there holds

$$H(P^+) \leq H(P).$$

This property also guarantees that the cost of the solutions produced by the rollout algorithm is monotonically nonincreasing. The paper by Bertsekas, Tsitsiklis, and Wu [1997] discusses situations where this property holds, and shows that with fairly simple modification, a rollout algorithm can be made sequentially improving (see also Exercise 10.22).

There are a number of variations of the basic rollout algorithm described above. Here are some examples:

(1) We may adapt the rollout framework to use multiple heuristic algorithms. In particular, let us assume that we have K algorithms $\mathcal{H}_1, \ldots, \mathcal{H}_K$. The kth of these algorithms, given an augmented partial solution P_y^+, produces a heuristic cost $H_k(P_y^+)$. We may then use in the flow selection via Eq. (10.35) a heuristic cost of the form

$$H(P_y^+) = \min_{k=1,\ldots,K} H_k(P_y^+),$$

or of the form

$$H(P_y^+) = \sum_{k=1}^{K} r_k H_k(P_y^+),$$

where r_k are some fixed scalar weights obtained by trial and error.

(2) We may incorporate *multistep lookahead* or *selective depth lookahead* into the rollout framework. Here we consider augmenting the current partial solution $P = \{x_{ij} \mid (i,j) \in S\}$ with all possible values for the flows of a finite sequence of arcs that are not in S. We run the base heuristic from each of the corresponding augmented partial solutions, we select the sequence of arc flows with minimum heuristic cost, and then augment the current partial solution P with the first arc flow in this sequence. As an illustration, let us recall the traveling salesman problem with the nearest neighbor method used as the base heuristic. An example rollout algorithm with two-step lookahead operates as follows: We begin each iteration with a path $\{i_1, \ldots, i_k\}$. We run the nearest neighbor heuristic starting from each of the paths $\{i_1, \ldots, i_k, i\}$ with $i \neq i_1, \ldots, i_k$, and obtain a corresponding tour. We then form the subset $\overline{I}$ consisting of the m nodes $i \neq i_1, \ldots, i_k$ that correspond to the m best tours thus obtained. We run the nearest neighbor heuristic starting from each of the paths $\{i_1, \ldots, i_k, i, j\}$

with $i \in \overline{I}$ and $j \neq i_1, \ldots, i_k, i$, and obtain a corresponding tour. We then select as the next node i_{k+1} of the path the node $i \in \overline{I}$ that corresponds to a minimum cost tour.

(3) We may use alternative methods for computing a cost $H(P_y^+)$ of a candidate augmented partial solution P_y^+ for use in the flow selection via Eq. (10.35). For example, instead of generating this cost via the base heuristic, we may calculate it as the optimal or approximately optimal cost of a suitable optimization problem. In particular, it is possible to use a cost derived from Lagrangian relaxation, whereby at a given partial solution, an appropriate dual problem is solved, and its optimal cost is used in place of the heuristic cost H in Eq. (10.35). Alternatively, a complementary solution may be constructed based on minimization of the corresponding Lagrangian function. As another example, one may use as cost of a partial solution, some heuristic measure of quality of the partial solution; this idea forms the basis for computer chess, where various positions are evaluated using a heuristic "position evaluation function."

Let us provide a few examples of rollout algorithms. The first example is very simple, but illustrates well the notions of sequential consistency and sequential improvement.

Example 10.13. (One-Dimensional Walk)

Consider a person who walks on a straight line and at each time period takes either a unit step to the left or a unit step to the right. There is a cost function assigning cost $f(i)$ to each integer i. Given an integer starting point on the line, the person wants to minimize the cost of the point where he will end up after a given and fixed number N of steps.

We can formulate this problem as a problem of selecting a path in a graph (see Fig. 10.15). In particular, without loss of generality, let us assume that the starting point is the origin, so that the person's position after N steps will be some integer in the interval $[-N, N]$. The nodes of the graph are identified with pairs (k, m), where k is the number of steps taken so far ($k = 1, \ldots, N$) and m is the person's position ($m \in [-k, k]$). A node (k, m) with $k < N$ has two outgoing arcs with end nodes $(k+1, m-1)$ (corresponding to a left step) and $(k+1, m+1)$ (corresponding to a right step). Let us consider paths whose starting node is $(0, 0)$ and the destination node is of the form (N, m), where m is of the form $N - 2l$ and $l \in [0, N]$ is the number of left steps taken. The problem then is to find the path of this type such that $f(m)$ is minimized.

Let the base heuristic be the algorithm, which, starting at a node (k, m), takes $N - k$ successive steps to the right and terminates at the node $(N, m + N - k)$. It can be seen that this algorithm is sequentially consistent [the base heuristic generates the path $(k, m), (k+1, m+1), \ldots, (N, m+N-k)$ starting from (k, m), and also the path $(k + 1, m + 1), \ldots, (N, m + N - k)$ starting from $(k + 1, m + 1)$, so the criterion for sequential consistency is fulfilled].

The rollout algorithm, at node (k, m) compares the cost of the destination node $(N, m+N-k)$ (corresponding to taking a step to the right and then following the base heuristic) and the cost of the destination node $(N, m+N-k-2)$ (corresponding to taking a step to the left and then following the base heuristic). Let us say that an integer $i \in [-N+2, N-2]$ is a *local minimum* if $f(i-2) \geq f(i)$ and $f(i) \leq f(i+2)$. Let us also say that N (or $-N$) is a local minimum if $f(N-2) \leq f(N)$ [or $f(-N) \leq f(-N+2)$, respectively]. Then it can be seen that starting from the origin $(0,0)$, the rollout algorithm obtains the *local minimum that is closest to* N, (see Fig. 10.15). This is no worse (and typically better) than the integer N obtained by the base heuristic. This example illustrates how the rollout algorithm may exhibit "intelligence" that is totally lacking from the base heuristic.

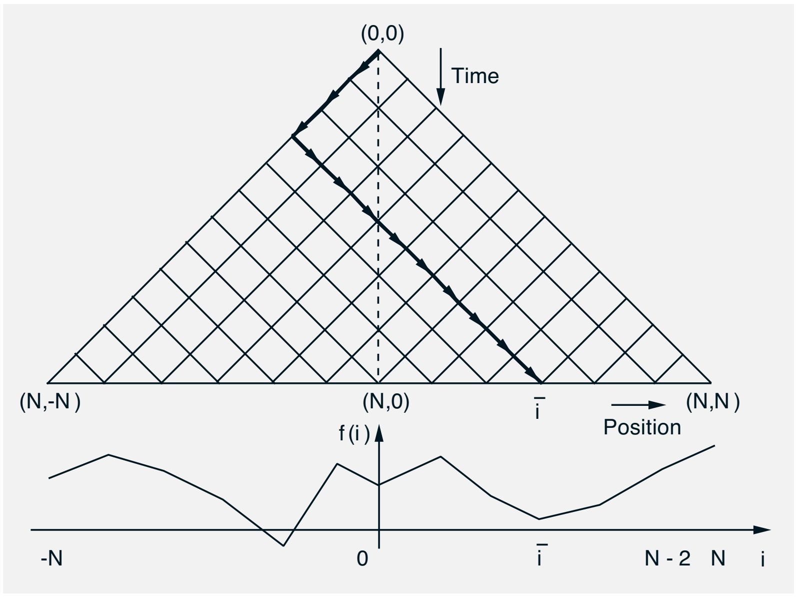


Figure 10.15: Illustration of the path generated by the rollout algorithm in Example 10.13. It keeps moving to the left up to the time where the base heuristic generates two destinations $(N, \bar{i})$ and $(N, \bar{i}-2)$ with $f(\bar{i}) \leq f(\bar{i}-2)$. Then it continues to move to the right ending at the destination $(N, \bar{i})$, which corresponds to the local minimum closest to N.

Consider next the case where the base heuristic is the algorithm that, starting at a node (k, m), compares the cost $f(m+N-k)$ (corresponding to taking all of the remaining $N-k$ steps to the right) and the cost $f(m-N+k)$ (corresponding to taking all of the remaining $N-k$ steps to the left), and accordingly moves to node

$$(N, m+N-k) \qquad \text{if} \qquad f(m+N-k) \leq f(m-N+k),$$

or to node

$$(N, m - N + k) \qquad \text{if} \qquad f(m - N + k) < f(m + N - k).$$

It can be seen that this base heuristic is not sequentially consistent, but is instead sequentially improving. It can then be verified that starting from the origin $(0, 0)$, the rollout algorithm obtains the *global minimum* of f in the interval $[-N, N]$, while the base heuristic obtains the better of the two points $-N$ and N.

Example 10.14. Constrained Traveling Salesman Problem

Consider the traveling salesman problem of Example 10.1, where we want to minimize the cost

$$\sum_{(i,j)\in T} a_{ij},$$

of a tour T, while satisfying the side constraints

$$\sum_{(i,j)\in T} c_{ij}^k \le d^k, \qquad \forall\ k = 1, \ldots, K.$$

A rollout algorithm starts with the trivial path $P = (s)$, where s is some initial node, progressively constructs a sequence of paths $P = (s, i_1, \ldots, i_m)$, $m = 1, \ldots, N - 1$, consisting of distinct nodes, and then completes a tour by adding the arc (i_{N-1}, s). The rollout procedure is as follows.

We introduce nonnegative penalty coefficients μ^k for the side constraints, and we form modified arc traversal costs $\hat{a}_{ij}$, given by

$$\hat{a}_{ij} = a_{ij} + \sum_{k=1}^{K} \mu^k c_{ij}^k.$$

The method of obtaining μ^k is immaterial for our purposes in this example, but we note that one possibility is to use the Lagrangian relaxation method of Section 10.3. We assume that we have a heuristic algorithm that can complete the current path $P = (s, i_1, \ldots, i_m)$ with a path $(i_{m+1}, \ldots, i_{N-1}, s)$, thereby obtaining a tour $T^*(P)$ that has approximately minimum modified cost. Some of the heuristics mentioned in Example 10.1, including the k-OPT heuristic, can be used for this purpose. Furthermore, we assume that by using another heuristic, we can complete the current path P to a tour $\hat{T}(P)$ that satisfies all the side constraints.

Given the current path $P = (s, i_1, \ldots, i_m)$, the rollout algorithm, considers the set A_m of all arcs $(i_m, j) \in \mathcal{A}$ such that j does not belong to P. For each of the nodes j such that $(i_m, j) \in A_m$, it considers the expanded path $P_e = (s, i_1, \ldots, i_m, j)$ and obtains the tours $T^*(P_e)$ and $\hat{T}(P_e)$, using the heuristics mentioned earlier. The rollout algorithm then adds to the current partial path P the node j for which the tour $T^*(P_e)$ satisfies the side constraints and has minimum cost (with respect to the arc costs a_{ij}); if no path

$T^*(P_e)$ satisfies the side constraints, the algorithm adds to the current path the node j for which the tour $\hat{T}(P_e)$ has minimum cost.

One of the drawbacks of the scheme just described is that it requires the approximate solution of a large number of traveling salesman problems. A faster variant is obtained if the arc set A_m above is restricted to be a suitably chosen subset of the eligible arcs (i_m, j), such for example those whose length does not exceed a certain threshold.

Finally, it is interesting to compare rollout algorithms with the local search methods of the preceding section. Both types of algorithms generate a sequence of solutions, but in the case of a rollout algorithm, the generated solutions are partial (except at termination), while in a local search method, the generated solutions are complete. In both types of algorithms, the next solution is generated from within a neighborhood of the current solution, but the selection criterion in rollout algorithms is the *estimated* cost of the solution as obtained by the base heuristic, while in local search methods, it is typically the *true* cost of the solution. Finally, in rollout algorithms, there is no concern about local minima and cycling, but there is also no provision for improving a complete solution after it is obtained.

There are interesting possibilities for combining a rollout algorithm with a local search method. In particular, one may use a local search method as part of a base heuristic in a rollout algorithm; here, the local search method could be fairly unsophisticated, since one may hope that the rollout process will provide an effective mechanism for solution improvement. Alternatively, one may first use a rollout algorithm to obtain a complete solution, and then use a local search method in an effort to improve this solution.

10.6 NOTES, SOURCES, AND EXERCISES

There is a great variety of integer constrained network flow problems, and the associated methodological and applications literature is vast. For textbook treatments at various levels of sophistication, which also cover broader aspects of integer programming, see Lawler [1976], Zoutendijk [1976], Papadimitriou and Steiglitz [1982], Minoux [1986a], Schrijver [1986], Nemhauser and Wolsey [1988], Bogart [1990], Pulleyblank, Cook, Cunningham, and Schrijver [1993], Cameron [1994], and Cook, Cunningham, Pulleyblank, and Schrijver [1998]. Volumes 7 and 8 of the Handbooks in Operations Research and Management Science, edited by Ball, Magnanti, Monma, and Nemhauser [1995a], [1995b], are devoted to network theory and applications, and include several excellent survey papers with large bibliographies. O'hEigeartaigh, Lenstra, and Rinnoy Kan [1985] provide an extensive bibliography on combinatorial optimization. Von Randow [1982],

[1985] gives an extensive bibliography on integer programming and related subjects.

The traveling salesman problem has been associated with many of the important investigations in discrete optimization. It was first considered in a modern setting by Dantzig, Fulkerson, and Johnson [1954], whose paper stimulated much interest and research. The edited volume by Lawler, Lenstra, Rinnoy Kan, and Shmoys [1985] focuses on the traveling salesman problem and its variations, and the papers by Junger, Reinelt, and Rinaldi [1995], and by Johnson and McGeoch [1997] provide extensive surveys of the subject. There is a large literature on the use of polyhedral approximations to the feasible set of integer programming problems and the traveling salesman problem in particular; see, for example, the papers by Cornuejols, Fonlupt, and Naddef [1985], Grötschel and Padberg [1985], Padberg and Grötschel [1985], Pulleyblank [1983], and the books by Nemhauser and Wolsey [1988], and Schrijver [1986]. The papers by Burkard [1990], Gilmore, Lawler, and Shmoys [1985], and Tsitsiklis [1992] discuss some special cases of the traveling salesman problem and some extensions.

The monograph by Martello and Toth [1990] is devoted to generalized assignment problems, including ones with integer constraints. The book by Kershenbaum [1993] provides a lot of material on tree construction and network design algorithms for data communications; see also Monma and Sheng [1986], Minoux [1989], Bertsekas and Gallager [1992], and Grötschel, Monma, and Stoer [1995]. Exact and heuristic methods for the Steiner tree problem are surveyed by Winter [1987] and Voß [1992].

Matching problems are discussed in detail in the monograph by Lovasz and Plummer [1985], the survey by Gerards [1995], and Chapter 10 of the book by Murty [1992]. For vehicle and arc routing problems, see the surveys by Assad and Golden [1995], Desrosiers, Dumas, Solomon, and Soumis [1995], Eiselt, Gendreau, and Laporte [1995a], [1995b], Federgruen and Simchi-Levi [1995], Fisher [1995], and Powell, Jaillet, and Odoni [1995].

An important application of multidimensional assignment problems arises in the context of multi-target tracking and data association; see Blackman [1986], Bar-Shalom and Fortman [1988], Pattipati, Deb, Bar-Shalom, and Washburn [1992], Poore [1994], Poore and Robertson [1997]. The material on the error bounds for the enforced separation heuristic in three-dimensional assignment problems (Exercise 10.31) is apparently new.

Integer multicommodity flow problems are discussed by Barnhart, Hane, and Vance [1997]. Nonlinear, nonconvex network optimization is discussed by Lamar [1993], Bell and Lamar [1993], as well as in general texts on global optimization; see Pardalos and Rosen [1987], Floudas [1995], and Horst, Pardalos, and Thoai [1995]. For a textbook treatment of scheduling (cf. Exercises 10.23-10.27), see Pinedo [1995].

Branch-and-bound has its origins in the traveling salesman paper by Dantzig, Fulkerson, and Johnson [1954]. Their paper was followed by Croes [1958], Eastman [1958], and Land and Doig [1960], who considered versions of the branch-and-bound method in the context of various integer programming problems. The term "branch-and-bound" was first used by Little, Murty, Sweeney, and Karel [1963], in the context of the traveling salesman problem. Balas and Toth [1985], and Nemhauser and Wolsey [1988] provide extensive surveys of branch-and-bound.

Lagrangian relaxation was suggested in the context of discrete optimization by Held and Karp [1970], [1971]. Subgradient methods were introduced by Shor in the Soviet Union during the middle 60s. The convergence properties of subgradient methods and their variations are discussed in a number of sources, including Auslender [1976], Goffin [1977], Shapiro [1979], Shor [1985], Poljak [1987], Hiriart-Urruty and Lemarechal [1993], Brannlund [1993], Bertsekas [1995b], and Goffin and Kiwiel [1996].

Cutting plane methods were proposed by Cheney and Goldstein [1959], and by Kelley [1960]; see also the book by Goldstein [1967]. Central cutting plane methods were introduced by Elzinga and Moore [1975]. More recent proposals, some of which relate to interior point methods, are discussed in Goffin and Vial [1990], Goffin, Haurie, and Vial [1992], Ye [1992], Kortanek and No [1993], Goffin, Luo, and Ye [1993], [1996], Atkinson and Vaidya [1995], Nesterov [1995], Luo [1996], and Kiwiel [1997b].

Three historically important references on decomposition methods are Dantzig and Wolfe [1960], Benders [1962], and Everett [1963]. An early text on large-scale optimization and decomposition is Lasdon [1970]; see also Geoffrion [1970], [1974]. Subgradient methods have been applied to the solution of multicommodity flow problems using a decomposition framework by Kennington and Shalaby [1977]. The book by Censor and Zenios [1997] discusses several applications of decomposition in a variety of algorithmic contexts.

The literature of local search methods is extensive. The edited volume by Aarts and Lenstra [1997] contains several surveys of broad classes of methods. Osman and Laporte [1996] provide an extensive bibliography.

The book by Goldberg [1989] focuses on genetic algorithms. Tabu search was initiated with the works of Glover [1986] and Hansen [1986]. The book by Glover and Laguna [1997], and the surveys by Glover [1989], [1990], Glover, Taillard, and de Verra [1993] provide detailed expositions and give many references.

Simulated annealing was proposed by Kirkpatrick, Gelatt, and Vecchi [1983] based on earlier suggestions by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller [1953]; see also Cerny [1985]. The main theoretical convergence properties of the method were established by Hajek [1988] and Tsitsiklis [1989]; see also the papers by Connors and Kumar [1989], Gelfand and Mitter [1989], and Bertsimas and Tsitsiklis [1993], and the book by Korst, Aarts, and Korst [1989]. A framework for integration of local search methods is presented by Fox [1993], [1995].

Rollout algorithms for discrete optimization were proposed in the book by Bertsekas and Tsitsiklis [1996] in the context of the neuro-dynamic programming methodology, and in the paper by Bertsekas, Tsitsiklis, and Wu [1997]. An application to scheduling using the framework of the quiz problem (cf. Exercises 10.28 and 10.29) is described by Bertsekas and Castañon [1998]. The idea of sequential selection of candidates for participation in a solution is implicit in several combinatorial optimization contexts. For example this idea is embodied in the sequential fan candidate list strategy as applied in tabu search (see Glover, Taillard, and de Werra [1993]). A similar idea is also used in the sequential automatic test procedures of Pattipati (see e.g., Pattipati and Alexandridis [1990]).

E X E R C I S E S

10.1

Consider the symmetric traveling salesman problem with the graph shown in Fig. 10.16.

(a) Find a suboptimal solution using the nearest neighbor heuristic starting from node 1.

(b) Find a suboptimal solution by first solving an assignment problem, and by then merging subtours.

(c) Try to improve the solutions found in (a) and (b) by using the 2-OPT heuristic.

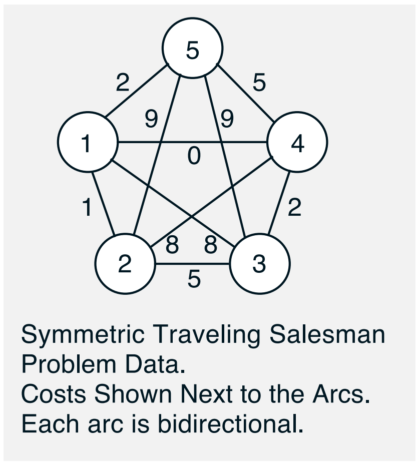

5
1
4
2
3
2
5
9
9
0
1
2
8
8
5
Symmetric Traveling Salesman Problem Data.
Costs Shown Next to the Arcs.
Each arc is bidirectional.

Figure 10.16: Data for a symmetric traveling salesman problem (cf. Exercise 10.1). The arc costs are shown next to the arcs. Each arc is bidirectional.

10.2 (Minimum Cost Cycles)

Consider a strongly connected graph with a nonnegative cost for each arc. We want to find a forward cycle of minimum cost that contains all nodes but is not necessarily simple; that is, a node or an arc may be traversed multiple times.

(a) Convert this problem into a traveling salesman problem. *Hint*: Construct a complete graph with cost of an arc (i, j) equal to the shortest distance from i to j in the original graph.

(b) Apply your method of part (a) to the graph of Fig. 10.17.

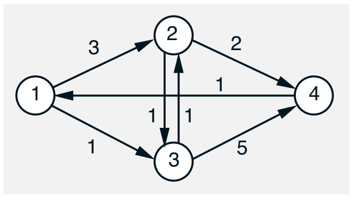


Figure 10.17: Data for a minimum cost cycle problem (cf. Exercise 10.2). The arc costs are shown next to the arcs.

10.3

Consider the problem of checking whether a given graph contains a simple cycle that passes through all the nodes. (The cycle need not be forward.) Formulate this problem as a symmetric traveling salesman problem. *Hint*: Consider a complete graph where the cost of an arc (i, j) is 1 if (i, j) or (j, i) is an arc of the original graph, and is 2 otherwise.

10.4

Show that an asymmetric traveling salesman problem with nodes $1, \ldots, N$ and arc costs a_{ij} can be converted to a symmetric traveling salesman problem involving a graph with nodes $1, \ldots, N, N+1, \ldots, 2N$, and the arc costs

$$\overline{a}_{i(N+j)} = \begin{cases} a_{ij} & \text{if } i, j = 1, \ldots, N,\ i \neq j, \\ -M & \text{if } i = j, \end{cases}$$

where M is a sufficiently large number. *Hint*: All arcs with cost $-M$ must be included in an optimal tour of the symmetric version.

10.5

Consider the problem of finding a shortest (forward) path from an origin node s to a destination node t of a graph with given arc lengths, subject to the additional constraint that the path passes through every node exactly once.

(a) Show that the problem can be converted to a traveling salesman problem by adding an artificial arc (t, s) of length $-M$, where M is a sufficiently large number.

(b) (Longest Path Problem) Consider the problem of finding a simple forward path from s to t that has a maximum number of arcs. Show that the problem can be converted to a traveling salesman problem.

10.6

Consider the problem of finding a shortest (forward) path in a graph with given arc lengths, subject to the constraint that the path passes through every node exactly once (the choice of start and end nodes of the path is subject to optimization). Formulate the problem as a traveling salesman problem.

10.7 (Traveling Salesman Problem/Triangle Inequality)

Consider a symmetric traveling salesman problem where the arc costs are nonnegative and satisfy the following *triangle inequality*:

$$a_{ij} \leq a_{ik} + a_{kj}, \qquad \text{for all nodes } i, j, k.$$

This problem has some special algorithmic properties.

(a) Consider a procedure, which given a cycle $\{i_0, i_1, \ldots, i_K, i_0\}$ that contains all the nodes (but passes through some of them multiple times), obtains a tour by deleting nodes after their first appearance in the cycle; e.g., in a 5-node problem, starting from the cycle $\{1, 3, 5, 2, 3, 4, 2, 1\}$, the procedure produces the tour $\{1, 3, 5, 2, 4, 1\}$. Use the triangle inequality to show that the tour thus obtained has no greater cost than the original cycle.

(b) Starting with a spanning tree of the graph, use the procedure of part (a) to construct a tour with cost equal to at most two times the total cost of the spanning tree. *Hint*: The cycle should cross each arc of the spanning tree exactly once in each direction. "Double" each arc of the spanning tree. Use the fact that if a graph is connected and each of its nodes has even degree, there is a cycle that contains all the arcs of the graph exactly once (cf. Exercise 1.5).

(c) (Double tree heuristic) Start with a minimum cost spanning tree of the graph, and use part (b) to construct a tour with cost equal to at most twice the optimal tour cost.

(d) Verify that the problem of Fig. 10.18 satisfies the triangle inequality. Apply the method of part (c) to this problem.

10.8 (Christofides' Traveling Salesman Heuristic)

Consider a symmetric traveling salesman problem where the arc costs are nonnegative and satisfy the triangle inequality (cf. the preceding exercise). Let R

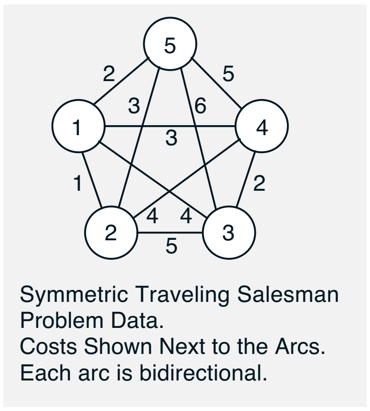

5
1
4
2
3
2
5
3
6
3
1
2
4
4
5
Symmetric Traveling Salesman Problem Data.
Costs Shown Next to the Arcs.
Each arc is bidirectional.

Figure 10.18: Data for a symmetric traveling salesman problem (cf. Exercises 10.7 and 10.8). The arc costs are shown next to the arcs.

be a minimum cost spanning tree of the graph (cf. Exercise 2.30), and let S be the subset of the nodes that has an odd number of incident arcs in R. A *perfect matching* of the nodes of S is a subset of arcs such that every node of S is an end node of exactly one arc of the subset and each arc of the subset has end nodes in S. Suppose that M is a perfect matching of the nodes of S that has minimum sum of arc costs. Construct a tour that consists of the arcs of M and some of the arcs of R, and show that its weight is no more than 3/2 times the optimal tour cost. Solve the problem of Fig. 10.18 using this heuristic, and find the ratio of the solution cost to the optimal tour cost. *Hint*: Note that the total cost of the arcs of M is at most 1/2 the optimal tour cost. Also, use the fact that if a graph is connected and each of its nodes has even degree, there is a cycle that contains all the arcs of the graph exactly once (cf. Exercise 1.5).

10.9 (K-Traveling Salesmen Problem)

Consider the version of the traveling salesman problem where there are K salesmen that start at city 1, return to city 1, and collectively must visit all other cities exactly once. Transform the problem into an ordinary traveling salesman problem. *Hint*: Split city 1 into K cities.

10.10 (Degree-Constrained Minimum Weight Spanning Trees)

Consider the minimum weight spanning tree problem, subject to the additional constraint that the number of tree arcs that are incident to a single given node s should be no greater than a given integer k. Consider adding a nonnegative weight w to the weight of all incident arcs of node s, solving the corresponding unconstrained spanning tree problem, and gradually increasing w until the degree constraint is satisfied.

(a) State a polynomial algorithm for doing this and derive its running time.

(b) Use this algorithm to solve the problem of Fig. 10.19, where the degree of node 1 is required to be no more than 2.

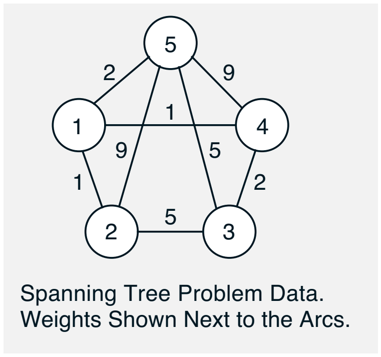


Figure 10.19: Data for a minimum weight spanning tree problem (cf. Exercises 10.10 and 10.11). The arc weights are shown next to the arcs.

10.11 (Steiner Tree Problem Heuristic)

We are given a connected graph G with a nonnegative weight a_{ij} for each arc $(i, j) \in A$. We assume that if an arc (i, j) is present, the reverse arc (j, i) is also present, and $a_{ij} = a_{ji}$. Consider the problem of finding a tree in G that spans a given subset of nodes S and has minimum weight over all such trees.

(a) Let W^* be the weight of this tree. Consider the graph $I(G)$, which has node set S and is complete (has an arc connecting every pair of its nodes). Let the weight for each arc (i, j) of $I(G)$ be equal to the shortest distance in the graph G from the node $i \in S$ to the node $j \in S$. Let T be a minimum weight spanning tree of $I(G)$. Show that the weight of T is no greater than $2W^*$. *Hint*: Consider a minimum weight tour in $I(G)$. Show that the weight of this tour is no less than the weight of T and no more than $2W^*$.

(b) Construct a heuristic based on part (a) and apply it to the problem of Fig. 10.19, where $S = \{1, 3, 5\}$.

10.12 (A General Heuristic for Spanning Tree Problems)

Consider a minimum weight spanning tree problem with an additional side constraint denoted by C (for example, a degree constraint on each node). A general heuristic (given by Deo and Kumar [1997]) is to solve the problem neglecting the constraint C, and then to add a scalar penalty to the cost of the arcs that "contribute most" to violation of C. This is then repeated as many times as desired.

(a) Construct a heuristic of this type for the capacitated spanning tree problem (cf. Example 10.3).

(b) Adapt this heuristic to a capacitated Steiner tree problem.

10.13

Consider the Königsberg bridge problem (cf. Fig. 10.6).

(a) Suppose that there existed a second bridge connecting the islands B and C, and also another bridge connecting the land areas A and D. Construct an Euler cycle that crosses each of the bridges exactly once.

(b) Suppose the bridge connecting the islands B and C has collapsed. Construct an Euler path, i.e., a path (not necessarily a cycle) that passes through each arc of the graph exactly once.

(c) Construct an optimal postman cycle assuming all arcs have cost 1.

10.14

Formulate the capacitated spanning tree problem given in Fig. 10.5 as an integer-constrained network flow problem.

10.15 (Network Formulation of Nonbipartite Matching)

Consider the nonbipartite matching problem of Example 10.4. Replace each node i with a pair of nodes i and i'. For every arc (i, j) of the original problem, introduce an arc (i, j') with value a_{ij} and an arc (j, i') also with value a_{ij}. Show that the problem can be formulated as the assignment-like problem involving the conservation of flow inequalities

$$\sum_{j'} x_{ij'} \leq 1, \qquad \forall\, i,$$

$$\sum_{i} x_{ij'} \leq 1, \qquad \forall\, j',$$

the integer constraints $x_{ij'} \in \{0, 1\}$, and the side constraints

$$\sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} + \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji} \leq 1, \qquad \forall\, i \in \mathcal{N},$$

or

$$\sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} + \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji} = 1, \qquad \forall\, i \in \mathcal{N},$$

in the case where a perfect matching is sought.

10.16 (Matching Solution of the Chinese Postman Problem)

Given a Chinese postman problem, delete all nodes of even degree together with all their incident arcs. Find a perfect matching of minimum cost in the remaining graph. Create an expanded version of the original problem's graph by adding an extra copy of each arc of the minimum cost matching. Show that an Euler cycle of the expanded graph is an optimal solution to the Chinese postman problem.

10.17 (Solution of the Directed Chinese Postman Problem)

Consider expanding the graph of the directed Chinese postman problem by duplicating arcs so that the number of incoming arcs to each node is equal to the number of its outgoing arcs. A forward Euler cycle of the expanded graph corresponds to a solution of the directed Chinese postman problem. Show that the optimal expanded graph is obtained by minimizing

$$\sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

subject to the constraints

$$\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j\mid(j,i)\in\mathcal{A}\}} x_{ji} = d_i, \qquad \forall\ i\in\mathcal{N},$$

$$0 \le x_{ij}, \qquad \forall\ (i,j)\in\mathcal{A},$$

where d_i is the difference between the number of incoming arcs to i and the number of outgoing arcs from i.

10.18 (Shortest Paths and Branch-and-Bound)

Consider a general integer-constrained problem of the form

$$\begin{aligned} &\text{minimize} \ \ f(x_1,\ldots,x_n)\\ &\text{subject to} \ \ x\in X, \qquad x_i\in\{0,1\}, \quad i=1,\ldots,n, \end{aligned}$$

where X is some set. Construct a branch-and-bound tree that starts with a subproblem where the integer constraints are relaxed, and proceeds with successive restriction of the variables $x_1,\ldots,x_n$ to the values 0 or 1.

(a) Show that the original integer-constrained problem is equivalent to a single origin/single destination shortest path problem that involves the branch-and-bound tree. *Hint*: As an example, for the traveling salesman problem, nodes of the tree correspond to sequences $(i_1,\ldots,i_k)$ of distinct cities, and arcs correspond to pairs of nodes $(i_1,\ldots,i_k)$ and $(i_1,\ldots,i_k,i_{k+1})$.

(b) Modify the label correcting method of Section 2.5.2 so that it becomes similar to the branch-and-bound method (see also the discussion in Section 2.5.2).

10.19

Use the branch-and-bound method to solve the capacitated spanning tree problem of Fig. 10.5.

10.20 (Simulated Annealing)

In the context of simulated annealing, assume that T is kept constant and let p_{xy} be the probability that when the current solution is x, the next solution sampled is y. Consider the special case where $p_{xy} = p_{yx}$ for all feasible solutions x and y, and assume that the Markov chain defined by the probabilities p_{xy} is irreducible, in the sense that there is positive probability to go from any x to any y, with one or more samples. Show that the steady-state probability of a solution x is

$$\pi_x = \frac{e^{-f(x)/T}}{C},$$

where

$$C = \sum_{x\in F} e^{-f(x)/T}.$$

Hint: This exercise assumes some basic knowledge of the theory of Markov chains. Let q_{xy} be the probability that y is the next solution if x is the current solution, i.e.,

$$q_{xy} = \begin{cases} p_{xy} e^{-\big(f(y)-f(x)\big)/T} & \text{if } f(y) > f(x), \\ p_{xy} & \text{otherwise.} \end{cases}$$

Show that for all x and y we have $\pi_y q_{yx} = \pi_x q_{xy}$, and that $\pi_y = \sum_{x\in F} \pi_x q_{xy}$. This equality together with $\sum_{x\in F} \pi_x = 1$ is sufficient to show the result.

10.21 (Rollout Algorithms Based on Greedy Algorithms)

In the context of the rollout algorithm, suppose that given a partial solution $P = \big\{x_{ij} \mid (i,j) \in S\big\}$, we have an estimate $c(P)$ of the optimal cost over all feasible solutions that are consistent with P, in the sense that there exists a complementary solution $\overline{P} = \big\{x_{ij} \mid (i,j) \notin S\big\}$ such that $P \cup \overline{P}$ is feasible. Consider a heuristic algorithm, which is *greedy* with respect to $c(P)$, in the sense that it starts from $S = \emptyset$, and given the partial solution $P = \big\{x_{ij} \mid (i,j) \in S\big\}$, it selects a set of arcs T, forms the collection F_T of all possible values of the arc flows $y = \big\{y_{ij} \mid (i,j) \in T\big\}$, and finds

$$\overline{y} = \arg\min_{y\in F_T} c(P_y^+). \tag{10.36}$$

where

$$P_y^+ = \Big\{\big\{x_{ij} \mid (i,j) \in S\big\}, \big\{y_{ij} \mid (i,j) \in T\big\}\Big\}.$$

It then augments P with the arc flows $\overline{y}$ thus obtained, and repeats up to obtaining a complete solution. Assume that the set of arcs T selected depends only on P. Furthermore, the ties in the minimization of Eq. (10.36) are resolved in a fixed manner that depends only on P. Show that the rollout algorithm that uses the greedy algorithm as a base heuristic is sequentially consistent.

10.22 (Sequentially Improving Rollout Algorithm)

Consider a variant of the rollout algorithm that starts with the empty set of arcs, and maintains, in addition to the current partial solution $P = \{x_{ij} \mid (i,j) \in S\}$, a complementary solution $P' = \{x'_{ij} \mid (i,j) \notin S\}$, and the corresponding (complete) flow vector $x' = P \cup P'$. At the typical iteration, we select a subset T of arcs that are not in S, and we consider the collection F_T of all possible values of the arc flows $y = \{y_{ij} \mid (i,j) \in T\}$. Then, if

$$\min_{y \in F_T} H(P_y^+) < f(x'),$$

we augment the current partial solution $\{x_{ij} \mid (i,j) \in S\}$ with the arc flows $\overline{y} = \{\overline{y}_{ij} \mid (i,j) \in T\}$ that attain the minimum above, and we set x' equal to the complete solution generated by the base heuristic starting from $P_{\overline{y}}^+$. Otherwise, we augment the current partial solution to $\{x_{ij} \mid (i,j) \in S\}$ with the arc flows $\{x'_{ij} \mid (i,j) \in T\}$ and we leave x' unchanged. Prove that this rollout algorithm is sequentially improving in the sense that the heuristic costs of the partial solutions generated are monotonically nonincreasing.

10.23 (Scheduling Problems Viewed as Assignment Problems)

A machine can be used to perform a subset of N given tasks over T time periods. At each time period t, only a subset $A(t)$ of tasks can be performed. Each task j has value $v_j(t)$ when performed at period t.

(a) Formulate the problem of finding the sequence of tasks of maximal total value as an assignment problem. *Hint*: Assign time periods to tasks.

(b) Suppose that there are in addition some precedence constraints of the general form: Task j must be performed before task j' can be performed. Formulate the problem as an assignment problem with side constraints and integer constraints. Give an example where the integer constraints are essential.

(c) Repeat part (b) for the case where there are no precedence constraints, but instead some of the tasks require more than one time period.

10.24 (Scheduling and the Interchange Argument)

In some scheduling problems it is useful to try to characterize a globally optimal solution based on the fact that it is locally optimal with respect to the 2-OPT heuristic. This is known as the *interchange argument*, and amounts to starting with an optimal schedule and checking to see what happens when any two tasks in the schedule are interchanged. As an example, suppose that we have N jobs to process in sequential order with the ith job requiring a given time T_i for its execution. If job i is completed at time t, the reward is $\alpha^t R_i$, where α is a given discount factor with $0 < \alpha < 1$. The problem is to find a schedule that maximizes the total reward. Suppose that $L = (i_0, \ldots, i_{k-1}, i, j, i_{k+2}, \ldots, i_{N-1})$ is an optimal job schedule, and consider the schedule $L' = (i_0, \ldots, i_{k-1}, j, i, i_{k+2}, \ldots, i_{N-1})$ obtained by interchanging i and j. Let t_k be the time of completion of job i_{k-1}. Compare the rewards of the two schedules, and show that

$$\frac{\alpha^{T_i} R_i}{1 - \alpha^{T_i}} \geq \frac{\alpha^{T_j} R_j}{1 - \alpha^{T_j}}.$$

Conclude that scheduling jobs in order of decreasing $\alpha^{T_i} R_i / \big(1 - \alpha^{T_i}\big)$ is optimal.

10.25 (Weighted Shortest Processing Time First Rule)

We want to schedule N tasks, the ith of which requires T_i time units. Let t_i denote the time of completion of the ith task, i.e.,

$$t_i = T_i + \sum_{\substack{\text{tasks } k \\ \text{completed before } i}} T_k.$$

Let w_i denote a positive weight indicating the importance of early completion of the ith task. Use an interchange argument (cf. Exercise 10.24) to show that in order to minimize the total weighted completion time $\sum_{i=1}^N w_i t_i$ we must order the tasks in decreasing order of w_i / T_i.

10.26

A busy professor has to complete N projects. Each project i has a deadline d_i and the time it takes the professor to complete it is T_i. The professor can work on only one project at a time and must complete it before moving on to a new project. For a given order of completion of the projects, denote by t_i the time of completion of project i, i.e.,

$$t_i = T_i + \sum_{\substack{\text{projects } k \\ \text{completed before } i}} T_k.$$

The professor wants to order the projects so as to minimize the maximum tardiness, given by

$$\max_{i \in \{1, \ldots, N\}} \max[0,\, t_i - d_i].$$

Use an interchange argument (cf. Exercise 10.24) to show that it is optimal to complete the projects in the order of their deadlines (do the project with the closest deadline first).

10.27 (Hardy's Theorem)

Let $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$ be monotonically nondecreasing sequences of numbers. Let us associate with each $i = 1, \ldots, n$ a distinct index j_i, and consider the expression $\sum_{i=1}^{n} a_i b_{j_i}$. Use an interchange argument (cf. Exercise 10.24) to show that this expression is maximized when $j_i = i$ for all i, and is minimized when $j_i = n - i + 1$ for all i.

10.28 (The Quiz Problem)

Consider a quiz contest where a person is given a list of N questions and can answer these questions in any order he chooses. Question i will be answered correctly with probability p_i, independently of earlier answers, and the person will then receive a reward R_i. At the first incorrect answer, the quiz terminates and the person is allowed to keep his previous rewards. The problem is to maximize the expected reward by choosing optimally the ordering of the questions.

(a) Show that to maximize the expected reward, questions should be answered in decreasing order of $p_i R_i/(1 - p_i)$. *Hint*: Use an interchange argument (cf. Exercise 10.24).

(b) Consider the variant of the problem where there is a maximum number of questions that can be answered, which is smaller than the number of questions that are available. Show that it is not necessarily optimal to answer the questions in order of decreasing $p_i R_i/(1 - p_i)$. *Hint*: Try the case where only one out of two available questions can be answered.

(c) Give a 2-OPT algorithm to solve the problem where the number of available questions is one more than the maximum number of questions that can be answered.

10.29 (Rollout Algorithm for the Quiz Problem)

Consider the quiz problem of Exercise 10.28 for the case where the maximum number of questions that can be answered is less or equal to the number of questions that are available. Consider the heuristic which answers questions in decreasing order of $p_i R_i/(1 - p_i)$, and use it as a base heuristic in a rollout algorithm. Show that the cost of the rollout algorithm is no worse than the cost of the base heuristic. *Hint*: Prove sequential consistency of the base heuristic.

10.30

This exercise shows that nondifferentiabilities of the dual function given in Section 10.3, often tend to arise at the most interesting points and thus cannot be ignored. Show that if there is a duality gap, then the dual function q is nondifferentiable at every dual optimal solution. *Hint*: Assume that q has a unique subgradient at a dual optimal solution μ^* and derive a contradiction by showing that any vector x_{μ^*} that minimizes $L(x, \mu^*)$ is primal optimal.

10.31 (Enforced Separation in 3-Dimensional Assignment)

Consider the 3-dimensional assignment problem of Example 10.7 that involves a set of jobs J, a set of machines M, and a set of workers W. We assume that each of the sets J, M, and W contains n elements, and that the constraints are equality constraints. Suppose that the problem is *ϵ-separable*, in the sense that for some $\overline{\beta}_{jm}$ and $\overline{\gamma}_{mw}$, and some $\epsilon \geq 0$, we have

$$|\overline{\beta}_{jm} + \overline{\gamma}_{mw} - a_{jmw}| \leq \epsilon, \qquad \forall\, j \in J,\, m \in M,\, w \in W,$$

where a_{jmw} is the value of the triplet (j, m, w).

(a) Show that if the problem is solved with a_{jmw} replaced by $\overline{\beta}_{jm} + \overline{\gamma}_{mw}$, the 3-dimensional assignment obtained achieves the optimal cost of the original problem within $2n\epsilon$.

(b) Suppose that we don't know $\overline{\beta}_{jm}$ and $\overline{\gamma}_{mw}$, and that we use the enforced separation approach of Example 10.7. Thus, we first solve the jobs-to-machines 2-dimensional assignment problem with values

$$b_{jm} = \max_{w \in W} a_{jmw}.$$

Let j_m be the job assigned to machine m, according to the solution of this problem. We then solve the machines-to-workers 2-dimensional assignment problem with values

$$c_{mw} = a_{j_m m w}.$$

Let w_m be the worker assigned to machine m, according to the solution of this problem. Show that the 3-dimensional assignment $\{(j_m, m, w_m) \mid m \in M\}$ achieves the optimal value of the original problem within $4n\epsilon$.

(c) Show that the result of part (b) also holds when b_{jm} is defined by

$$b_{jm} = a_{jm\overline{w}_m},$$

where $\overline{w}_m$ is any worker, instead of $b_{jm} = \max_{w \in W} a_{jmw}$.

(d) Show that the result of parts (b) and (c) also holds if J and W contain more than n elements, and we have the inequality constraints

$$\sum_{m \in M} \sum_{w \in W} x_{jmw} \leq 1, \qquad \forall\, j \in J,$$

$$\sum_{j \in J} \sum_{m \in M} x_{jmw} \leq 1, \qquad \forall\, w \in W,$$

in place of equality constraints.

10.32 (Lagrangian Relaxation in Multidimensional Assignment)

Apply the Lagrangian relaxation method to the multidimensional assignment problem of Example 10.7, in a way that requires the solution of 2-dimensional assignment problems. Derive the form of the corresponding subgradient algorithm.

10.33 (Separable Problems with Integer/Simplex Constraints)

Consider the problem

$$\begin{aligned} &\text{minimize} \quad \sum_{j=1}^{n} f_j(x_j) \\ &\text{subject to} \quad \sum_{j=1}^{n} x_j \le A, \\ &\qquad\qquad x_j \in \{0,1,\ldots,m_j\}, \qquad j=1,\ldots,n, \end{aligned}$$

where A and $m_1,\ldots,m_n$ are given positive integers, and each function f_j is convex over the interval $[0,m_j]$. Consider an iterative algorithm (due to Ibaraki and Katoh [1988]) that starts at $(0,\ldots,0)$ and maintains a feasible vector $(x_1,\ldots,x_n)$. At the typical iteration, we consider the set of indices $J = \{j \mid x_j < m_j\}$. If J is empty or $\sum_{j=1}^{n} x_j = A$, the algorithm terminates. Otherwise, we find an index $\overline{j} \in J$ that maximizes $f_j(x_j) - f_j(x_j+1)$. If $f_{\overline{j}}(x_{\overline{j}}) - f_{\overline{j}}(x_{\overline{j}}+1) \le 0$, the algorithm terminates. Otherwise, we increase $x_{\overline{j}}$ by one unit, and go to the next iteration. Show that upon termination, the algorithm yields an optimal solution. *Note*: The book by Ibaraki and Katoh [1988] contains a lot of material on this problem, and addresses the issues of efficient implementation.

10.34 (Constraint Relaxation and Lagrangian Relaxation)

The purpose of this exercise is to compare the lower bounds obtained by relaxing integer constraints and by dualizing the side constraints. Consider the nonlinear network optimization problem with a cost function $f(x)$, the conservation of flow constraints, and the additional constraint

$$x \in X = \big\{x \mid x_{ij} \in X_{ij},\ (i,j) \in \mathcal{A},\ g_t(x) \le 0,\ t=1,\ldots,r\big\},$$

where X_{ij} are given subsets of the real line and the functions g_t are *linear*. We assume that f is convex over the entire space of flow vectors x. We introduce a Lagrange multiplier μ_t for each of the side constraints $g_t(x) \le 0$, and we form the corresponding Lagrangian function

$$L(x,\mu) = f(x) + \sum_{t=1}^{r} \mu_t g_t(x).$$

Let C denote the set of all x satisfying the conservation of flow constraints, let f^* denote the optimal primal cost,

$$f^* = \inf_{x\in C,\, x_{ij}\in X_{ij},\, g_t(x)\le 0} f(x),$$

and let q^* denote the optimal dual cost,

$$q^* = \sup_{\mu\ge 0} q(\mu) = \sup_{\mu \ge 0} \inf_{x\in C,\, x_{ij}\in X_{ij}} L(x,\mu).$$

Let $\hat{X}_{ij}$ denote the interval which is the convex hull of the set X_{ij}, and denote by $\hat{f}$ the optimal cost of the problem, where each set X_{ij} is replaced by $\hat{X}_{ij}$,

$$\hat{f} = \inf_{x \in C,\, x_{ij} \in \hat{X}_{ij},\, g_t(x) \leq 0} f(x). \tag{10.37}$$

Note that this is a convex problem even if X_{ij} embodies integer constraints.

(a) Show that $\hat{f} \leq q^* \leq f^*$. *Hint*: Use Prop. 8.3 to show that problem (10.37) has no duality gap and compare its dual cost with q^*.

(b) Assume that f is linear. Show that $\hat{f} = q^*$. *Hint*: The problem involved in the definition of the dual function of problem (10.37) is a minimum cost flow problem.

(c) Assume that C is a general polyhedron; that is, C is specified by a finite number of linear equality and inequality constraints (rather than the conservation of flow constraints). Provide an example where f is linear and we have $\hat{f} < q^*$.

10.35 (Duality Gap of the Knapsack Problem)

Given objects $i = 1, \ldots, n$ with positive weights w_i and values v_i, we want to assemble a subset of the objects so that the sum of the weights of the subset does not exceed a given $T > 0$, and the sum of the values of the subset is maximized. This is the knapsack problem, which is a special case of a generalized assignment problem (see Example 8.7). The problem can be written as

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \\ &\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq T, \qquad x_i \in \{0, 1\}, \quad i = 1, \ldots, n. \end{aligned}$$

(a) Let f^* and q^* be the optimal primal and dual costs, respectively. Show that

$$0 \leq q^* - f^* \leq \max_{i=1,\ldots,n} v_i.$$

(b) Consider the problem where T is multiplied by a positive integer k and each object is replaced by k replicas of itself, while the object weights and values stay the same. Let $f^*(k)$ and $q^*(k)$ be the corresponding primal and dual costs. Show that

$$\frac{q^*(k) - f^*(k)}{f^*(k)} \leq \frac{1}{k} \frac{\max_{i=1,\ldots,n} v_i}{f^*},$$

so that the relative value of the duality gap tends to 0 as $k \to \infty$. *Note*: This exercise illustrates a generic property of many separable problems with integer constraints: as the number of variables increases, the duality gap decreases in relative terms (see Bertsekas [1982], Section 5.5, or Bertsekas [1995b], Section 5.1, for an analysis and a geometrical interpretation of this phenomenon).

10.36 (Convergence of the Subgradient Method)

Consider the subgradient method $\mu^{k+1} = [\mu^k + s^k g^k]^+$, where the stepsize is given by

$$s^k = \frac{q^* - q(\mu^k)}{\|g^k\|^2}$$

and q^* is the optimal dual cost (this stepsize requires knowledge of q^*, which is very restrictive, but the following Exercise 10.37 removes this restriction). Assume that there exists at least one optimal dual solution.

(a) Use Eq. (10.16) to show that $\{\mu^k\}$ is bounded.

(b) Use the fact that $\{g^k\}$ is bounded (since the dual function is piecewise linear), and Eq. (10.16) to show that $q(\mu^k) \to q^*$.

10.37 (A Convergent Variation of the Subgradient Method)

This exercise provides a convergence result for a common variation of the subgradient method (the result is due to Brannlund [1993]; see also Goffin and Kiwiel [1996]). Consider the iteration $\mu^{k+1} = [\mu^k + s^k g^k]^+$, where

$$s^k = \frac{\tilde{q} - q(\mu^k)}{\|g^k\|^2}.$$

(a) Suppose that $\tilde{q}$ is an *underestimate* of the optimal dual cost q^* such that $q(\mu^k) < \tilde{q} \le q^*$. [Here $\tilde{q}$ is fixed and the algorithm stops at μ^k if $q(\mu^k) \ge \tilde{q}$.] Use the fact that $\{g^k\}$ is bounded to show that either for some $\bar{k}$ we have $q(\mu^{\bar{k}}) \ge \tilde{q}$ or else $q(\mu^k) \to \tilde{q}$. *Hint*: Consider the function $\min\{q(\mu), \tilde{q}\}$ and use the results of Exercise 10.36.

(b) Suppose that $\tilde{q}$ is an *overestimate* of the optimal dual cost, that is, $\tilde{q} > q^*$. Use the fact that $\{g^k\}$ is bounded to show that the length of the path traveled by the method is infinite, that is,

$$\sum_{k=0}^{\infty} s^k \|g^k\| = \sum_{k=0}^{\infty} \frac{\tilde{q} - q(\mu^k)}{\|g^k\|} = \infty.$$

(c) Let δ^0 and B be two positive scalars. Consider the following version of the subgradient method. Given μ^k, apply successive subgradient iterations with $\tilde{q} = q(\mu^k) + \delta^k$ in the stepsize formula in place of $q(\mu^*)$, until one of the following two occurs:

   (1) The dual cost exceeds $q(\mu^k) + \delta^k/2$.

   (2) The length of the path traveled starting from μ^k exceeds B.

   Then set μ^{k+1} to the iterate with highest dual cost thus far. Furthermore, in case (1), set $\delta^{k+1} = \delta^k$, while in case (2), set $\delta^{k+1} = \delta^k/2$. Use the fact that $\{g^k\}$ is bounded to show that $q(\mu^k) \to q^*$.

10.38 (Convergence Rate of the Subgradient Method)

Consider the subgradient method of Exercise 10.36, and let μ^* be an optimal dual solution.

(a) Show that

$$\liminf_{k\to\infty} \sqrt{k}\big(q(\mu^*) - q(\mu^k)\big) = 0.$$

Hint: Use Eq. (10.16) to show that $\sum_{k=0}^{\infty}\big(q(\mu^*) - q(\mu^k)\big)^2 < \infty$. Assume that $\sqrt{k}\big(q(\mu^*) - q(\mu^k)\big) \geq \epsilon$ for some $\epsilon > 0$ and arbitrarily large k, and reach a contradiction.

(b) Assume that for some $a > 0$ and all k, we have $q(\mu^*) - q(\mu^k) \geq a\|\mu^* - \mu^k\|$. Use Eq. (10.16) to show that for all k we have

$$\|\mu^{k+1} - \mu^*\| \leq r\|\mu^k - \mu^*\|,$$

where $r = \sqrt{1 - a^2/b^2}$ and b is an upper bound on $\|g^k\|$.

10.39

Consider the cutting plane method.

(a) Give an example where the generated sequence $q(\mu^k)$ is not monotonically nondecreasing.

(b) Give an example where, at the kth iteration, the method finds an optimal dual solution μ^k but does not terminate because the criterion $q(\mu^k) = Q^k(\mu^k)$ is not satisfied.

10.40 (Computational Rollout Problem)

Consider the rollout algorithm for the traveling salesman problem using as base heuristic the nearest neighbor method, whereby we start from some simple path and at each iteration, we add a node that does not close a cycle and minimizes the cost of the enlarged path (see the paragraph following the description of the rollout algorithm iteration in Section 10.5). Write a computer program to apply this algorithm to the problem involving Hamilton's 20-node graph (Exercise 1.35) for the case where all arcs have randomly chosen costs from the range $[0, 1]$. For node pairs for which there is no arc, introduce an artificial arc with cost randomly chosen from the range $[100, 101]$. Compare the performances of the rollout algorithm and the nearest neighbor heuristic, and compile relevant statistics by running a suitable large collection of randomly generated problem instances. Verify that the rollout algorithm performs at least as well as the nearest neighbor heuristic for each instance (since it is sequentially consistent).

References

Aarts, E., and Lenstra, J. K., 1997. Local Search in Combinatorial Optimization, Wiley, N. Y.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B., 1989. "Network Flows," in Handbooks in Operations Research and Management Science, Vol. 1, Optimization, Nemhauser, G. L., Rinnooy-Kan, A. H. G., and Todd M. J. (eds.), North-Holland, Amsterdam, pp. 211-369.

Ahuja, R. K., Mehlhorn, K., Orlin, J. B., and Tarjan, R. E., 1990. "Faster Algorithms for the Shortest Path Problem," J. ACM, Vol. 37, 1990, pp. 213-223.

Ahuja, R. K., and Orlin, J. B., 1987. Private Communication.

Ahuja, R. K., and Orlin, J. B., 1989. "A Fast and Simple Algorithm for the Maximum Flow Problem," Operations Research, Vol. 37, pp. 748-759.

Amini, M. M., 1994. "Vectorization of an Auction Algorithm for Linear Cost Assignment Problem," Comput. Ind. Eng., Vol. 26, pp. 141-149.

Arezki, Y., and Van Vliet, D., 1990. "A Full Analytical Implementation of the PARTAN/Frank-Wolfe Algorithm for Equilibrium Assignment," Transportation Science, Vol. 24, pp. 58-62.

Assad, A. A., and Golden, B. L., 1995. "Arc Routing Methods and Applications," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L., (eds.), Vol. 8, North-Holland, Amsterdam, pp. 375-483.

Atkinson, D. S., and Vaidya, P. M., 1995. "A Cutting Plane Algorithm for Convex Programming that Uses Analytic Centers," Math. Programming, Vol. 69, pp. 1-44.

Auchmuty, G., 1989. "Variational Principles for Variational Inequalities," Numer. Functional Analysis and Optimization, Vol. 10, pp. 863-874.

Auslender, A., 1976. Optimization: Methodes Numeriques, Mason, Paris.

Balas, E., Miller, D., Pekny, J., and Toth, P., 1991. "A Parallel Shortest Path Algorithm for the Assignment Problem," J. ACM, Vol. 38, pp. 985-1004.

Balas, E., and Toth, P., 1985. "Branch and Bound Methods," in The Traveling Salesman Problem, Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B. (eds.), Wiley, N. Y., pp. 361-401.

Balinski, M. L., 1985. "Signature Methods for the Assignment Problem," Operations Research, Vol. 33, pp. 527-537.

Balinski, M. L., 1986. "A Competitive (Dual) Simplex Method for the Assignment Problem," Math. Programming, Vol. 34, pp. 125-141.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L., 1995a. Network Models, Handbooks in OR and MS, Vol. 7, North-Holland, Amsterdam.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L., 1995b. Network Routing, Handbooks in OR and MS, Vol. 8, North-Holland, Amsterdam.

Bar-Shalom, Y., and Fortman, T. E., 1988. Tracking and Data Association, Academic Press, N. Y.

Barnhart, C., Hane, C. H., and Vance, P. H., 1997. "Integer Multicommodity Flow Problems," in Network Optimization, Pardalos, P. M., Hearn, D. W., and Hager, W. W. (eds.), Springer-Verlag, N. Y., pp. 17-31.

Barr, R., Glover, F., and Klingman, D., 1977. "The Alternating Basis Algorithm for Assignment Problems," Math. Programming, Vol. 13, pp. 1-13.

Barr, R., Glover, F., and Klingman, D., 1978. "Generalized Alternating Path Algorithm for Transportation Problems," European J. of Operations Research, Vol. 2, pp. 137-144.

Barr, R., Glover, F., and Klingman, D., 1979. "Enhancement of Spanning Tree Labeling Procedures for Network Optimization," INFOR, Vol. 17, pp. 16-34.

Barr, R., and Hickman, B. L., 1994. "Parallel Simplex for Large Pure Network Problems - Computational Testing and Sources of Speedup," Operations Research, Vol. 42, pp. 65-80.

Bazaraa, M. S., Jarvis, J. J., and Sherali, H. D., 1990. Linear Programming and Network Flows (2nd Ed.), Wiley, N. Y.

Bazaraa, M. S., Sherali, H. D., and Shetty, C. M., 1993. Nonlinear Programming Theory and Algorithms (2nd Ed.), Wiley, N. Y.

Bell, G. J., and Lamar, B. W., 1997. "Solution Methods for Nonconvex Network Flow Problems," in Network Optimization, Pardalos, P. M., Hearn, D. W., and Hager, W. W. (eds.), Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, N. Y., pp. 32-50.

Bellman, R., 1957. Dynamic Programming, Princeton Univ. Press, Princeton, N. J.

Benders, J. F., 1962. "Partitioning Procedures for Solving Mixed Variables Programming Problems," Numer. Math., Vol. 4, pp. 238-252.

Beraldi, P., and Guerriero, F., 1997. "A Parallel Asynchronous Implementation of the Epsilon-Relaxation Method for the Linear Minimum Cost Flow Problem," Parallel Computing, Vol. 23, pp. 1021-1044.

Beraldi, P., Guerriero, F., and Musmanno, R., 1996. "Parallel Algorithms for Solving the Convex Minimum Cost Flow Problem," Tech. Report PARCOLAB No. 8/96, Dept. of Electronics, Informatics, and Systems, Univ. of Calabria.

Beraldi, P., Guerriero, F., and Musmanno, R., 1997. "Efficient Parallel Algorithms for the Minimum Cost Flow Problem," J. of Optimization Theory and Applications, Vol. 95, pp. 501-530.

Berge, C., 1962. The Theory of Graphs and its Applications, Wiley, N. Y.

Berge, C., and Ghouila-Houri, A., 1962. Programming, Games, and Transportation Networks, Wiley, N. Y.

Bertsekas, D. P., 1975a. "Nondifferentiable Optimization via Approximation," Math. Programming Studies, Vol. 3, North-Holland, Amsterdam, pp. 1-25.

Bertsekas, D. P., 1975b. "Necessary and Sufficient Conditions for a Penalty Method to be Exact," Math. Programming, Vol. 9, pp. 87-99.

Bertsekas, D. P., 1979a. "A Distributed Algorithm for the Assignment Problem," Lab. for Information and Decision Systems Working Paper, M.I.T., Cambridge, MA.

Bertsekas, D. P., 1979b. "Algorithms for Nonlinear Multicommodity Network Flow Problems," in International Symposium on Systems Optimization and Analysis, Bensoussan, A., and Lions, J. L. (eds.), Springer-Verlag, N. Y., pp. 210-224.

Bertsekas, D. P., 1980. "A Class of Optimal Routing Algorithms for Communication Networks," Proc. of the Fifth International Conference on Computer Communication, Atlanta, Ga., pp. 71-76.

Bertsekas, D. P., 1981. "A New Algorithm for the Assignment Problem," Math. Programming, Vol. 21, pp. 152-171.

Bertsekas, D. P., 1982. Constrained Optimization and Lagrange Multiplier Methods, Academic Press, N. Y. (republished in 1996 by Athena Scientific, Belmont, MA).

Bertsekas, D. P., 1985. "A Unified Framework for Minimum Cost Network Flow Problems," Math. Programming, Vol. 32, pp. 125-145.

Bertsekas, D. P., 1986a. "Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems," Lab. for Information and Decision Systems Report P-1606, M.I.T., Cambridge, MA.

Bertsekas, D. P., 1986b. "Distributed Relaxation Methods for Linear Network Flow Problems," Proceedings of 25th IEEE Conference on Decision and Control, Athens, Greece, pp. 2101-2106.

Bertsekas, D. P., 1988. "The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem," Annals of Operations Research, Vol. 14, pp. 105-123.

Bertsekas, D. P., 1990. "The Auction Algorithm for Assignment and Other Network Flow Problems: A Tutorial," Interfaces, Vol. 20, pp. 133-149.

Bertsekas, D. P., 1991a. Linear Network Optimization: Algorithms and Codes, MIT Press, Cambridge, MA.

Bertsekas, D. P., 1991b. "An Auction Algorithm for Shortest Paths," SIAM J. on Optimization, Vol. 1, pp. 425-447.

Bertsekas, D. P., 1992a. "Auction Algorithms for Network Flow Problems: A Tutorial Introduction," Computational Optimization and Applications, Vol. 1, pp. 7-66.

Bertsekas, D. P., 1992b. "Modified Auction Algorithms for Shortest Paths," Lab. for Information and Decision Systems Report P-2150, M.I.T., Cambridge, MA.

Bertsekas, D. P., 1992c. "An Auction Sequential Shortest Path Algorithm for the Minimum Cost Network Flow Problem," Lab. for Information and Decision Systems Report P-2146, M.I.T.

Bertsekas, D. P., 1993a. "A Simple and Fast Label Correcting Algorithm for Shortest Paths," Networks, Vol. 23, pp. 703-709.

Bertsekas, D. P., 1993b. "Mathematical Equivalence of the Auction Algorithm for Assignment and the ϵ-Relaxation (Preflow-Push) Method for Min Cost Flow," in Large Scale Optimization: State of the Art, Hager, W. W., Hearn, D. W., and Pardalos, P. M. (eds.), Kluwer, Boston, pp. 27-46.

Bertsekas, D. P., 1995a. Dynamic Programming and Optimal Control, Vols. I and II, Athena Scientific, Belmont, MA.

Bertsekas, D. P., 1995b. Nonlinear Programming, Athena Scientific, Belmont, MA.

Bertsekas, D. P., 1995c. "An Auction Algorithm for the Max-Flow Problem," J. of Optimization Theory and Applications, Vol. 87, pp. 69-101.

Bertsekas, D. P., 1996. "Thevenin Decomposition and Network Optimization," J. of Optimization Theory and Applications, Vol. 89, pp. 1-15.

Bertsekas, D. P., and Castañon, D. A., 1989. "The Auction Algorithm for Transportation Problems," Annals of Operations Research, Vol. 20, pp. 67-96.

Bertsekas, D. P., and Castañon, D. A., 1991. "Parallel Synchronous and

Asynchronous Implementations of the Auction Algorithm," Parallel Computing, Vol. 17, pp. 707-732.

Bertsekas, D. P., and Castañon, D. A., 1992. "A Forward/Reverse Auction Algorithm for Asymmetric Assignment Problems," Computational Optimization and Applications, Vol. 1, pp. 277-297.

Bertsekas, D. P., and Castañon, D. A., 1993a. "Asynchronous Hungarian Methods for the Assignment Problem," ORSA J. on Computing, Vol. 5, pp. 261-274.

Bertsekas, D. P., and Castañon, D. A., 1993b. "Parallel Primal-Dual Methods for the Minimum Cost Flow Problem," Computational Optimization and Applications, Vol. 2, pp. 317-336.

Bertsekas, D. P., and Castañon, D. A., 1993c. "A Generic Auction Algorithm for the Minimum Cost Network Flow Problem," Computational Optimization and Applications, Vol. 2, pp. 229-260.

Bertsekas, D. P., and Castañon, D. A., 1998. "Solving Stochastic Scheduling Problems Using Rollout Algorithms," Lab. for Information and Decision Systems Report P-12413, M.I.T., Cambridge, MA.

Bertsekas, D. P., Castañon, D. A., Eckstein, J., and Zenios, S., 1995. "Parallel Computing in Network Optimization," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 331-399.

Bertsekas, D. P., Castañon, D. A., and Tsaknakis, H., 1993. "Reverse Auction and the Solution of Inequality Constrained Assignment Problems," SIAM J. on Optimization, Vol. 3, pp. 268-299.

Bertsekas, D. P., and El Baz, D., 1987. "Distributed Asynchronous Relaxation Methods for Convex Network Flow Problems," SIAM J. on Control and Optimization, Vol. 25, pp. 74-85.

Bertsekas, D. P., and Eckstein, J., 1987. "Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems," Proc. of IFAC '87, Munich, Germany.

Bertsekas, D. P., and Eckstein, J., 1988. "Dual Coordinate Step Methods for Linear Network Flow Problems," Math. Programming, Series B, Vol. 42, pp. 203-243.

Bertsekas, D. P., and Gafni, E. M., 1982. "Projection Methods for Variational Inequalities with Application to the Traffic Assignment Problem," Math. Progr. Studies, Vol. 17, North-Holland, Amsterdam, pp. 139-159.

Bertsekas, D. P., and Gafni, E. M., 1983. "Projected Newton Methods and Optimization of Multicommodity Flows," IEEE Trans. on Auto. Control, Vol. 28, pp. 1090-1096.

Bertsekas, D. P., Gafni, E. M., and Gallager, R. G., 1984. "Second Derivative Algorithms for Minimum Delay Distributed Routing in Networks," IEEE Trans. on Communications, Vol. 32, pp. 911-919.

Bertsekas, D. P., and Gallager, R. G., 1992. Data Networks, (2nd Ed.), Prentice-Hall, Englewood Cliffs, N. J.

Bertsekas, D. P., Guerriero, F., and Musmanno, R., 1996. "Parallel Asynchronous Label Correcting Methods for Shortest Paths," J. of Optimization Theory and Applications, Vol. 88, pp. 297-320.

Bertsekas, D. P., Hosein, P., and Tseng, P., 1987. "Relaxation Methods for Network Flow Problems with Convex Arc Costs," SIAM J. on Control and Optimization, Vol. 25, pp. 1219-1243.

Bertsekas, D. P, and Mitter, S. K., 1971. "Steepest Descent for Optimization Problems with Nondifferentiable Cost Functionals," Proc. 5th Annual Princeton Confer. Inform. Sci. Systems, Princeton, N. J., pp. 347-351.

Bertsekas, D. P., and Mitter, S. K., 1973. "Descent Numerical Methods for Optimization Problems with Nondifferentiable Cost Functions," SIAM J. on Control, Vol. 11, pp. 637-652.

Bertsekas, D. P., Pallottino, S., and Scutellà, M. G., 1995. "Polynomial Auction Algorithms for Shortest Paths," Computational Optimization and Applications, Vol. 4, pp. 99-125.

Bertsekas, D. P., Polymenakos, L. C., and Tseng, P., 1997a. "An ϵ-Relaxation Method for Separable Convex Cost Network Flow Problems," SIAM J. on Optimization, Vol. 7, pp. 853-870.

Bertsekas, D. P., Polymenakos, L. C., and Tseng, P., 1997b. "Epsilon-Relaxation and Auction Methods for Separable Convex Cost Network Flow Problems," in Network Optimization, Pardalos, P. M., Hearn, D. W., and Hager, W. W. (eds.), Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, N. Y., pp. 103-126.

Bertsekas, D. P., and Tseng, P., 1988a. "Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems," Operations Research, Vol. 36, pp. 93-114.

Bertsekas, D. P., and Tseng, P., 1988b. "RELAX: A Computer Code for Minimum Cost Network Flow Problems," Annals of Operations Research, Vol. 13, pp. 127-190.

Bertsekas, D. P., and Tseng, P., 1990. "RELAXT-III: A New and Improved Version of the RELAX Code," Lab. for Information and Decision Systems Report P-1990, M.I.T., Cambridge, MA.

Bertsekas, D. P., and Tseng, P., 1994. "RELAX-IV: A Faster Version of the RELAX Code for Solving Minimum Cost Flow Problems," Laboratory for Information and Decision Systems Report P-2276, M.I.T., Cambridge, MA.

Bertsekas, D. P., and Tsitsiklis, J. N., 1989. Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N. J. (republished in 1997 by Athena Scientific, Belmont, MA).

Bertsekas, D. P., and Tsitsiklis, J. N., 1996. Neuro-Dynamic Programming, Athena Scientific, Belmont, MA.

Bertsekas, D. P., Tsitsiklis, J. N., and Wu, C., 1997. "Rollout Algorithms for Combinatorial Optimization," Heuristics, Vol. 3, pp. 245-262.

Bertsimas, D., and Tsitsiklis, J. N., 1993. "Simulated Annealing," Stat. Sci., Vol. 8, pp. 10-15.

Bertsimas, D., and Tsitsiklis, J. N., 1997. Introduction to Linear Optimization, Athena Scientific, Belmont, MA.

Birkhoff, G., and Diaz, J. B., 1956. "Nonlinear Network Problems," Quart. Appl. Math., Vol. 13, pp. 431-444.

Bland, R. G., and Jensen, D. L., 1985. "On the Computational Behavior of a Polynomial-Time Network Flow Algorithm," Tech. Report 661, School of Operations Research and Industrial Engineering, Cornell University.

Blackman, S. S., 1986. Multi-Target Tracking with Radar Applications, Artech House, Dehdam, MA.

Bogart, K. P., 1990. Introductory Combinatorics, Harcourt Brace Jovanovich, Inc., New York, N. Y.

Bradley, G. H., Brown, G. G., and Graves, G. W., 1977. "Design and Implementation of Large-Scale Primal Transshipment Problems," Management Science, Vol. 24, pp. 1-38.

Brannlund, U., 1993. On Relaxation Methods for Nonsmooth Convex Optimization, Doctoral Thesis, Royal Institute of Technology, Stockhorm, Sweden.

Brown, G. G., and McBride, R. D., 1984. "Solving Generalized Networks," Management Science, Vol. 30, pp. 1497-1523.

Burkard, R. E., 1990. "Special Cases of Traveling Salesman Problems and Heuristics," Acta Math. Appl. Sin., Vol. 6, pp. 273-288.

Busacker, R. G., and Gowen, P. J., 1961. "A Procedure for Determining a Family of Minimal-Cost Network Flow Patterns," O.R.O. Technical Report No. 15, Operational Research Office, John Hopkins University, Baltimore, MD.

Busacker, R. G., and Saaty, T. L., 1965. Finite Graphs and Networks: An Introduction with Applications, McGraw-Hill, N. Y.

Cameron, P. J., 1994. Combinatorics: Topics, Techniques, Algorithms, Cambridge Univ. Press, Cambridge, England.

Cantor, D. G., and Gerla, M., 1974. "Optimal Routing in a Packet Switched Computer Network," IEEE Trans. on Computers, Vol. 23, pp. 1062-1069.

Carpaneto, G., Martello, S., and Toth, P., 1988. "Algorithms and Codes for the Assignment Problem," Annals of Operations Research, Vol. 13, pp. 193-223.

Carraresi, P., and Sodini, C., 1986. "An Efficient Algorithm for the Bipartite Matching Problem," Eur. J. Operations Research, Vol. 23, pp. 86-93.

Castañon, D. A., 1990. "Efficient Algorithms for Finding the K Best Paths Through a Trellis," IEEE Trans. on Aerospace and Electronic Systems, Vol. 26, pp. 405-410.

Castañon, D. A., 1993. "Reverse Auction Algorithms for Assignment Problems," in Algorithms for Network Flows and Matching, Johnson, D. S., and McGeoch, C. C. (eds.), American Math. Soc., Providence, RI, pp. 407-429.

Censor, Y., and Zenios, S. A., 1992. "The Proximal Minimization Algorithm with D-Functions," J. Opt. Theory and Appl., Vol. 73, pp. 451-464.

Censor, Y., and Zenios, S. A., 1997. Parallel Optimization: Theory, Algorithms, and Applications, Oxford University Press, N. Y.

Cerny, V., 1985. "A Thermodynamical Approach to the Travelling Salesman Problem: An Efficient Simulation Algorithm," J. Opt. Theory and Applications, Vol. 45, pp. 41-51.

Cerulli, R., De Leone, R., and Piacente, G., 1994. "A Modified Auction Algorithm for the Shortest Path Problem," Optimization Methods and Software, Vol. 4, pp. 209-224.

Cerulli, R., Festa, P., and Raiconi, G., 1997a. "Graph Collapsing in Shortest Path Auction Algorithms," Univ. of Salerno Tech. Report n. 6/97.

Cerulli, R., Festa, P., and Raiconi, G., 1997b. "An Efficient Auction Algorithm for the Shortest Path Problem Using Virtual Source Concept," Univ. of Salerno Tech. Report n. 6/97.

Chajakis, E. D., and Zenios, S. A., 1991. "Synchronous and Asynchronous Implementations of Relaxation Algorithms for Nonlinear Network Optimization," Parallel Computing, Vol. 17, pp. 873-894.

Chen, G., and Teboulle, M., 1993. "Convergence Analysis of a Proximal-Like Minimization Algorithm Using Bregman Functions," SIAM J. on Optimization, Vol. 3, pp. 538-543.

Chen, Z. L., and Powell, W. B., 1997. "A Note on Bertsekas' Small-Label-First Strategy," Networks, Vol. 29, pp. 111-116.

Cheney, E. W., and Goldstein, A. A., 1959. "Newton's Method for Convex Programming and Tchebycheff Approximation," Numer. Math., Vol. I, pp. 253-268.

Cheriyan, J., and Maheshwari, S. N., 1989. "Analysis of Preflow Push Algorithms for Maximum Network Flow," SIAM J. Computing, Vol. 18, pp. 1057-1086.

Cherkasky, R. V., 1977. "Algorithm for Construction of Maximum Flow in Networks with Complexity of $O(V^2\sqrt{E})$ Operations," Mathematical Methods of Solution of Economical Problems, Vol. 7, pp. 112-125.

Christofides, N., 1975. Graph Theory: An Algorithmic Approach, Academic Press, N. Y.

Chvatal, V., 1983. Linear Programming, W. H. Freeman and Co., N. Y.

Connors, D. P., and Kumar, P. R., 1989. "Simulated Annealing Type Markov Chains and their Order Balance Equations," SIAM J. on Control and Optimization, Vol. 27, pp. 1440-1461.

Cook, W., Cunningham, W., Pulleyblank, W., and Schrijver, A., 1998. Combinatorial Optimization, Wiley, N. Y.

Cornuejols, G., Fonlupt, J., and Naddef, D., 1985. "The Traveling Salesman Problem on a Graph and Some Related Polyhedra," Math. Programming, Vol. 33, pp. 1-27.

Cottle, R. W., and Pang, J. S., 1982. "On the Convergence of a Block Successive Over-Relaxation Method for a Class of Linear Complementarity Problems," Math. Progr. Studies, Vol. 17, pp. 126-138.

Croes, G. A., 1958. "A Method for Solving Traveling Salesman Problems," Operations Research, Vol. 6, pp. 791-812.

Cunningham, W. H., 1976. "A Network Simplex Method," Math. Programming, Vol. 4, pp. 105-116.

Cunningham, W. H., 1979. "Theoretical Properties of the Network Simplex Method," Math. of Operations Research, Vol. 11, pp. 196-208.

Dafermos, S., 1980. "Traffic Equilibrium and Variational Inequalities," Transportation Science, Vol. 14, pp. 42-54.

Dafermos, S., 1982. "Relaxation Algorithms for the General Asymmetric Traffic Equilibrium Problem," Transportation Science, Vol. 16, pp. 231-240.

Dafermos, S., and Sparrow, F. T., 1969. "The Traffic Assignment Problem for a General Network," J. Res. Nat. Bureau of Standards, Vol. 73B, pp. 91-118.

Dantzig, G. B., 1951. "Application of the Simplex Method to a Transportation Problem," in Activity Analysis of Production and Allocation, T. C.

Koopmans (ed.), Wiley, N. Y., pp. 359-373.

Dantzig, G. B., 1960. "On the Shortest Route Problem Through a Network," Management Science, Vol. 6, pp. 187-190.

Dantzig, G. B., 1963. Linear Programming and Extensions, Princeton Univ. Press, Princeton, N. J.

Dantzig, G. B., 1967. "All Shortest Routes in a Graph," in Theory of Graphs, P. Rosenthier (ed.), Gordan and Breach, N. Y., pp. 92-92.

Dantzig, G. B., and Fulkerson, D. R., 1956. "On the Max-Flow Min-Cut Theorem of Networks," in Linear Inequalities and Related Systems, Kuhn, H. W., and Tucker, A. W. (eds.), Annals of Mathematics Study 38, Princeton Univ. Press, pp. 215-221.

Dantzig, G. B., and Wolfe, P., 1960. "Decomposition Principle for Linear Programs," Operations Research, Vol. 8, pp. 101-111.

Dantzig, G. B., Fulkerson, D. R., and Johnson, S. M., 1954. "Solution of a Large-Scale Traveling-Salesman Problem," Operations Research, Vol. 2, pp. 393-410.

De Leone, R., Meyer, R. R., and Zakarian, A., 1995. "An ϵ-Relaxation Algorithm for Convex Network Flow Problems," Computer Sciences Department Technical Report, University of Wisconsin, Madison, WI.

Dembo, R. S., 1987. "A Primal Truncated Newton Algorithm for Large-Scale Unconstrained Optimization," Math. Programming Studies, Vol. 31, pp. 43-72.

Dembo, R. S., and Klincewicz, J. G., 1981. "A Scaled Reduced Gradient Algorithm for Network Flow Problems with Convex Separable Costs," Math. Programming Studies, Vol. 15, pp. 125-147.

Dembo, R. S., and Tulowitzki, U., 1988. "Computing Equilibria on Large Multicommodity Networks: An Application of Truncated Quadratic Programming Algorithms," Networks, Vol. 18, pp. 273-284.

Denardo, E. V., and Fox, B. L., 1979. "Shortest-Route Methods: 1. Reaching, Pruning and Buckets," Operations Research, Vol. 27, pp. 161-186.

Dennis, J. B., 1959. Mathematical Programming and Electical Circuits, Technology Press of M.I.T., Cambridge, MA.

Deo, N., and Kumar, N., 1997. "Computation of Constrained Spanning Trees: A Unified Approach," in Network Optimization, Pardalos, P. M., Hearn, D. W., and Hager, W. W. (eds.), Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, N. Y., pp. 194-220.

Deo, N., and Pang, C., 1984. "Shortest Path Algorithms: Taxonomy and Annotation," Networks, Vol. 14, pp. 275-323.

Derigs, U., 1985. "The Shortest Augmenting Path Method for Solving Assignment Problems – Motivation and Computational Experience," Annals of Operations Research, Vol. 4, pp. 57-102.

Derigs, U., and Meier, W., 1989. "Implementing Goldberg's Max-Flow Algorithm – A Computational Investigation," Zeitschrif fur Operations Research, Vol. 33, pp. 383-403.

Desrosiers, J., Dumas, Y., Solomon, M. M., and Soumis, F., 1995. "Time Constrained Routing and Scheduling," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 8, North-Holland, Amsterdam, pp. 35-139.

Dial, R. B., 1969. "Algorithm 360: Shortest Path Forest with Topological Ordering," Comm. ACM, Vol. 12, pp. 632-633.

Dial, R., Glover, F., Karney, D., and Klingman, D., 1979. "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees," Networks, Vol. 9, pp. 215-248.

Dijkstra, E., 1959. "A Note on Two Problems in Connexion with Graphs," Numer. Math., Vol. 1, pp. 269-271.

Dinic, E. A., 1970. "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," Soviet Math. Doklady, Vol. 11, pp. 1277-1280.

Dreyfus, S. E., 1969. "An Appraisal of Some Shortest-Path Algorithms," Operations Research, Vol. 17, pp. 395-412.

Duffin, R. J., 1947. "Nonlinear Networks. IIa," Bull. Amer. Math. Soc., Vol. 53, pp. 963-971.

Eastman, W. L., 1958. Linear Programming with Pattern Constraints, Ph.D. Thesis, Harvard University, Cambridge, MA.

Eckstein, J., 1994. "Nonlinear Proximal Point Algorithms Using Bregman Functions, with Applications to Convex Programming," Math. of Operations Research, Vol. 18, pp. 202-226.

Edmonds, J., 1965. "Paths, Trees, and Flowers," Canadian J. of Math., Vol. 17, pp. 449-467.

Edmonds, J., and Johnson, E. L., 1973. "Matching, Euler Tours, and the Chinese Postman," Math. Programming, Vol. 5, pp. 88-124.

Edmonds, J., and Karp, R. M., 1972. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," J. ACM, Vol. 19, pp. 248-264.

Eiselt, H. A., Gendreau, M., and Laporte, G., 1995a. "Arc Routing Problems, Part 1: The Chinese Postman Problem," Operations Research, Vol. 43, pp. 231-242.

Eiselt, H. A., Gendreau, M., and Laporte, G., 1995b. "Arc Routing Problems, Part 2: The Rural Postman Problem," Operations Research, Vol. 43, pp. 399-414.

Elias, P., Feinstein, A., and Shannon, C. E., 1956. "Note on Maximum Flow Through a Network," IRE Trans. Info. Theory, Vol. IT-2, pp. 117-119.

Egervary, J., 1931. "Matrixok Kombinatoricus Tulajonsagairol," Mat. Es Fiz. Lapok, Vol. 38, pp. 16-28.

El Baz, D., 1989. "A Computational Experience with Distributed Asynchronous Iterative Methods for Convex Network Flow Problems," Proc. of the 28th IEEE Conference on Decision and Control, Tampa, Fl., pp. 590-591.

El Baz, D., 1996. "Asynchronous Gradient Algorithms for a Class of Convex Separable Network Flow Problems," Computational Optimization and Applications, Vol. 5, pp. 187-205.

El Baz, D., Spiteri, P., Miellou, J. C., and Gazen, D., 1996. "Asynchronous Iterative Algorithms with Flexible Communication for Nonlinear Network Flow Problems," J. of Parallel and Distributed Computing, Vol. 38, pp. 1-15.

Elam, J., Glover, F., and Klingman, D., 1979. "A Strongly Convergent Primal Simplex Algorithm for Generalized Networks," Math. of Operations Research, Vol. 4, pp. 39-59.

Elmaghraby, S. E., 1978. Activity Networks: Project Planning and Control by Network Models, Wiley, N. Y.

Elzinga, J., and Moore, T. G., 1975. "A Central Cutting Plane Algorithm for the Convex Programming Problem," Math. Programming, Vol. 8, pp. 134-145.

Engquist, M., 1982. "A Successive Shortest Path Algorithm for the Assignment Problem," INFOR, Vol. 20, pp. 370-384.

Ephremides, A., 1986. "The Routing Problem in Computer Networks," in Communication and Networks, Blake, I. F., and Poor, H. V. (eds.), Springer-Verlag, N. Y., pp. 299-325.

Ephremides, A., and Verdu, S., 1989. "Control and Optimization Methods in Communication Network Problems," IEEE Trans. on Automatic Control, Vol. 34, pp. 930-942.

Esau, L. R., and Williams, K. C., 1966. "On Teleprocessing System Design. A Method for Approximating the Optimal Network," IBM System J., Vol. 5, pp. 142-147.

Escudero, L. F., 1985. "Performance Evaluation of Independent Superbasic Sets on Nonlinear Replicated Networks," Eur. J. Operations Research, Vol.

23, pp. 343-355.

Everett, H., 1963. "Generalized Lagrange Multiplier Method for Solving Problems of Optimal Allocation of Resources," Operations Research, Vol. 11, pp. 399-417.

Falcone, M., 1987. "A Numerical Approach to the Infinite Horizon Problem of Deterministic Control Theory," Appl. Math. Opt., Vol. 15, pp. 1-13.

Federgruen, A., and Simchi-Levi, D., 1995. "Analysis of Vehicle and Inventory-Routing Problems," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 8, North-Holland, Amsterdam, pp. 297-373.

Ferris, M. C., 1991. "Finite Termination of the Proximal Point Algorithm," Math. Programming, Vol. 50, pp. 359-366.

Fisher, M., 1995. "Vehicle Routing," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 8, North-Holland, Amsterdam, pp. 1-33.

Florian, M., Guélat, J., and Spiess, H., 1987. "An Efficient Implementation of the "PARTAN" Variant of the Linear Approximation Method for the Network Equilibrium Problem," Networks, Vol. 17, pp. 319-339.

Florian, M. S., and Hearn, D., 1995. "Network Equilibrium Models and Algorithms," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 8, North-Holland, Amsterdam, pp. 485-550.

Florian, M. S., and Nguyen, S., 1974. "A Method for Computing Network Equilibrium with Elastic Demands," Transportation Science, Vol. 8, pp. 321-332.

Florian, M. S., and Nguyen, S., 1976. "An Application and Validation of Equilibrium Trip Assignment Methods," Transportation Science, Vol. 10, pp. 374-390.

Florian, M. S., Nguyen, S., and Pallottino, S., 1981. "A Dual Simplex Algorithm for Finding All Shortest Paths," Networks, Vol. 11, pp. 367-378.

Floudas, C. A., 1995. Nonlinear and Mixed-Integer Optimization: Fundamentals and Applications, Oxford University Press, N. Y.

Floyd, R. W., 1962. "Algorithm 97: Shortest Path," Comm. ACM, Vol. 5, pp. 345.

Ford, L. R., Jr., 1956. "Network Flow Theory," Report P-923, The Rand Corporation, Santa Monica, CA.

Ford, L. R., Jr., and Fulkerson, D. R., 1956a. "Solving the Transportation Problem," Management Science, Vol. 3, pp. 24-32.

Ford, L. R., Jr., and Fulkerson, D. R., 1956b. "Maximal Flow Through a Network," Can. J. of Math., Vol. 8, pp. 339-404.

Ford, L. R., Jr., and Fulkerson, D. R., 1957. "A Primal-Dual Algorithm for the Capacitated Hitchcock Problem," Naval Res. Logist. Quart., Vol. 4, pp. 47-54.

Ford, L. R., Jr., and Fulkerson, D. R., 1962. Flows in Networks, Princeton Univ. Press, Princeton, N. J.

Fox, B. L., 1993. "Integrating and Accelerating Tabu Search, Simulated Annealing, and Genetic Algorithms," Annals of Operations Research, Vol. 41, pp. 47-67.

Fox, B. L., 1995. "Faster Simulated Annealing," SIAM J. Optimization, Vol. 41, pp. 47-67.

Frank, H., and Frisch, I. T., 1970. Communication, Transmission, and Transportation Networks, Addison-Wesley, Reading, MA.

Fratta, L., Gerla, M., and Kleinrock, L., 1973. "The Flow-Deviation Method: An Approach to Store-and-Forward Computer Communication Network Design," Networks, Vol. 3, pp. 97-133.

Fredman, M. L., and Tarjan, R. E., 1984. "Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms," Proc. 25th Annual Symp. on Found. of Comp. Sci., pp. 338-346.

Fukushima, M., 1984a. "A Modified Frank-Wolfe Algorithm for Solving the Traffic Assignment Problem," Transportation Research, Vol. 18B, pp. 169–177.

Fukushima, M., 1984b. "On the Dual Approach to the Traffic Assignment Problem," Transportation Research, Vol. 18B, pp. 235-245.

Fukushima, M., 1992. "Equivalent Differentiable Optimization Problems and Descent Methods for Asymmetric Variational Inequalities," Math. Programming, Vol. 53, pp. 99-110.

Fulkerson, D. R., 1961. "An Out-of-Kilter Method for Minimal Cost Flow Problems," SIAM J. Appl. Math., Vol. 9, pp. 18-27.

Fulkerson, D. R., and Dantzig, G. B., 1955. "Computation of Maximum Flow in Networks," Naval Res. Log. Quart., Vol. 2, pp. 277-283.

Gafni, E. M., 1979. "Convergence of a Routing Algorithm," M.S. Thesis, Dept. of Electrical Engineering, Univ. of Illinois, Urbana, Ill.

Gafni, E. M., and Bertsekas, D. P., 1984. "Two-Metric Projection Methods for Constrained Optimization," SIAM J. on Control and Optimization, Vol. 22, pp. 936-964.

Gale, D., 1957. "A Theorem of Flows in Networks," Pacific J. Math., Vol. 7, pp. 1073-1082.

Gale, D., Kuhn, H. W., and Tucker, A. W., 1951. "Linear Programming and the Theory of Games," in Activity Analysis of Production and Allocation, T. C. Koopmans (ed.), Wiley, N. Y.

Galil, Z., 1980. "$O\big(V^{5/3}E^{2/3}\big)$ Algorithm for the Maximum Flow Problem," Acta Informatica, Vol. 14, pp. 221-242.

Galil, Z., and Naamad, A., 1980. "$O\big(VE\log^2 V\big)$ Algorithm for the Maximum Flow Problem," J. of Comput. Sys. Sci., Vol. 21, pp. 203-217.

Gallager, R. G., 1977. "A Minimum Delay Routing Algorithm Using Distributed Computation," IEEE Trans. on Communications, Vol. 23, pp. 73-85.

Gallo, G. S., and Pallottino, S., 1982. "A New Algorithm to Find the Shortest Paths Between All Pairs of Nodes," Discrete Applied Mathematics, Vol. 4, pp. 23-35.

Gallo, G. S., and Pallottino, S., 1986. "Shortest Path Methods: A Unified Approach," Math. Programming Studies, Vol. 26, pp. 38-64.

Gallo, G. S., and Pallottino, S., 1988. "Shortest Path Algorithms," Annals of Operations Research, Vol. 7, pp. 3-79.

Garey, M. R., and Johnson, D. S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., San Francisco, Ca.

Gartner, N. H., 1980a. "Optimal Traffic Assignment with Elastic Demands: A Review. Part I. Analysis Framework," Transportation Science, Vol. 14, pp. 174-191.

Gartner, N. H., 1980b. "Optimal Traffic Assignment with Elastic Demands: A Review. Part II. Algorithmic Approaches," Transportation Science, Vol. 14, pp. 192-208.

Gavish, B., Schweitzer, P., and Shlifer, E., 1977. "The Zero Pivot Phenomenon in Transportation Problems and its Computational Implications," Math. Programming, Vol. 12, pp. 226-240.

Gelfand, S. B., and Mitter, S. K., 1989. "Simulated Annealing with Noisy or Imprecise Measurements," J. Opt. Theory and Applications, Vol. 69, pp. 49-62.

Geoffrion, A. M., 1970. "Elements of Large-Scale Mathematical Programming, I, II," Management Science, Vol. 16, pp. 652-675, 676-691.

Geoffrion, A. M., 1974. "Lagrangian Relaxation for Integer Programming," Math. Programming Studies, Vol. 2, pp. 82-114.

Gerards, A. M. H., 1995. "Matching," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 135-224.

Gibby, D., Glover, F., Klingman, D., and Mead, M., 1983. "A Comparison of Pivot Selection Rules for Primal Simplex Based Network Codes," Operations Research Letters, Vol. 2, pp. 199-202.

Gill, P. E., Murray, W., and Wright, M. H., 1981. Practical Optimization, Academic Press, N. Y.

Gilmore, P. C., Lawler, E. L., and Shmoys, D. B., 1985. "Well-Solved Special Cases," in The Traveling Salesman Problem, Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B. (eds.), Wiley, N. Y., pp. 87-143.

Glover, F., 1986. "Future Paths for Integer Programming and Links to Artificial Intelligence," Computers and Operations Research, Vol. 13, pp. 533-549.

Glover, F., 1989. "Tabu Search: Part I," ORSA J. on Computing, Vol. 1, pp. 190-206.

Glover, F., 1990. "Tabu Search: Part II," ORSA J. on Computing, Vol. 2, pp. 4-32.

Glover, F., Glover, R., and Klingman, D., 1986. "The Threshold Shortest Path Algorithm," Math. Programming Studies, Vol. 26, pp. 12-37.

Glover, F., Glover, R., and Klingman, D., 1986. "Threshold Assignment Algorithm," Math. Programming Studies, Vol. 26, pp. 12-37.

Glover, F., Karney, D., and Klingman, D., 1974. "Implementation and Computational Comparisons of Primal, Dual, and Primal-Dual Computer Codes for Minimum Cost Network Flow Problem," Networks, Vol. 4, pp. 191-212.

Glover, F., Karney, D., Klingman, D., and Napier, A., 1974. "A Computation Study on Start Procedures, Basis Change Criteria, and Solution Algorithms for Transportation Problems," Management Science, Vol. 20, pp. 793-819.

Glover, F., Klingman, D., Mote, J., and Whitman, D., 1984. "A Primal Simplex Variant for the Maximum Flow Problem," Naval Res. Logist. Quart., Vol. 31, pp. 41-61.

Glover, F., Klingman, D., and Phillips, N., 1985. "A New Polynomially Bounded Shortest Path Algorithm," Operations Research, Vol. 33, pp. 65-73.

Glover, F., Klingman, D., and Phillips, N., 1992. Network Models in Optimization and Their Applications in Practice, Wiley, N. Y.

Glover, F., Klingman, D., Phillips, N., and Schneider, R. F., 1985. "New Polynomial Shortest Path Algorithms and Their Computational Attributes," Management Science, Vol. 31, pp. 1106-1128.

Glover, F., Klingman, D., and Stutz, J., 1973. "Extension of the Augmented Predecessor Index Method to Generalized Netork Problems," Transportation Science, Vol. 7, pp. 377-384.

Glover, F., Klingman, D., and Stutz, J., 1974. "Augmented Threaded Index Method for Network Optimization," INFOR, Vol. 12, pp. 293-298.

Glover, F., and Laguna, M., 1997. Tabu Search, Kluwer, Boston.

Glover, F., Taillard, E., and de Verra, D., 1993. "A User's Guide to Tabu Search," Annals of Operations Research, Vol. 41, pp. 3-28.

Goffin, J. L., 1977. "On Convergence Rates of Subgradient Optimization Methods," Math. Programming, Vol. 13, pp. 329-347.

Goffin, J. L., Haurie, A., and Vial, J. P., 1992. "Decomposition and Nondifferentiable Optimization with the Projective Algorithm," Management Science, Vol. 38, pp. 284-302.

Goffin, J. L., and Kiwiel, K. C, 1996. 'Convergence of a Simple Subgradient Level Method," Unpublished Report, to appear in Math. Programming.

Goffin, J. L., Luo, Z.-Q., and Ye, Y., 1993. "On the Complexity of a Column Generation Algorithm for Convex or Quasiconvex Feasibility Problems," in Large Scale Optimization: State of the Art, Hager, W. W., Hearn, D. W., and Pardalos, P. M. (eds.), Kluwer.

Goffin, J. L., Luo, Z.-Q., and Ye, Y., 1996. "Further Complexity Analysis of a Primal-Dual Column Generation Algorithm for Convex or Quasiconvex Feasibility Problems," SIAM J. on Optimization, Vol. 6, pp. 638-652.

Goffin, J. L., and Vial, J. P., 1990. "Cutting Planes and Column Generation Techniques with the Projective Algorithm," J. Opt. Th. and Appl., Vol. 65, pp. 409-429.

Goldberg, A. V., 1987. "Efficient Graph Algorithms for Sequential and Parallel Computers," Tech. Report TR-374, Laboratory for Computer Science, M.I.T., Cambridge, MA.

Goldberg, A. V., 1993. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm," Proc. 3rd Integer Progr. and Combinatorial Optimization Conf., pp. 251-266.

Goldberg, A. V., and Tarjan, R. E., 1986. "A New Approach to the Maximum Flow Problem," Proc. 18th ACM STOC, pp. 136-146.

Goldberg, A. V., and Tarjan, R. E., 1990. "Solving Minimum Cost Flow Problems by Successive Approximation," Math. of Operations Research, Vol. 15, pp. 430-466.

Goldberg, D. E., 1989. Genetic Algorithms in Search, Optimization, and Machine Learning, Addison Wesley, Reading, MA.

Goldfarb, D., 1985. "Efficient Dual Simplex Algorithms for the Assignment Problem," Math. Programming, Vol. 33, pp. 187-203.

Goldfarb, D., and Hao, J., 1990. "A Primal Simplex Algorithm that Solves the Maximum Flow Problem in at Most nm Pivots and $O(n^2m)$ Time," Math. Programming, Vol. 47, pp. 353-365.

Goldfarb, D., Hao, J., and Kai, S., 1990a. "Anti-Stalling Pivot Rules for the Network Simplex Algorithm," Networks, Vol. 20, pp. 79-91.

Goldfarb, D., Hao, J., and Kai, S., 1990b. "Efficient Shortest Path Simplex Algorithms," Operations Research, Vol. 38, pp. 624-628.

Goldfarb, D., and Reid, J. K., 1977. "A Practicable Steepest Edge Simplex Algorithm," Math. Programming, Vol. 12, pp. 361-371.

Goldstein, A. A., 1967. Constructive Real Analysis, Harper and Row, N. Y.

Gondran, M., and Minoux, M., 1984. Graphs and Algorithms, Wiley, N. Y.

Gonzalez, R., and Rofman, E., 1985. "On Deterministic Control Problems: An Approximation Procedure for the Optimal Cost, Parts I, II," SIAM J. on Control and Optimization, Vol. 23, pp. 242-285.

Graham, R. L., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G., 1979. "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," Annals of Discrete Math., Vol. 5, pp. 287-326.

Grötschel, M., Monma, C. L., and Stoer, M., 1995. "Design of Survivable Networks," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 617-672.

Grötschel, M., and Padberg, M. W., 1985. "Polyhedral Theory," in The Traveling Salesman Problem, Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B. (eds.), Wiley, N. Y., pp. 251-305.

Guerriero, F., Lacagnina, V., Musmanno, R., and Pecorella, A., 1996. "Efficient Node Selection Strategies in Label-Correcting Methods for the K Shortest Paths Problem," Technical Report PARCOLAB No. 6/96, Department of Electronics, Informatics and Systems, University of Calabria.

Guler, O., 1992. "New Proximal Point Algorithms for Convex Minimization," SIAM J. on Optimization, Vol. 2, pp. 649-664.

Hajek, B., 1988. "Cooling Schedules for Optimal Annealing," Math. of Operations Research, Vol. 13, pp. 311-329.

Hall, M., Jr., 1956. "An Algorithm for Distinct Representatives," Amer. Math. Monthly, Vol. 51, pp. 716-717.

Hansen, P., 1986. "The Steepest Ascent Mildest Descent Heuristic for Combinatorial Optimization," Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy.

Hearn, D. W., and Lawphongpanich, S., 1990. "A Dual Ascent Algorithm for Traffic Assignment Problems," Transportation Research, Vol. 24B, pp. 423-430.

Hearn, D. W., Lawphongpanish, S., and Nguyen, S., 1984. "Convex Programming Formulation of the Asymmetric Traffic Assignment Problem," Transportation Research, Vol. 18B, pp. 357-365.

Hearn, D. W., Lawphongpanish, S., and Ventura, J. A., 1985. "Finiteness in Restricted Simplicial Decomposition," Operations Research Letters, Vol. 4, pp. 125-130.

Hearn, D. W., Lawphongpanish, S., and Ventura, J. A., 1987. "Restricted Simplicial Decomposition: Computation and Extensions," Math. Programming Studies, Vol. 31, pp. 99-118.

Held, M., and Karp, R. M., 1970. "The Traveling Salesman Problem and Minimum Spanning Trees," Operations Research, Vol. 18, pp. 1138-1162.

Held, M., and Karp, R. M., 1971. "The Traveling Salesman Problem and Minimum Spanning Trees: Part II," Math. Programming, Vol. 1, pp. 6-25.

Helgason, R. V., and Kennington, J. L., 1977. "An Efficient Procedure for Implementing a Dual-Simplex Network Flow Algorithm," AIIE Transactions, Vol. 9, pp. 63-68.

Helgason, R. V., and Kennington, J. L., 1995. "Primal-Simplex Algorithms for Minimum Cost Network Flows," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 85-133.

Helgason, R. V., Kennington, J. L., and Stewart, B. D., 1993. "The One-to-One Shortest-Path Problem: An Empirical Analysis with the Two-Tree Dijkstra Algorithm," Computational Optimization and Applications, Vol. 1, pp. 47-75.

Hiriart-Urruty, J.-B., and Lemarechal, C., 1993. Convex Analysis and Minimization Algorithms, Vols. I and II, Springer-Verlag, Berlin and N. Y.

Hochbaum, D. S., and Shantikumar, J. G., 1990. "Convex Separable Optimization is not Much Harder than Linear Optimization," J. ACM, Vol. 37, pp. 843-862.

Hoffman, A. J., 1960. "Some Recent Applications of the Theory of Linear Inequalities to Extremal Combinatorial Analysis," Proc. Symp. Appl.

Math., Vol. 10, pp. 113-128.

Hoffman, A. J., and Kuhn, H. W., 1956. "Systems of Distinct Representatives and Linear Programming," Amer. Math. Monthly, Vol. 63, pp. 455-460.

Hoffman, K., and Kunze, R., 1971. Linear Algebra, Prentice-Hall, Englewood Cliffs, N. J.

Holloway, C. A., 1974. "An Extension of the Frank and Wolfe Method of Feasible Directions," Math. Programming, Vol. 6, pp. 14-27.

Hopcroft, J. E., and Karp, R. M., 1973. "A $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs," SIAM J. on Computing, Vol. 2, pp. 225-231.

Horst, R., Pardalos, P. M., and Thoai, N. V., 1995. Introduction to Global Optimization, Kluwer Academic Publishers, N. Y.

Hu, T. C., 1969. Integer Programming and Network Flows, Addison-Wesley, Reading, MA.

Hung, M., 1983. "A Polynomial Simplex Method for the Assignment Problem," Operations Research, Vol. 31, pp. 595-600.

Ibaraki, T., and Katoh, N., 1988. Resource Allocation Problems: Algorithmic Approaches, M.I.T. Press, Cambridge, MA.

Iri, M., 1969. Network Flows, Transportation, and Scheduling, Academic Press, N. Y.

Iusem, A. N., Svaiter, B., and Teboulle, M., 1994. "Entropy-Like Proximal Methods in Convex Programming," Math. Operations Research, Vol. 19, pp. 790-814.

Jensen, P. A., and Barnes, J. W., 1980. Network Flow Programming, Wiley, N. Y.

Jewell, W. S., 1962. "Optimal Flow Through Networks with Gains," Operations Research, Vol. 10, pp. 476-499.

Johnson, D. B., 1977. "Efficient Algorithms for Shortest Paths in Sparse Networks," J. ACM, Vol. 24, pp. 1-13.

Johnson, D. S., and Papadimitriou, C. H., 1985. "Computational Complexity," in The Traveling Salesman Problem, Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B. (eds.), Wiley, N. Y., pp. 37-85.

Johnson, D. S., and McGeoch, L., 1997. "The Traveling Salesman Problem: A Case Study," in Local Search in Combinatorial Optimization, Aarts, E., and Lenstra, J. K. (eds.), Wiley, N. Y.

Johnson, E. L., 1966. "Networks and Basic Solutions," Operations Research, Vol. 14, pp. 619-624.

Johnson, E. L., 1972. "On Shortest Paths and Sorting," Proc. 25th ACM Annual Conference, pp. 510-517.

Jonker, R., and Volgenant, A., 1986. "Improving the Hungarian Assignment Algorithm," Operations Research Letters, Vol. 5, pp. 171-175.

Jonker, R., and Volgenant, A., 1987. "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems," Computing, Vol. 38, pp. 325-340.

Junger, M., Reinelt, G., and Rinaldi, G., 1995. "The Traveling Salesman Problem," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 7, North-Holland, Amsterdam, pp. 225-330.

Karzanov, A. V., 1974. "Determining the Maximal Flow in a Network with the Method of Preflows," Soviet Math Dokl., Vol. 15, pp. 1277-1280.

Karzanov, A. V., and McCormick, S. T., 1997. "Polynomial Methods for Separable Convex Optimization in Unimodular Linear Spaces with Applications to Circulations and Co-circulations in Network," SIAM J. on Computing, Vol. 26, pp. 1245-1275.

Kelley, J. E., 1960. "The Cutting-Plane Method for Solving Convex Programs," J. Soc. Indust. Appl. Math., Vol. 8, pp. 703-712.

Kennington, J., and Helgason, R., 1980. Algorithms for Network Programming, Wiley, N. Y.

Kennington, J., and Shalaby, M., 1977. "An Effective Subgradient Procedure for Minimal Cost Multicommodity Flow Problems," Management Science, Vol. 23, pp. 994-1004.

Kernighan, B. W., and Lin, S., 1970. "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Tech. Journal, Vol. 49, pp. 291-307.

Kershenbaum, A., 1981. "A Note on Finding Shortest Path Trees," Networks, Vol. 11, pp. 399-400.

Kershenbaum, A., 1993. Network Design Algorithms, McGraw-Hill, N. Y.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., 1983. "Optimization by Simulated Annealing," Science, Vol. 220, pp. 621-680.

Kiwiel, K. C., 1997a. "Proximal Minimization Methods with Generalized Bregman Functions," SIAM J. on Control and Optimization, Vol. 35, pp. 1142-1168.

Kiwiel, K. C., 1997b. "Efficiency of the Analytic Center Cutting Plane Method for Convex Minimization," SIAM J. on Optimization, Vol. 7, pp. 336-346.

Klee, V., and Minty, G. J., 1972. "How Good is the Simplex Algorithm?," in Inequalities III, O. Shisha (ed.), Academic Press, N. Y., pp. 159-175.

Klein, M., 1967. "A Primal Method for Minimal Cost Flow with Applications to the Assignment and Transportation Problems," Management Science, Vol. 14, pp. 205-220.

Klessig, R. W., 1974. "An Algorithm for Nonlinear Multicommodity Flow Problems," Networks, Vol. 4, pp. 343-355.

Klincewitz, J. C., 1989. "Implementing an Exact Newton Method for Separable Convex Transportation Problems," Networks, Vol. 19, pp. 95-105.

König, D., 1931. "Graphok es Matrixok," Mat. Es Fiz. Lapok, Vol. 38, pp. 116-119.

Korst, J., Aarts, E. H., and Korst, A., 1989. Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing, Wiley, N. Y.

Kortanek, K. O., and No, H., 1993. "A Central Cutting Plane Algorithm for Convex Semi-Infinite Programming Problems," SIAM J. on Optimization, Vol. 3, pp. 901-918.

Kuhn, H. W., 1955. "The Hungarian Method for the Assignment Problem," Naval Research Logistics Quarterly, Vol. 2, pp. 83-97.

Kumar, V., Grama, A., Gupta, A., and Karypis, G., 1994. Introduction to Parallel Computing, Benjamin/Cummings, Redwood City, CA.

Kushner, H. J., 1990. "Numerical Methods for Continuous Control Problems in Continuous Time," SIAM J. on Control and Optimization, Vol. 28, pp. 999-1048.

Kushner, H. J., and Dupuis, P. G., 1992. Numerical Methods for Stochastic Control Problems in Continuous Time, Springer-Verlag, N. Y.

Kwan Mei-Ko, 1962. "Graphic Programming Using Odd or Even Points," Chinese Math., Vol. 1, pp. 273-277.

Lamar, B. W., 1993. "An Improved Branch and Bound Algorithm for Minimum Concave Cost Network Flow Problems," in Network Optimization Problems, Du, D.-Z., and Pardalos, P. M. (eds.), World Scientific Publ., Singapore, pp. 261-287.

Land, A. H., and Doig, A. G., 1960. "An Automatic Method for Solving Discrete Programming Problems," Econometrica, Vol. 28, pp. 497-520.

Larsson, T., and Patricksson, M., 1992. "Simplicial Decomposition with Disaggregated Representation for the Traffic Assignment Problem," Transportation Science, Vol. 26, pp. 4-17.

Lasdon, L. S., 1970. Optimization Theory for Large Systems, Macmillian, N. Y.

Lawphongpanich, S., and Hearn, D., 1984. "Simplicial Decomposition of the Asymmetric Traffic Assignment Problems," Transportation Research, Vol. 18B, pp. 123-133.

Lawphongpanich, S., and Hearn, D. W., 1986. "Restricted Simplicial Decomposition with Application to the Traffic Assignment Problem," Ricerca Operativa, Vol. 38, pp. 97-120.

Lawler, E., 1976. Combinatorial Optimization: Networks and Matroids, Holt, Reinhart, and Winston, N. Y.

Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B., 1985. The Traveling Salesman Problem, Wiley, N. Y.

LeBlanc, L. J., Helgason, R. V., and Boyce, D. E., 1985. "Improved Efficiency of the Frank-Wolfe Algorithm for Convex Network Programs," Transportation Science, Vol. 19, pp. 445–462.

LeBlanc, L. J., Morlok, E. K., and Pierskalla, W. P., 1974. "An Accurate and Efficient Approach to Equilibrium Traffic Assignment on Congested Networks," Transportation Research Record, TRB-National Academy of Sciences, Vol. 491, pp. 12-23.

LeBlanc, L. J., Morlok, E. K., and Pierskalla, W. P., 1975. "An Efficient Approach to Solving the Road Network Equilibrium Traffic Assignment Problem," Transportation Research, Vol. 9, pp. 309-318.

Leventhal, T., Nemhauser, G., and Trotter, Jr., L., 1973. "A Column Generation Algorithm for Optimal Traffic Assignment," Transportation Science, Vol. 7, pp. 168-176.

Lemarechal, C., 1974. "An Algorithm for Minimizing Convex Functions," in Information Processing '74, Rosenfeld, J. L. (ed.), North Holland Publ. Co., Amsterdam, pp. 552-556.

Little, J. D. C., Murty, K. G., Sweeney, D. W., and Karel, C., 1963. "An Algorithm for the Traveling Salesman Problem," Operations Research, Vol. 11, pp. 972-989.

Lovasz, L., and Plummer, M. D., 1985. Matching Theory, North-Holland, Amsterdam.

Luenberger, D. G., 1969. Optimization by Vector Space Methods, Wiley, N. Y.

Luenberger, D. G., 1984. Linear and Nonlinear Programming, Addison-Wesley, Reading, MA.

Luo, Z.-Q., 1997. "Analysis of a Cutting Plane Method that Uses Weighted Analytic Center and Multiple Cuts," SIAM J. of Optimization, Vol. 7, pp.

697-716.

Luo, Z.-Q., and Tseng, P., 1994. "On the Rate of Convergence of a Distributed Asynchronous Routing Algorithm," IEEE Trans. on Automatic Control, Vol. 39, pp. 1123-1129.

Malhotra, V. M., Kumar, M. P., and Maheshwari, S. N., 1978. "An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks," Inform. Process. Lett., Vol. 7, pp. 277-278.

Marcotte, P., 1985. "A New Algorithm for Solving Variational Inequalities with Application to the Traffic Assignment Problem," Math. Programming Studies, Vol. 33, pp. 339-351.

Marcotte, P., and Dussault, J.-P., 1987. "A Note on a Globally Convergent Newton Method for Solving Monotone Variational Inequalities," Operations Research Letters, Vol. 6, pp. 35-42.

Marcotte, P., and Guélat, J., 1988. "Adaptation of a Modified Newton Method for Solving the Asymmetric Traffic Equilibrium Problem," Transportation Science, Vol. 22, pp. 112-124.

Martello, S., and Toth, P., 1990. Knapsack Problems, Wiley, N. Y.

Martinet, B., 1970. "Regularisation d'Inequations Variationnelles par Approximations Successives," Rev. Francaise Inf. Rech. Oper., Vol. 4, pp. 154-159.

McGinnis, L. F., 1983. "Implementation and Testing of a Primal-Dual Algorithm for the Assignment Problem," Operations Research, Vol. 31, pp. 277-291.

Mendelssohn, N. S., and Dulmage, A. L., 1958. "Some Generalizations of Distinct Representatives," Canad. J. Math., Vol. 10, pp. 230-241.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E., 1953. "Equation of State Calculations by Fast Computing Machines," J. of Chemical Physisc, Vol. 21, pp. 1087-1092.

Meyer, R. R., 1979. "Two-Segment Separable Programming," Management Science, Vol. 25, pp. 385-395.

Miller, D., Pekny, J., and Thompson, G. L., 1990. "Solution of Large Dense Transportation Problems Using a Parallel Primal Algorithm," Operations Research Letters, Vol. 9, pp. 319-324.

Minty, G. J., 1957. "A Comment on the Shortest Route Problem," Operations Research, Vol. 5, p. 724.

Minty, G. J., 1960. "Monotone Networks," Proc. Roy. Soc. London, A, Vol. 257, pp. 194-212.

Minieka, E., 1978. Optimization Algorithms for Networks and Graphs, Marcel Dekker, N. Y.

Minoux, M., 1986a. Mathematical Programming: Theory and Algorithms, Wiley, N. Y.

Minoux, M., 1986b. "Solving Integer Minimum Cost Flows with Separable Convex Cost Objective Polynomially," Math. Programming Studies, Vol. 26, pp. 237-239.

Minoux, M., 1989. "Network Synthesis and Optimum Network Design Problems: Models, Solution Methods,and Applications," Networks, Vol. 19, pp. 313-360.

Monma, C. L., and Sheng, D. D., 1986. "Backbone Network Design and Performance Analysis: A Methodology for Packet Switching Networks," IEEE J. Select. Areas Comm., Vol. SAC-4, pp. 946-965.

Mulvey, J., 1978a. "Pivot Strategies for Primal-Simplex Network Codes," J. ACM, Vol. 25, pp. 266-270.

Mulvey, J., 1978b. "Testing a Large-Scale Network Optimization Program," Math. Programming, Vol. 15, pp. 291-314.

Murty, K. G., 1992. Network Programming, Prentice-Hall, Englewood Cliffs, N. J.

Nagurney, A., 1988. "An Equilibration Scheme for the Traffic Assignment Problem with Elastic Demands," Transportation Research, Vol. 22B, pp. 73-79.

Nagurney, A., 1993. Network Economics: A Variational Inequality Approach, Kluwer, Dordrecht, The Netherlands.

Nemhauser, G. L., and Wolsey, L. A., 1988. Integer and Combinatorial Optimization, Wiley, N. Y.

Nesterov, Y., 1995. "Complexity Estimates of Some Cutting Plane Methods Based on Analytic Barrier," Math. Programming, Vol. 69, pp. 149-176.

Nesterov, Y., and Nemirovskii, A., 1994. Interior Point Polynomial Algorithms in Convex Programming, SIAM, Phila., PA.

Nguyen, S., 1974. "An Algorithm for the Traffic Assignment Problem," Transportation Science, Vol. 8, pp. 203-216.

Nicholson, T., 1966. "Finding the Shortest Route Between Two Points in a Network," The Computer Journal, Vol. 9, pp. 275-280.

Nilsson, N. J., 1971. Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, N. Y.

Nilsson, N. J., 1980. Principles of Artificial Intelligence, Tioga, Palo Alto, CA.

O'hEigeartaigh, M., Lenstra, S. K., and Rinnoy Kan, A. H. G. (eds.), 1985. Combinatorial Optimization: Annotated Bibliographies, Wiley, N. Y.

Ortega, J. M., and Rheinboldt, W. C., 1970. Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, N. Y.

Osman, I. H., and Laporte, G., 1996. "Metaheuristics: A Bibliography," Annals of Operations Research, Vol. 63, pp. 513-628.

Padberg, M. W., and Grötschel, M., 1985. "Polyhedral Computations," in The Traveling Salesman Problem, Lawler, E., Lenstra, J. K., Rinnoy Kan, A. H. G., and Shmoys, D. B. (eds.), Wiley, N. Y., pp. 307-360.

Pallottino, S., 1984. "Shortest Path Methods: Complexity, Interrelations and New Propositions," Networks, Vol. 14, pp. 257-267.

Pallottino, S., and Scutellà, M. G., 1991. "Strongly Polynomial Algorithms for Shortest Paths," Ricerca Operativa, Vol. 60, pp. 33-53.

Pallottino, S., and Scutellà, M. G., 1997a. "Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects," Proc. of the International Colloquium on Equilibrium in Transportation Models, Montreal, Canada.

Pallottino, S., and Scutellà, M. G., 1997b. "Dual Algorithms for the Shortest Path Tree Problem," Networks, Vol. 29, pp. 125-133.

Pang, J.-S., 1984. "Solution of the General Multicommodity Spatial Equilibrium Problem by Variational and Complementarity Methods," J. of Regional Science, Vol. 24, pp. 403-414.

Pang, J.-S., and Yu, C.-S., 1984. "Linearized Simplicial Decomposition Methods for Computing Traffic Equilibria on Networks," Networks, Vol. 14, pp. 427-438.

Papadimitriou, C. H., and Steiglitz, K., 1982. Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, N. J.

Pape, U., 1974. "Implementation and Efficiency of Moore - Algorithms for the Shortest Path Problem," Math. Programming, Vol. 7, pp. 212-222.

Pardalos, P. M., and Rosen, J. B., 1987. Constrained Global Optimization: Algorithms and Applications, Springer-Verlag, N. Y.

Patricksson, M., 1991. "Algorithms for Urban Traffic Network Equilibria," Linköping Studies in Science and Technology, Department of Mathematics, Thesis No. 263, Linköping University, Linköping, Sweden.

Pattipati, K. R., and Alexandridis, M. G., 1990. "Application of Heuristic Search and Information Theory to Sequential Fault Diagnosis," IEEE Trans. on Systems, Man, and Cybernetics, Vol. 20, pp. 872-887.

Pattipati, K. R., Deb, S., Bar-Shalom, Y., and Washburn, R. B., 1992. "A

New Relaxation Algorithm and Passive Sensor Data Association," IEEE Trans. Automatic Control, Vol. 37, pp. 198-213.

Pearl, J., 1984. Heuristics, Addison-Wesley, Reading, MA.

Peters, J., 1990. "The Network Simplex Method on a Multiprocessor," Networks, Vol. 20, pp. 845-859.

Phillips, C., and Zenios, S. A., 1989. "Experiences with Large Scale Network Optimization on the Connection Machine," in The Impact of Recent Computing Advances on Operations Research, Vol. 9, Elsevier, Amsterdam, The Netherlands, pp. 169-180.

Pinar, M. C., and Zenios, S. A., 1992. "Parallel Decomposition of Multicommodity Network Flows Using a Linear-Quadratic Penalty Algorithm," ORSA J. on Computing, Vol. 4, pp. 235-249.

Pinar, M. C., and Zenios, S. A., 1993. "Solving Nonlinear Programs with Embedded Network Structures," in Network Optimization Problems, Du, D.-Z., and Pardalos, P. M. (eds.), World Scientific Publ., Singapore, pp. 177-202.

Pinar, M. C., and Zenios, S. A., 1994. "On Smoothing Exact Penalty Functions for Convex Constrained Optimization," SIAM J. on Optimization, Vol. 4, pp. 486-511.

Pinedo, M., 1995. Scheduling: Theory, Algorithms, and Systems, Prentice-Hall, Englewood Cliffs, N. J.

Poljak, B. T., 1987. Introduction to Optimization, Optimization Software Inc., N. Y.

Polymenakos, L. C., 1995. "ϵ-Relaxation and Auction Algorithms for the Convex Cost Network Flow Problem," Ph.D. Thesis, Electrical Engineering and Computer Science Dept, M.I.T., Cambridge, MA.

Polymenakos, L. C., and Bertsekas, D. P., 1994. "Parallel Shortest Path Auction Algorithms," Parallel Computing, Vol. 20, pp. 1221-1247.

Polymenakos, L. C., Bertsekas, D. P., and Tsitsiklis, J. N., 1998. "Efficient Algorithms for Continuous-Space Shortest Path Problems," IEEE Trans. on Automatic Control, Vol. AC-43, pp. 278-283.

Poore, A. B., 1994. "Multidimensional Assignment Formulation of Data Association Problems Arising from Multitarget Tracking and Multisensor Data Fusion," Computational Optimization and Applications, Vol. 3, pp. 27-57.

Poore, A. B., and Robertson, A. J. A., 1997. New Lagrangian Relaxation Based Algorithm for a Class of Multidimensional Assignment Problems," Computational Optimization and Applications, Vol. 8, pp. 129-150.

Powell, W. B., Jaillet, P., and Odoni, A., 1995. "Stochastic and Dynamic

Networks and Routing," Handbooks in OR and MS, Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (eds.), Vol. 8, North-Holland, Amsterdam, pp. 141-295.

Powell, W. B., Berkkam, E., and Lustig, I. J., 1993. "On Algorithms for Nonlinear Dynamic Networks," in Network Optimization Problems, Du, D.-Z., and Pardalos, P. M. (eds.), World Scientific Publ., Singapore, pp. 177-202.

Pulleyblank, W., 1983. "Polyhedral Combinatorics," in Mathematical Programming: The State of the Art - Bonn 1982, by Bachem, A., Grötschel, M., and Korte, B., (eds.), Springer, Berlin, pp. 312-345.

Pulleyblank, W., Cook, W., Cunningham, W., and Schrijver, A., 1993. An Introduction to Combinatorial Optimization, Wiley, N. Y.

Resende, M. G. C., and Veiga, G., 1993. "An Implementation of the Dual Affine Scaling Algorithm for Minimum-Cost Flow on Bipartite Uncapacitated Networks," SIAM J. on Optimization, Vol. 3, pp. 516-537.

Resende, M. G. C., and Pardalos, P. M., 1996. "Interior Point Algorithms for Network Flow Problems," Advances in Linear and Integer Programming, Oxford Lecture Ser. Math. Appl., Vol. 4, Oxford Univ. Press, New York, pp. 145-185.

Rockafellar, R. T., 1967. "Convex Programming and Systems of Elementary Monotonic Relations," J. of Math. Analysis and Applications, Vol. 19, pp. 543-564.

Rockafellar, R. T., 1969. "The Elementary Vectors of a Subspace of R^N," in Combinatorial Mathematics and its Applications, by Bose, R. C., and Dowling, T. A. (eds.), University of North Carolina Press, pp. 104-127.

Rockafellar, R. T., 1970. Convex Analysis, Princeton Univ. Press, Princeton, N. J.

Rockafellar, R. T., 1976. "Monotone Operators and the Proximal Point Algorithm," SIAM J. on Control and Optimization, Vol. 14, pp. 877-898.

Rockafellar, R. T., 1981. "Monotropic Programming: Descent Algorithms and Duality," in Nonlinear Programming 4, by Mangasarian, O. L., Meyer, R. R., and Robinson, S. M. (eds.), Academic Press, N. Y., pp. 327-366.

Rockafellar, R. T., 1984. Network Flows and Monotropic Programming, Wiley, N. Y.

Rudin, W., 1976. Real Analysis, McGraw Hill, N. Y.

Sahni, S., and Gonzalez, T., 1976. "P-Complete Approximation Problems," J. ACM, Vol. 23, pp. 555-565.

Schwartz, B. L., 1994. "A Computational Analysis of the Auction Algorithm," Eur. J. of Operations Research, Vol. 74, pp. 161-169.

Sheffi, Y., 1985. Urban Transportation Networks. Equilibrium Analysis with Mathematical Programming Methods, Prentice-Hall, Englewood Cliffs, N. J.

Shier, D. R., 1979. "On Algorithms for Finding the K Shortest Paths in a Network," Networks, Vol. 9, pp. 195-214.

Shier, D. R., and Witzgall, C., 1981. "Properties of Labeling Methods for Determining Shortest Path Trees," J. Res. Natl. Bureau of Standards, Vol. 86, pp. 317-330.

Shiloach, Y., and Vishkin, U., 1982. "An $O(n^2 \log n)$ Parallel Max-Flow Algorithm," J. Algorithms, Vol. 3, pp. 128-146.

Schrijver, A., 1986. Theory of Linear and Integer Programming, Wiley, N. Y.

Shapiro, J. E., 1979. Mathematical Programming Structures and Algorithms, Wiley, N. Y.

Shor, N. Z., 1985. Minimization Methods for Nondifferentiable Functions, Springer-Verlag, Berlin.

Srinivasan, V., and Thompson, G. L., 1973. "Benefit-Cost Analysis of Coding Techniques for Primal Transportation Algorithm," J. ACM, Vol. 20, pp. 194-213.

Strang, G., 1976. Linear Algebra and Its Applications, Academic Press, N. Y.

Suchet, C., 1949. Electrical Engineering, Vol. 68, pp. 843-844.

Tabourier, Y., 1973. "All Shortest Distances in a Graph: An Improvement to Dantzig's Inductive Algorithm," Disc. Math., Vol. 4, pp. 83-87.

Tardos, E., 1985. "A Strongly Polynomial Minimum Cost Circulation Algorithm," Combinatorica, Vol. 5, pp. 247-255.

Teboulle, M., 1992. "Entropic Proximal Mappings with Applications to Nonlinear Programming," Math. of Operations Research, Vol. 17, pp. 1-21.

Toint, P. L., and Tuyttens, D., 1990. "On Large Scale Nonlinear Network Optimization," Math. Programming, Vol. 48, pp. 125-159.

Tseng, P., 1986. "Relaxation Methods for Monotropic Programming Problems," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA.

Tseng, P., 1991. "Relaxation Method for Large Scale Linear Programming Using Decomposition," Math. of Operations Research, Vol. 17, pp. 859-880.

Tseng, P., 1998. "An ϵ-Out-of-Kilter Method for Monotropic Programming," Department of Mathematics Report, Univ. of Washington, Seattle, Wash.

Tseng, P., and Bertsekas, D. P., 1987. "Relaxation Methods for Linear Programs," Math. of Operations Research, Vol. 12, pp. 569-596.

Tseng, P., and Bertsekas, D. P., 1990. "Relaxation Methods for Monotropic Programs," Math. Programming, Vol. 46, 1990, pp. 127-151.

Tseng, P., and Bertsekas, D. P., 1993. "On the Convergence of the Exponential Multiplier Method for Convex Programming," Math. Programming, Vol. 60, pp. 1-19.

Tseng, P., and Bertsekas, D. P., 1996. "An Epsilon-Relaxation Method for Separable Convex Cost Generalized Network Flow Problems," Lab. for Information and Decision Systems Report P-2374, M.I.T., Cambridge, MA.

Tseng, P., Bertsekas, D. P., and Tsitsiklis, J. N., 1990. "Partially Asynchronous Parallel Algorithms for Network Flow and Other Problems," SIAM J. on Control and Optimization, Vol. 28, pp. 678-710.

Tsitsiklis, J. N., 1989. "Markov Chains with Rare Transitions and Simulated Annealing," Math. of Operations Research, Vol. 14, pp. 70-90.

Tsitsiklis, J. N., 1992. "Special Cases of Traveling Salesman and Repairman Problems with Time Windows," Networks, Vol. 22, pp. 263-282.

Tsitsiklis, J. N., 1995. "Efficient Algorithms for Globally Optimal Trajectories," IEEE Trans. on Automatic Control, Vol. 40, pp. 1528-1538.

Tsitsiklis, J. N., and Bertsekas, D. P., 1986. "Distributed Asynchronous Optimal Routing in Data Networks," IEEE Trans. on Automatic Control, Vol. 31, pp. 325-331.

Ventura, J. A., and Hearn, D. W., 1993. "Restricted Simplicial Decomposition for Convex Constrained Problems," Math. Programming, Vol. 59, pp. 71-85.

Voß, S., 1992. "Steiner's Problem in Graphs: Heuristic Methods,", Discrete Applied Math., Vol. 40, pp. 45-72.

Von Randow, R., 1982. Integer Programming and Related Areas: A Classified Bibliography 1978-1981, Lecture Notes in Economics and Mathematical Systems, Vol. 197, Springer-Verlag, N. Y.

Von Randow, R., 1985. Integer Programming and Related Areas: A Classified Bibliography 1982-1984, Lecture Notes in Economics and Mathematical Systems, Vol. 243, Springer-Verlag, N. Y.

Warshall, S., 1962. "A Theorem on Boolean Matrices," J. ACM, Vol. 9, pp. 11-12.

Wein, J., and Zenios, S. A., 1991. "On the Massively Parallel Solution of the Assignment Problem," J. of Parallel and Distributed Computing, Vol.

13, pp. 228-236.

Whitting, P. D., and Hillier, J. A., 1960. “A Method for Finding the Shortest Route Through a Road Network,” Operations Research Quart., Vol. 11, pp. 37-40.

Winter, P., 1987. “Steiner Problem in Networks: A Survey,” Networks, Vol. 17, pp. 129-167.

Wright, S. J., 1997. Primal-Dual Interior Point Methods, SIAM, Phila., PA.

Ye, Y., 1992. “A Potential Reduction Algorithm Allowing Column Generation,” SIAM J. on Optimization, Vol. 2, pp. 7-20.

Ye, Y., 1997. Interior Point Algorithms: Theory and Analysis, Wiley, N. Y.

Zadeh, N., 1973a. “A Bad Network Problem for the Simplex Method and Other Minimum Cost Flow Algorithms,” Math. Programming, Vol. 5, pp. 255-266.

Zadeh, N., 1973b. “More Pathological Examples for Network Flow Problems,” Math. Programming, Vol. 5, pp. 217-224.

Zadeh, N., 1979. “Near Equivalence of Network Flow Algorithms,” Technical Report No. 26, Dept. of Operations Research, Stanford University, CA.

Zenios, S. A., and Mulvey, J. M., 1986. “Relaxation Techniques for Strictly Convex Network Problems,” Annals of Operations Research, Vol. 5, pp. 517-538.

Zoutendijk, G., 1976. Mathematical Programming Methods, North Holland, Amsterdam.