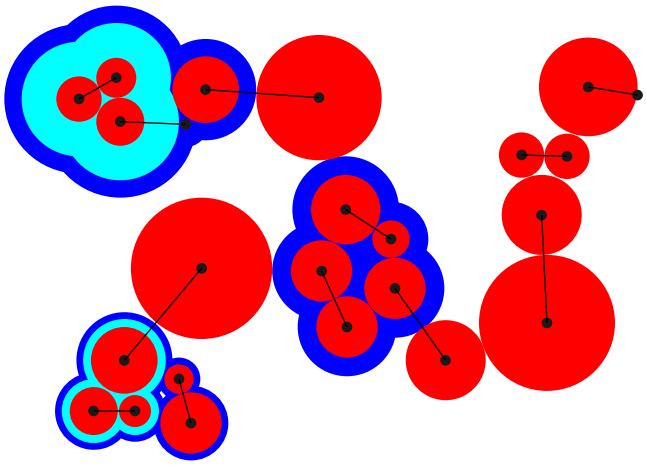
Combinatorial Optimization

(September 18, 1997)



William J. Cook William H. Cunningham
William R. Pulleyblank Alexander Schrijver

Contents

Preface

Combinatorial optimization is a lively field of applied mathematics, combining techniques from combinatorics, linear programming, and the theory of algorithms, to solve optimization problems over discrete structures. There are a number of classic texts in this field, but we felt that there is a place for a new treatment of the subject, covering some of the advances that have been made in the past decade. We set out to describe the material in an elementary text, suitable for a one semester course. The urge to include advanced topics proved to be irresistible, however, and the manuscript, in time, grew beyond the bounds of what one could reasonably expect to cover in a single course. We hope that this is a plus for the book, allowing the instructor to pick and choose among the topics that are treated. In this way, the book may be suitable for both graduate and undergraduate courses, given in departments of mathematics, operations research, and computer science. An advanced theoretical course might spend a lecture or two on chapter 2 and sections 3.1 and 3.2, then concentrate on 3.3, 3.4, 4.1, most of chapters 5 and 6 and some of chapters 8 and 9. An introductory course might cover chapter 2, sections 3.1 to 3.3, section 4.1 and one of 4.2 or 4.3, and sections 5.1 through 5.3. A course oriented more towards integer linear programming and polyhedral methods could be based mainly on chapters 6 and 7 and would include section 3.6.

The most challenging exercises have been marked in boldface. These should probably only be used in advanced courses.

The only real prerequisite for reading our text is a certain mathematical maturity. We do make frequent use of linear programming duality, so a reader unfamiliar with this subject matter should be prepared to study the linear programming appendix before proceeding with the main part of the text.

We benefitted greatly from thoughtful comments given by many of our colleagues who read early drafts of the book. In particular, we would like to thank Hernan Abeledo, Dave Applegate, Bob Bixby, Eddie Cheng, Joseph Cheriyan, Collette Coullard, Satoru Fujishige, Grigor Gasparian, Jim Geelen, Luis Goddyn, Michel Goemans, Mark Hartmann, Mike Jünger, Jon Lee, Tom McCormick, Kazuo Murota, Myriam Preissmann, Irwin Pressman, Maurice

Queyranne, André Rohe, András Sebő, Éva Tardos, and Don Wagner. Work on this book was carried out at Bellcore, the University of Bonn, Carleton University, CWI Amsterdam, IBM Watson Research, Rice University, and the University of Waterloo.

CHAPTER 1

Problems and Algorithms

1.1 TWO PROBLEMS

The Traveling Salesman Problem

An oil company has a field consisting of 47 drilling platforms off the coast of Nigeria. Each platform has a set of controls that makes it possible to regulate the amount of crude oil flowing from the wells associated with the platform back to the onshore holding tanks. Periodically, it is necessary to visit certain of the platforms, in order to regulate the rates of flows. This traveling is done by means of a helicopter which leaves an onshore helicopter base, flies out to the required platforms, and then returns to the base.

Helicopters are expensive to operate! The oil company wants to have a method for routing these helicopters in such a way that the required platforms are visited, and the total flying time is minimized. If we make the assumption that the flying time is proportional to the distance traveled, then this problem is an example of the *Euclidean traveling salesman problem.* We are given a set V of points in the Euclidean plane. Each point has a pair of (x, y) coordinates, and the distance between points with coordinates (x_1, y_1) and (x_2, y_2) is just $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. We wish to find a simple circuit (or *tour*) passing through all the points in V, for which the length is minimized. We call such a tour *optimal.* In this case, V consists of the platforms to be visited, plus the onshore base.

Euclidean Traveling Salesman Problem

Input: A set V of points in the Euclidean plane.
Objective: Find a simple circuit passing through the points for which the sum of the lengths of the edges is minimized.

There are many methods that attempt to solve this problem. Most simple ones share the characteristic that they do not work very well, either from a point of view of solution quality or of running time. For example, suppose we wish simply to try all possible solutions, and then select the best. This will certainly find the shortest circuit. However, if $|V| = n$, then there are $(n-1)!/2$ different possible solutions (Exercise 1.1). Suppose we have at our disposal a computer capable of evaluating a single possibility in one nanosecond ($= 10^{-9}$ seconds). If we had only 23 platforms to visit, then it would take approximately 178 centuries to run through the possible tours!

Suppose, on the other hand, that we require a faster method, but which need not be guaranteed to produce the optimal solution. The "Nearest Neighbor Algorithm" proceeds as follows. Pick any starting point. Go to the nearest point not yet visited. Continue from there to the nearest unvisited point. Repeat this until all points have been visited, then return to the starting point. The result of applying this to a sample problem (from Gerd Reinelt's TSPLIB) is given in Figure 1.1. Notice that although each move is locally the

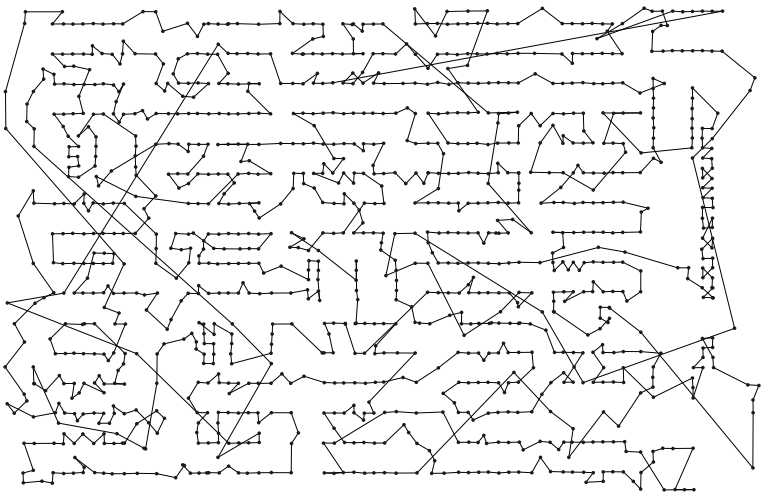


Figure 1.1. Nearest Neighbor solution

best possible, the overall result can be quite poor. First, it is easy to omit a point, which must be visited later at great expense. Second, at times you

may "paint yourself into a corner" where you are forced to make a long move to reach the nearest point where you can continue the tour.

The Matching Problem

A designer of logic circuits will use a plotter to draw a proposed circuit, so that it can be visually checked. The plotter operates by moving a pen back and forth and, at the same time, rolling a sheet of paper forwards and backwards beneath the pen. Each color of line is drawn independently, with a pen change before each new color. The problem is to minimize the time required to draw the figure. This time consists of two parts: "pen-down" time, when actual drawing is taking place, and "pen-up" time, when the pen is not contacting the paper, but is simply moving from the end of one line to be drawn to the start of another. Surprisingly, often more than half of the time is spent on pen-up movement. We have very little control over the pen-down time, but we can reduce the pen-up time considerably.

For example, suppose we wish to draw the circuit illustrated in Figure 1.2. Note first that the figure we are drawing is connected. This simplifies things,

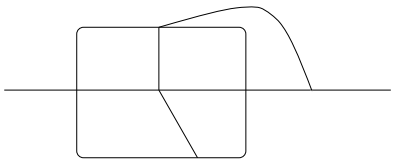


Figure 1.2. Circuit diagram

for reasons we discuss later. Can you convince yourself that some amount of pen-up motion is necessary? We define a *node* of the figure to be a point where two or more lines meet or cross, or where one line ends. In other words, it is a point of the figure from which a positive number of lines, other than two, emanates. We call a node *odd* if there is an odd number of lines coming out, and *even* otherwise. See Figure 1.3.

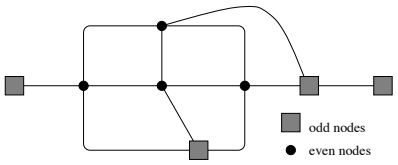


Figure 1.3. Odd and even nodes

One of the oldest theorems of graph theory implies that there will always be an even number of odd nodes. Another old theorem, due to Euler, states

that the figure can be traced, returning to the starting point, with no pen-up motion if and only if it is connected and there are no odd nodes.

We minimize pen-up motion by finding a set of new lines that we can add to the figure turning every odd node into an even node, and such that the total traversal time of the new lines is as small as possible.

Let $t(p, q)$ be the time required to draw a line from point p to q (pen-up or pen-down). If we make the assumption that $t(p, q)$ is proportional to the Euclidean distance between p and q, then t satisfies the *triangle inequality*: for any points p, q, r, we have $t(p, r) \leq t(p, q) + t(q, r)$. This is also satisfied, for example, when $t(p, q)$ is proportional to whichever direction of motion—horizontal or vertical—is greater.

Whenever t satisfies the triangle inequality, the optimal set of new lines will pair up the odd nodes. In the Euclidean case, the problem of finding these lines is an example of the *Euclidean matching problem*.

Euclidean Matching Problem

Input: A set V of points in the Euclidean plane.
Objective: Find a set of lines, such that each point is an end of exactly one line, and such that the sum of the lengths of the lines is minimized.

If the original figure is not connected, then we may add the extra lines and obtain a figure having no odd nodes, but which itself is not connected. In this case, some amount of extra pen-up motion is necessary. Moreover, the problem of minimizing this motion includes the Euclidean traveling salesman problem as a special case. For suppose we have an instance of the Euclidean traveling salesman problem which we wish to solve. We draw a figure consisting of one tiny circle in the location of each point. If we take one of these circles to be the pen's home position, then the problem of minimizing pen-up time is just the traveling salesman problem, assuming pen travel time is proportional to Euclidean distance.

Some Similarities and Differences

The Euclidean traveling salesman problem and the Euclidean matching problem are two prominent models in combinatorial optimization. The two problems have several similarities. First, each involves selecting sets of lines connecting points in the plane. Second, in both cases, the number of feasible solutions is far too large to consider them all in a reasonable amount of time. Third, most simple heuristics for the problems do not perform very well.

There is a major difference between the problems lurking under the surface, however. On the one hand, there exists an efficient algorithm, due to Edmonds, that will find an optimal solution for any instance of the Euclidean

matching problem. On the other hand, not only is no such algorithm known for the Euclidean traveling salesman problem, but most researchers believe that there simply does not exist such an algorithm!

The reason for this pessimistic view of the Euclidean traveling salesman problem lies in the theory of computational complexity, which we discuss in Chapter 9. Informally, the argument is that if there would exist an efficient algorithm for the Euclidean traveling salesman problem, then there would also exist such an efficient algorithm for *every* problem for which we could check the feasibility of at least one optimal solution efficiently. This last condition is not very strong, and just about every combinatorial optimization problem satisfies it.

Throughout the book, we will be walking the line between problems that are known to have efficient algorithms, and problems that are known to be just as difficult as the Euclidean traveling salesman problem. Most of our effort will be spent on describing models that lie on the "good side" of the dividing line, including an in-depth treatment of matching problems in Chapters 5 and 6. Besides being very important on their own, these "good" models form the building blocks for attacks on problems that lie on the "bad side." We will illustrate this with a discussion of the traveling salesman problem in Chapter 7, after we have assembled a toolkit of optimization techniques.

1.2 MEASURING RUNNING TIMES

Although the word "efficient," which we used above, is intuitive and would suffice for some purposes, it is important to have a means of quantifying this notion. We follow established conventions by estimating the efficiency of an algorithm by giving upper bounds on the number of steps it requires to solve a problem of a given size. Before we make this precise, a few words of warning are in order. Bounding the number of steps only provides an estimate of an algorithm's efficiency and it should not be taken as a hard and fast rule that having a better bound means better performance in practice. The reason for this is the bounds are taken over all possible instances of a given problem, whereas in practice you only want your algorithm to solve the instances you have in hand as quickly as possible. (It may not really concern you that some pathological examples could cause your algorithm to run and run and run.) A well-known example of this phenomenon is the simplex method for linear programming: it performs remarkably well on wide classes of problems, yet there are no good bounds on its behavior in general. It is fair to say, however, that this idea of the complexity of an algorithm does often point out the advantages of one method over another. Moreover, the widespread use of this notion has led to the discovery of many algorithms that turned out to be not only superior in a theoretical sense, but also much faster in practice. With this in mind, let us define more precisely what we mean by "giving upper bounds on the number of steps."

The concept of an algorithm can be expressed in terms of a Turing Machine or some other formal model of computation (see Chapter 9), but for now the intuitive notion of an algorithm as a list of instructions to solve a problem is sufficient. What we are concerned with is: How long does an algorithm take to solve a given problem? Rapid changes in computer architecture make it nearly pointless to measure all running times in terms of a particular machine. For this reason we measure running times on an abstract computer model where we count the number of "elementary" operations in the execution of the algorithm. Roughly speaking, an elementary operation is one for which the amount of work is bounded by a constant, that is, it is not dependent on the size of the problem instance. However, for the arithmetic operations of addition, multiplication, division, and comparison, we sometimes make an exception to this rule and count such operations as having unit cost, that is, the length of the numbers involved does not affect the cost of the operation. This is often appropriate, since the numbers occurring in many algorithms do not tend to grow as the algorithm proceeds. A second, more precise, model of computation counts the number of "bit operations": the numbers are represented in binary notation and the arithmetic operation is carried out bit by bit. This is more appropriate when the length of the numbers involved significantly affects the complexity of a problem (for example, testing whether a number is prime).

A combinatorial optimization problem usually consists of a discrete structure, such as a network or a family of sets, together with a set of numbers (which may represent costs or capacities, for example). We measure the size of such a problem by the length of an encoding of the structure (say in binary notation) plus the size of the set of numbers. (Either each number is counted as a single unit [when we are counting arithmetic operations] or we count the number of digits it takes to write each number in binary notation [when we are counting bit operations].) This measure, of course, depends on the particular encoding chosen, but if one is consistent with the types of encodings used, a robust measure can be obtained. Furthermore, in most cases the various choices of an encoding will differ in size by only a constant factor. So given an instance of a problem, we measure its size by an integer n, representing the number of bits in the encoding plus the size of the set of numbers. We can therefore make statements like, "the number of steps is bounded by $5n^2 + 3n$."

When analyzing an algorithm, we are mainly interested in its performance on instances of large size. This is due to the obvious reason that just about any method would solve a problem of small size. A superior algorithm will really start to shine when the problem sizes are such that a lesser method would not be able to handle the instances in any reasonable amount of time. Therefore, if an algorithm has a running-time bound of $5n^2 + 3n$, we would often ignore the $3n$ term, since it is negligible for large values of n. Furthermore, although a bound of $5n^2$ is clearly better than a bound of $17n^2$, it probably would not make the difference between being able to solve an instance of a problem and

not being able to solve it. So we normally concentrate on the magnitude of the bound, describing $5n^2 + 3n$ as "order n^2." There is a formal notation for this: If $f(n)$ and $g(n)$ are positive real-valued functions on the set of nonnegative integers, we say $f(n)$ is $O(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all large enough values of n. (The notation $O(g(n))$ is read "big oh of $g(n)$".) Thus $5n^2 + 3n$ is $O(n^2)$ and $35 \cdot 2^n + n^3$ is $O(2^n)$.

As an example of these ideas, consider once again the Nearest Neighbor Algorithm for the traveling salesman problem. We described this method as a fast (but sometimes poor) alternative to enumerating all $(n-1)!/2$ possible tours. We can quantify this easily with the big oh notation.

Let's first consider the arithmetic operation model. An instance of the Euclidean traveling salesman problem can be specified by giving the (x, y) coordinates of the n points to be visited. So the size of an instance is simply $2n$.

An easy (albeit somewhat inefficient) way to implement the Nearest Neighbor Algorithm is to set up an n-element array, where each object in the array has the three fields $\boxed{x \mid y \mid \text{mark}}$. We initialize the array by placing the (x, y) coordinates of each point v_i in the i^{th} object and setting all the mark fields to 0. A general pass of the algorithm takes a point v_j (say $j = 1$ on the first pass), scans through all n objects, computing the distance from v_i to v_j for all points v_i having mark equal to 0, while keeping track of the point v_{i^*} that has the least such distance. We then output v_{i^*} as the next point in the tour, set v_{i^*}'s mark field to 1 and continue the search from v_{i^*}. The algorithm terminates when we have visited all n points.

The initialization pass takes $3n$ elementary operations (excluding an approximately equal number of loop and control operations). A general pass takes n steps to check the mark field, plus at most $n-1$ distance calculations, each of which takes 3 additions, 2 multiplications, and 1 comparison (to keep track of the minimum distance). (Notice that we do not need to calculate a square root in the distance calculation, since we need only compare the values $(x_j - x_i)^2 + (y_j - y_i)^2$ to find the point v_{i^*} of minimum distance to v_j.) Since we execute the general step $n - 1$ times, we obtain an upper bound of $3n + (n-1)(n + 6(n-1))$ operations. That is, the Nearest Neighbor Algorithm takes $O(n^2)$ arithmetic operations.

To analyze the algorithm in the bit operation model, we need to measure the size of the input in a way that takes into acount the number of bits in the (x, y) coordinates. A standard estimate is $2nM$, where M is the maximum of $1 + \lceil \log(|x| + 1) \rceil$ and $1 + \lceil \log(|y| + 1) \rceil$ amongst the (x, y) coordinates. (We take logs with base 2. If t is a rational number, then $\lceil t \rceil$ is the smallest integer that is greater than or equal to t and $\lfloor t \rfloor$ is the greatest integer that is less than or equal to t.) The number of elementary operations in the algorithm only changes in the fact that we must now read M-bit-long numbers (so the initialization takes $2nM + n$ steps), and compute and compare the values

$(x_j - x_i)^2 + (y_j - y_i)^2$ bitwise (which takes $O(M^2)$ operations). So a quick estimate of the number of bit operations is $O(n^2M^2)$.

Our main goal will be to present algorithms that, like the Nearest Neighbor Algorithm, have running-time bounds of $O(n^k)$ for small values of k, whenever possible. Such "polynomial-time algorithms" have the nice property that their running times do not increase too rapidly as the problem sizes increase. (Compare n^3 and 2^n for $n = 100$.)

The above analysis shows that the Nearest Neighbor Algorithm is, in fact, polynomial-time in both the arithmetic model and the bit model. This will occur very often in the book. Typically, we will work with the arithmetic model, but a simple computation will show that the sizes of the numbers appearing in the algorithm do not grow too fast (that is, if t is the number of bits in the problem, then all of the numbers appearing will be $O(t^k)$ for some fixed k), and so a polynomial-time bound in the arithmetic model will directly imply a polynomial-time bound in the bit model. Indeed, throughout the text, whenever we say that an algorithm runs in "polynomial time," we implicitly mean that it runs in polynomial time in the bit model. It should be noted, however, that there are important problems (such as the linear programming problem) for which polynomial-time algorithms in the bit model are known, but no algorithm is known that is polynomial-time in the arithmetic model.

We will discuss the issue of computational complexity further in Chapter 9.

Exercises

1.1. Show that there are $(n-1)!/2$ distinct tours for a Euclidean traveling salesman problem on n points.

1.2. Suppose we have a computer capable of evaluating a feasible solution to a traveling salesman problem in one nanosecond ($= 10^{-9}$ seconds). How large a problem could we solve in 24 hours of computing time, if we tried all possible solutions? How would the size increase if we had a machine ten times faster? One hundred times faster?

CHAPTER 2

Optimal Trees and Paths

2.1 MINIMUM SPANNING TREES

A company has a number of offices and wants to design a communications network linking them. For certain pairs v, w of offices it is feasible to build a direct link joining v and w, and there is a known (positive) cost c_{vw} incurred if link vw is built. The company wants to construct enough direct links so that every pair of offices can communicate (perhaps indirectly). Subject to this condition, the company would like to minimize the total construction cost.

The above situation can be represented by a diagram (Figure 2.1) with a point for each office and a line segment joining v and w for each potential link. Notice that in this setting, unlike that of the Euclidean traveling salesman problem, we do not have the possibility of a direct connection between every pair of points. Moreover, the cost that we associate with the "feasible" pairs of points need not be just the distance between them. To describe such optimization problems more accurately we use the language of graph theory.

An (undirected) *graph* G consists of disjoint finite sets $V(G)$ of *nodes*, and $E(G)$ of *edges*, and a relation associating with each edge a pair of nodes, its *ends*. We say that an edge is *incident* to each of its ends, and that each end is *adjacent* to the other. We may write $G = (V, E)$ to mean that G has node-set V and edge-set E, although this does not define G. Two edges having the same ends are said to be *parallel*; an edge whose ends are the same is called a *loop*; graphs having neither loops nor parallel edges are called *simple*. We may write $e = vw$ to indicate that the ends of e are v and w. Strictly speaking, this should be done only if there are no parallel edges. In fact, in most applications, we can restrict attention to simple graphs. A *complete*

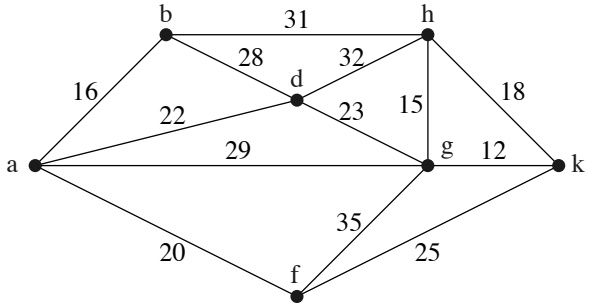


Figure 2.1. Network design problem

graph is a simple graph such that every pair of nodes is the set of ends of some edge.

A *subgraph* H of G is a graph such that $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and each $e \in E(H)$ has the same ends in H as in G. Although in general the sets of nodes and edges do not determine the graph, this is so when we know the graph is a subgraph of a given graph. So given a graph G and subsets P of edges and Q of nodes, we may refer unambiguously to "the subgraph (P,Q) of G." For $A \subseteq E$, $G\backslash A$ denotes the subgraph H obtained by *deleting* A, that is, $V(H) = V$ and $E(H) = E\backslash A$. Similarly, we can delete a subset B of V, if we also delete all edges incident with nodes in B. The resulting subgraph is denoted by $G\backslash B$ or by $G[V\backslash B]$; it may be referred to as the subgraph of G *induced* by $V\backslash B$. For $a \in V$ or E, we may abbreviate $G\backslash\{a\}$ to $G\backslash a$. A subgraph H of G is *spanning* if $V(H) = V(G)$.

Our standard name for a graph is G, and we often abbreviate $V(G)$ to V and $E(G)$ to E. We usually reserve the symbols n and m to denote $|V|$ and $|E|$, respectively. We extend this and other notation to subscripts and superscripts. For example, for graphs G' and G_1, we use n' to denote $|V(G')|$ and V_1 to denote $V(G_1)$.

A *path* P in a graph G is a sequence $v_0, e_1, v_1, \ldots, e_k, v_k$ where each v_i is a node, each e_i is an edge, and for $1 \leq i \leq k$, the ends of e_i are v_{i-1} and v_i. We say that P is *from* v_0 *to* v_k, or that it is a (v_0, v_k)*-path*. It is *closed* if $v_0 = v_k$; it is *edge-simple* if $e_1, \ldots, e_k$ are distinct; it is *simple* if $v_0, \ldots, v_k$ are distinct; it is a *circuit* if it is closed, $v_0, \ldots, v_{k-1}$ are distinct, and $k \geq 1$. We remark that if there is a path from u to v, then there is a simple one. The *length* of P is k, the number of edge-terms of P. The graph G is *connected* if every pair of nodes is joined by a path. A node v of a connected graph G is a *cut node* if $G \setminus v$ is not connected.

The requirement in the communications network design problem is that the subgraph consisting of all the centers and of the subset of links that we choose to build be connected. Suppose that each edge e of a graph G has a positive cost c_e, and the cost of a subgraph is the sum of the costs of its edges. Then the problem is:

Connector Problem

Given a connected graph G and a positive cost c_e for each $e \in E$, find a minimum-cost spanning connected subgraph of G.

Using the fact that the costs are positive, we can show that an optimal subgraph will be of a special type. First, we make the following observation.

Lemma 2.1 *An edge $e = uv$ of G is an edge of a circuit of G if and only if there is a path in $G \backslash e$ from u to v.* ∎

It follows that if we delete an edge of some circuit from a connected graph, the new graph is still connected. So an optimal solution to the connector problem will not have any circuits. A graph having no circuit is called a *forest*; a connected forest is called a *tree*. Hence we can solve the connector problem by solving the *minimum spanning tree* (MST) problem:

Minimum Spanning Tree Problem

Given a connected graph G and a real cost c_e for each $e \in E$, find a minimum cost spanning tree of G.

We remark that the connector problem and the MST problem are equivalent for positive edge costs. If we allow negative costs, this is no longer true. We shall solve the minimum spanning tree problem for arbitrary edge-costs. The possibility of negative costs in the connector problem is the subject of Exercise 2.7.

A second useful observation is the following.

Lemma 2.2 *A spanning connected subgraph of G is a spanning tree if and only if it has exactly $n - 1$ edges.* ∎

We leave its proof as Exercise 2.4.

Surprisingly simple algorithms will find a minimum spanning tree. We describe two such algorithms, both based on a "greedy" principle—that is, they make the cheapest choice at each step.

Kruskal's Algorithm for MST

Keep a spanning forest $H = (V, F)$ of G, with $F = \emptyset$ initially.
At each step add to F a least-cost edge $e \notin F$ such that H remains a forest.
Stop when H is a spanning tree.

If we apply Kruskal's Algorithm to the graph of Figure 2.1, edges are chosen in the order gk, gh, ab, af, ad, dg. This method was first described by Kruskal [1956]. The second algorithm is known as "Prim's Algorithm", and was described in Jarník [1930], Prim [1957], and Dijkstra [1959].

Prim's Algorithm for MST

Keep a tree $H = (V(H), T)$ with $V(H)$ initially $\{r\}$ for some $r \in V$, and T initially $\emptyset$.
At each step add to T a least-cost edge e not in T such that H remains a tree.
Stop when H is a spanning tree.

If Prim's Algorithm is applied to the graph of Figure 2.1 with $r = a$, edges are chosen in the order ab, af, ad, dg, gk, gh.

We show, first, that these algorithms do find a minimum spanning tree, and, second, that they have efficient implementations.

Validity of MST Algorithms

We begin with a fundamental characterization of connectivity of a graph. For a graph $G = (V,\ E)$ and $A \subseteq V$, we denote by $\delta(A)$ the set $\{e \in E$: e has an end in A and an end in $V \backslash A\}$ and by $\gamma(A)$ the set $\{e \in E$: both ends of e are in $A\}$. A set of the form $\delta(A)$ for some A is called a *cut* of G.

Theorem 2.3 *A graph $G = (V,\ E)$ is connected if and only if there is no set $A \subseteq V,\ \emptyset \neq A \neq V$, with $\delta(A) = \emptyset$.*

Proof: It is easy to see that, if $\delta(A) = \emptyset$ and $u \in A,\ v \notin A$, then there can be no path from u to v, and hence, if $\emptyset \neq A \neq V,\ G$ is not connected.

We must show that, if G is not connected, then there exists such a set A. Choose $u, v \in V$ such that there is no path from u to v. Define A to be $\{w \in V$: there exists a path from u to $w\}$. Then $u \in A$ and $v \notin A$, so $\emptyset \neq A \neq V$. We claim $\delta(A) = \emptyset$. For, if not, suppose that $p \in A,\ q \notin A$, and $e = pq \in E$. Then adding $e,\ q$ to any path from u to p gives a path from u to q, contradicting the fact that $q \notin A$. ∎

The following result allows us to show that both of the above minimum spanning tree algorithms (and, incidentally, a variety of others) work correctly. Let us call a subset A of edges of G *extendible* to an MST if A is contained in the edge-set of some MST of G.

Theorem 2.4 *Suppose that $B \subseteq E$, that B is extendible to an MST, and that e is a minimum-cost edge of some cut D satisfying $D \cap B = \emptyset$. Then $B \cup \{e\}$ is extendible to an MST.*

Before proving Theorem 2.4, we use it to prove that both algorithms are correct.

Theorem 2.5 *For any connected graph G with arbitrary edge costs c, Prim's Algorithm finds a minimum spanning tree.*

Proof: We begin by showing that at each step, $\delta(V(H))$ is the set of edges f such that adding f to T preserves the tree property of H. This follows from the fact that adding f creates a circuit if and only if both ends of f are in $V(H)$, by Lemma 2.1, and adding f makes H not connected if and only if neither end of f is in $V(H)$, by Theorem 2.3. Hence the algorithm chooses $e \in \delta(V(H))$ such that c_e is minimum. Now $\delta(V(H))$ cannot be empty until H is spanning, since G is connected. Therefore, the final H determined by the algorithm is a spanning tree of G. Moreover, since $\emptyset$ is extendible to an MST, at each step of the algorithm $B = T$, e, and $D = \delta(V(H))$ satisfy the hypotheses of Theorem 2.4. Therefore, the edge-set of the spanning tree H constructed by the algorithm is extendible to an MST, and hence H *is* an MST. ∎

For each node v of a graph G, let C_v be the set of nodes w such that there is a (v, w)-path in G. It is easy to see that $v \in C_w$ if and only if $w \in C_v$, so every node is in exactly one such set. The subgraphs of G of the form $G[C_v]$ are called the *components* of G. Obviously if G is connected, then it is its only component.

Theorem 2.6 *For any connected graph G with arbitrary edge costs c, Kruskal's Algorithm finds a minimum spanning tree.*

Proof: Let $S_1, \ldots, S_k$ be the node-sets of the components of $H = (V,\ F)$ at a given step of the algorithm. Then $f \in E$ can be added to F and preserve the forest property of H if and only if, by Lemma 2.1, the ends of f are in different S_i. In particular, any element of $\delta(S_i)$, for some i, has this property. It follows that the algorithm does construct a spanning tree, since if H is not connected and there is no such edge f, then $\delta(S_i) = \emptyset$ and $\emptyset \neq S_i \neq V$, which would imply that G is not connected. Moreover, if e is an edge chosen by the algorithm, B is the edge-set of the current spanning forest H when e is chosen, S_i is the node-set of a component of H containing an end of e, and $D = \delta(S_i)$, then $c_e = \min\{c_f : f \in D\}$. Hence, since $\emptyset$ is extendible to an MST, each $E(H)$ occurring in the algorithm is extendible to an MST by Theorem 2.4. It follows that the tree constructed by the algorithm is an MST. ∎

Finally, we need to provide a proof of Theorem 2.4. We use the following lemma, whose proof is left as an exercise.

Lemma 2.7 *Let $H = (V, T)$ be a spanning tree of G, let $e = vw$ be an edge of G but not H, and let f be an edge of a path in T from v to w. Then the subgraph $H' = (V, (T \cup \{e\})\backslash\{f\})$ is a spanning tree of G.* ∎

Proof of Theorem 2.4: Let $H = (V, T)$ be an MST such that $B \subseteq T$. If $e \in T$, then we are done, so suppose not. Let P be a path in H from v to w, where $vw = e$. Since there is no path in $G \backslash D$ from v to w, there is an edge f of P such that $f \in D$. Then $c_f \geq c_e$, and so by Lemma 2.7, $(V, (T \cup \{e\}) \backslash \{f\})$ is also an MST. Since $D \cap B = \emptyset$, it follows that $f \notin B$, so $B \cup \{e\}$ is extendible to an MST, as required. ∎

Efficiency of Minimum Spanning Tree Algorithms

Let us begin by describing a standard way to store a graph $G = (V,\ E)$ in a computer. We keep for each $v \in V$ a list L_v of the edges incident with v, and the other end of each edge. (Often the latter is enough to specify the edge.) If there is a cost associated with each edge, this is also stored with the edge. Notice that this means that each edge and cost is stored twice, in two different lists. In all complexity estimations we assume that $n = \mathrm{O}(m)$ and that $m = \mathrm{O}(n^2)$. Situations in which these assumptions do not hold are usually trivial, from the point of view of the problems we consider. Prim's Algorithm can be restated, using an observation from the proof of Theorem 2.5, as follows.

Prim's Algorithm

Initialize $H = (V(H), T)$ as $(\{r\},\ \emptyset)$;
While H is not a spanning tree
    Add to T a minimum-cost edge from $\delta(V(H))$.

Here is a straightforward implementation of this algorithm. We keep $V(H)$ as a *characteristic vector* x. (That is, $x_u = 1$ if $u \in V(H)$, and $x_u = 0$ if $u \notin V(H)$.) At each step we run through E, checking for each $f = uv$ whether $f \in \delta(V(H))$ by checking whether $x_u \neq x_v$, and if so comparing c_f to the current minimum encountered. So e can be chosen in $\mathrm{O}(m)$ time. Then x is updated by putting $x_v = 1$ where v is the end of e for which x_v was 0. This will be done $n - 1$ times, so we have a running time of $\mathrm{O}(nm)$.

Now we describe the improvement to this running time found by Prim and Dijkstra. We keep, for each $v \notin V(H)$, an edge $h(v)$ joining v to a node of H such that $c_{h(v)}$ is minimum. Then e can be chosen as the $h(v)$ that has smallest cost. The advantage of this is that only $\mathrm{O}(n)$ elementary steps are required to choose e. The disadvantage is that the values $h(v)$ need to be changed after each step. Say that w was added to $V(H)$ and v remains in $V \backslash V(H)$. Then $h(v)$ may have to be changed, but only if there is an edge $f = wv$ with $c_f < c_{h(v)}$. We can do all of these changes by going through L_w once, which is again $\mathrm{O}(n)$ work. So we do $\mathrm{O}(n)$ elementary steps per step of the algorithm and get a running time of $\mathrm{O}(n^2)$, an improvement on

$O(nm)$. Further improvements are presented, for example, in the monograph of Tarjan [1983].

Now we turn to the implementation of Kruskal's Algorithm. Notice that, once an edge $e = vw$ becomes unavailable to add to F, that is, H contains a path from v to w, it remains so. This means that finding the next edge to be added can be done simply by considering the edges in order of cost. That is, Kruskal's Algorithm can be restated, as follows.

Kruskal's Algorithm for MST

Order E as $\{e_1, \ldots, e_m\}$, where $c_{e_1} \leq c_{e_2} \leq \ldots \leq c_{e_m}$;
Initialize $H = (V, F)$ as $(V, \emptyset)$;
For $i = 1$ to m
    If the ends of e_i are in different components of H
        Add e_i to F.

Therefore, implementation of Kruskal's Algorithm requires first sorting m numbers. This can be accomplished in $O(m \log m)$ time by any one of a number of sorting algorithms.

To do the other step, we keep the partition of V into "blocks": the node-sets of components of H. The operations to be performed are $2m$ "finds": steps in which we find the block P containing a given v, and $n-1$ "merges": steps in which two blocks P, Q are replaced by $P \cup Q$, because an edge uv with $u \in P$, $v \in Q$ has been added to F. We keep for each v the name *block*(v) of the block containing v, so each merge can be done by changing *block*(v) to *block*(u) for every $v \in P$ and some $u \in Q$. It is important always to do this for the smaller of the two blocks being merged, that is, we take $|P| \leq |Q|$ (and so we need to keep the cardinalities of the blocks). To find the elements of blocks quickly, we keep each block also as a linked list. After a merge, the lists can also be updated in constant time. It is easy to see that the main work in this phase of the algorithm is the updating of *block*(v), which could require as much as $n/2$ elementary steps for a single merge. However, it can be proved that for each v, *block*(v) changes at most $\log n$ times. See Exercise 2.14. Therefore, the total work in the second phase of the algorithm is $O(m \log n) = O(m \log m)$, and we get a running time for Kruskal's Algorithm of $O(m \log m)$. Again, a discussion of further improvements can be found in Tarjan [1983].

Minimum Spanning Trees and Linear Programming

There is an interesting connection between minimum spanning trees and linear programming. Namely, there is a linear-programming problem for which every minimum spanning tree provides an optimal solution. This fact will be useful in Chapter 7, in connection with the traveling salesman problem.

Consider the following linear-programming problem. (For any set A and vector $p \in \mathbf{R}^A$ and any $B \subseteq A$, we use the abbreviation $p(B)$ to mean $\sum(p_j : j \in B)$. We denote the set of real numbers by $\mathbf{R}$, the set of integers by $\mathbf{Z}$, the set of nonnegative integers by $\mathbf{Z}_+$.

$$\text{Minimize } c^T x \tag{2.1}$$

$$\text{subject to}$$

$$x(\gamma(S)) \leq |S| - 1, \text{ for all } S, \ \emptyset \neq S \subset V \tag{2.2}$$

$$x(E) = |V| - 1 \tag{2.3}$$

$$x_e \geq 0, \text{ for all } e \in E. \tag{2.4}$$

(Do not be alarmed by the number of constraints.) Let S be a nonempty subset of nodes, let T be the edge-set of a spanning tree, and let x^0 be the characteristic vector of T. Notice that $x^0(\gamma(S))$ is just $|T \cap \gamma(S)|$, and since T contains no circuit, this will be at most $|S| - 1$. Also $x^0 \geq 0$ and $x^0(E) = |V| - 1$, so x^0 is a feasible solution of (2.1). Moreover, $c^T x^0 = c(T)$, that is, this feasible solution has objective value equal to the cost of the corresponding spanning tree. So, in particular, the optimal objective value of (2.1) is a lower bound on the cost of an MST. But in fact these two values are equal, a theorem of Edmonds [1971].

Theorem 2.8 *Let x^0 be the characteristic vector of an MST with respect to costs c_e. Then x^0 is an optimal solution of (2.1).*

Proof: We begin by writing an equivalent form of (2.1) that is easier to deal with. For a subset A of the edges, let $\kappa(A)$ denote the number of components of the subgraph (V, A) of G. Consider the problem

$$\text{Minimize } c^T x \tag{2.5}$$

$$\text{subject to}$$

$$x(A) \leq |V| - \kappa(A), \text{ for all } A \subset E \tag{2.6}$$

$$x(E) = |V| - 1 \tag{2.7}$$

$$x_e \geq 0, \text{ for all } e \in E. \tag{2.8}$$

We claim that the two problems have exactly the same feasible solutions, and thus the same optimal solutions. It is easy to see that every constraint of the form $x(\gamma(S)) \leq |S| - 1$ is implied by an inequality of the type (2.6); namely take $A = \gamma(S)$ and observe that $\kappa(\gamma(S)) \geq |V \setminus S| + 1$. On the other hand, we will show that every constraint of the form (2.6) is implied by a combination of constraints from (2.2) and (2.4). Let $A \subseteq E$, and let $S_1, \ldots, S_k$ be the node-sets of the components of the subgraph (V, A). Then $x(A) \leq \sum_{i=1}^k x(\gamma(S_i)) \leq \sum_{i=1}^k (|S_i| - 1) = |V| - k$.

Now it is enough to show that x^0 is optimal for problem (2.5), and further, it is enough to show that this is true where x^0 is the characteristic vector of

a spanning tree T generated by Kruskal's Algorithm. We will show in this case that x^0 is optimal by showing that Kruskal's Algorithm can be used to compute a feasible solution of the dual linear-programming problem to (2.5) that satisfies complementary slackness with x^0. It is easier to write the dual of (2.5) if we first replace minimize $c^T x$ by the equivalent maximize $-c^T x$. Now the dual problem is

$$\text{Minimize } \sum_{A \subseteq E} (|V| - \kappa(A)) y_A \tag{2.9}$$

$$\text{subject to}$$

$$\sum (y_A : \ e \in A) \geq -c_e, \text{ for all } e \in E \tag{2.10}$$

$$y_A \geq 0, \text{ for all } A \subset E. \tag{2.11}$$

Notice that y_E is not required to be nonnegative. Let $e_1, \ldots, e_m$ be the order in which Kruskal's Algorithm considers the edges. (Here we are following the second version of the statement of the algorithm.) Let R_i denote $\{e_1, \ldots, e_i\}$ for $1 \leq i \leq m$. Here is the definition of our dual solution y^0. We let $y^0_A = 0$ unless A is one of the R_i, we put $y^0_{R_i} = c_{e_{i+1}} - c_{e_i}$, for $1 \leq i \leq m-1$, and we put $y^0_{R_m} = -c_{e_m}$. It follows from the ordering of the edges that $y^0_A \geq 0$ for $A \neq E$. Now consider the constraints (2.10). Then, where $e = e_i$, we have

$$\sum (y^0_A : \ e \in A) = \sum_{j=i}^{m} y^0_{R_j} = \sum_{j=i}^{m-1} (c_{e_{i+1}} - c_{e_i}) - c_{e_m} = -c_{e_i} = -c_e.$$

In other words, all of these inequalities hold with equality. So we now know that y^0 is a feasible solution to (2.9), and also that the complementary slackness conditions of the form, $x^0_e > 0$ implies equality in the corresponding dual constraint, are satisfied. There is only one more condition to check, that $y^0_A > 0$ implies that x^0 satisfies (2.6) with equality. For this, we know that $A = R_i$ for some i. If (2.6) is not an equality for this R_i, then there is some edge of R_i whose addition to $T \cap R_i$ would decrease the number of components of $(V, T \cap R_i)$. But such an edge would have ends in two different components of $(V, R_i \cap T)$, and therefore would have been added to T by Kruskal's Algorithm. Therefore, x^0 and y^0 satisfy the complementary slackness conditions. It follows that x^0 is an optimal solution to (2.5), and hence to (2.1). ∎

Notice that, since any spanning tree that provides an optimal solution of the linear-programming problem must be an MST, and since the proof used only the fact that T was generated by Kruskal's Algorithm, we have actually given a second proof that Kruskal's Algorithm computes an MST.

Exercises

2.1. Find a minimum spanning tree in the graph of Figure 2.2 using: (a) Kruskal's Algorithm; (b) Prim's Algorithm with the indicated choice of r.

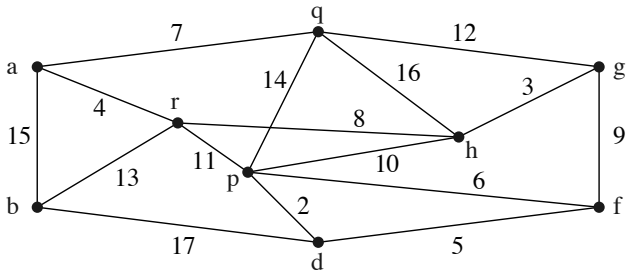


Figure 2.2. MST exercise

2.2. Find a dual solution to the linear-programming problem 2.1 for the graph of Figure 2.2.

2.3. Show that we may assume in the MST problem that the input graph is simple.

2.4. Prove Lemma 2.2.

2.5. Prove Lemma 2.7.

2.6. Give an algorithm to find a minimum-cost forest of a graph, where edge-costs are not assumed to be positive.

2.7. Give an algorithm to solve the connector problem where negative costs are allowed.

2.8. Show that any MST problem can be reduced to an MST problem with positive edge-costs.

2.9. Prove that if $H = (V, T)$ is an MST, and $e \in T$, then there is a cut D with $e \in D$ and $c_e = \min\{c_f : f \in D\}$.

2.10. Prove that a spanning tree $H = (V, T)$ of G is an MST if and only if for every $e = vw \in E \backslash T$ and every edge f of a (v, w) path in T, $c_e \geq c_f$.

2.11. Show that the following algorithm finds an MST of a connected graph G. Begin with $H = G$. At each step, find (if one exists) a maximum-cost edge e such that $H \backslash e$ is connected, and delete e from H. Try this algorithm on the example of Exercise 2.1.

2.12. Show that there is an $O(m)$ algorithm to find *some* spanning tree of a connected graph.

2.13. In the implementation of Prim's Algorithm, suppose we keep for each $v \in V(H)$ an edge $h(v)$ joining v to a node of $V \backslash V(H)$ whose cost is minimum. Does this idea lead to an $O(n^2)$ running time?

2.14. For the implementation of Kruskal's Algorithm described in the text, show that for each $v \in V$, $block(v)$ is changed at most $\log n$ times.

2.15. Here is another way to do finds and merges in Kruskal's Algorithm. Each block S has a distinguished node $name(S) \in S$. Each $v \in S$ different from $name(S)$ has a predecessor $p(v) \in S$ such that evaluating $p(v)$,

then $p(p(v)), \ldots$, we eventually get to $name(S)$. With each $name(S)$, we also keep $|S|$. Show how this idea can be used to implement Kruskal's Algorithm, so that the running time is $O(m \log m)$.

2.16. Suppose that, instead of the *sum* of the costs of edges of a spanning tree, we wish to minimize the *maximum* cost of an edge of a spanning tree. That is, we want the most expensive edge of the tree to be as cheap as possible. This is called the *minmax spanning tree problem*. Prove that every MST actually solves this problem. Is the converse true?

2.17. Here is a different and more general way to solve the minmax spanning tree problem of Exercise 2.16. Show that the optimal value of the objective is the smallest cost c_e such that $\{f : f \in E,\ c_f \leq c_e\}$ contains the edge-set of a spanning tree of G. Use this observation and the result of Exercise 2.12 to design an $O(m^2)$ algorithm. Can you improve it to $O(m \log m)$?

2.2 SHORTEST PATHS

Suppose that we wish to make a table of the minimum driving distances from the corner of Bay Street and Baxter Road to every other street corner in the city of Bridgetown, Barbados. By this we mean that the routes must follow city streets, obeying the directions on one-way streets. We can associate a graph with the "network" of the city streets, but the notion of direction imposed by the one-way streets leads to the idea of a directed graph.

A *directed graph* or *digraph* G consists of disjoint finite sets $V = V(G)$ of *nodes* and $E = E(G)$ of *arcs*, and functions associating to each $e \in E$ a *tail* $t(e) \in V$ and a *head* $h(e) \in V$. In other words, each arc has two *end* nodes, to which it is said to be *incident*, and a direction from one to the other. The street map of Bridgetown defines a digraph whose nodes are the street corners. There is an arc for each section of street joining (directly) two corners, and for each direction in which it is legal to drive along it.

The terminology and notation of digraph theory is similar to that of graph theory. In fact, to every digraph there corresponds a graph, obtained by letting the arcs be the edges and ignoring the arc directions. Whenever we use a digraph term or notation without definition, it means what it does for the associated undirected graph. Hence we get immediately notions like *loop* and *path* in a digraph. In addition, notions of *subdigraph* and *deletion* of arcs and nodes are defined exactly in analogy with corresponding terms for graphs. But some differences also appear. Two arcs of a digraph are *parallel* if they have the same head and the same tail, and a digraph is *simple* if it has no loops or parallel arcs. Hence a digraph may be simple as a digraph, but not as a graph. When we write $e = vw$ for an arc of a digraph G, we mean that $v = t(e),\ w = h(e)$. An arc e_i of a path $P :\ v_0, e_1, v_1, \ldots, e_k, v_k$ is *forward* if $t(e_i) = v_{i-1}$ and $h(e_i) = v_i$ and is *reverse* otherwise. A path in which every arc is forward is a *directed path* or *dipath*. A *dicircuit* is a dipath that is also

a circuit. If each $e \in E$ has a real cost c_e, the *cost* $c(P)$ (with respect to c) of the dipath P is defined to be $\sum_{i=1}^{k} c_{e_i}$. In Figure 2.3 a digraph with arc-costs is represented pictorially.

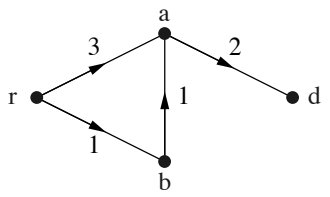


Figure 2.3. A Digraph with arc-costs

Shortest Path Problem

Input: A digraph G, a node $r \in V$, and a real cost vector $(c_e : e \in E)$.
Objective: To find, for each $v \in V$, a dipath from r to v of least cost (if one exists).

There are many direct applications of shortest path problems. We shall see that there are also many more difficult problems in combinatorial optimization for which solution algorithms use shortest path algorithms as subroutines.

One reason that a least-cost dipath to some $v \in V$ may not exist, is that G has no dipath at all from r to v. We could modify the algorithms we shall describe to detect this, but it is more convenient to be able to assume that it never happens. One way to do this is to check for this condition in advance by a graph-theoretic method. Instead, however, we can modify the given G so that there is an arc from r to v for every $v \in V$. Where this requires adding a new arc, we assign it a sufficiently large cost (how large?) so that a least-cost dipath will include it only if there was no dipath from r to v in the original digraph G. So we assume that dipaths exist from r to all the nodes. Notice that we may also assume that G is simple. (Why?)

Here is the basic idea behind all the methods for solving the shortest path problem. Suppose we know that there exists a dipath from r to v of cost y_v for each $v \in V$, and we find an arc $vw \in E$ satisfying $y_v + c_{vw} < y_w$. Since appending vw to the dipath to v gives a dipath to w, we know that there is a cheaper dipath to w, of cost $y_v + c_{vw}$. In particular, it follows that if $y_v,\ v \in V$, is the *least cost* of a dipath to v, then y satisfies

$$y_v + c_{vw} \geq y_w, \text{ for all } vw \in E. \tag{2.12}$$

We call $y = (y_v : v \in V)$ a *feasible potential* if it satisfies (2.12) and $y_r = 0$. Notice that (2.12) is the essential requirement, since subtracting y_r from each y_v preserves (2.12) and makes $y_r = 0$. Feasible potentials provide lower bounds for shortest path costs, as the following result shows.

Proposition 2.9 *Let y be a feasible potential and let P be a dipath from r to v. Then $c(P) \geq y_v$.*

Proof: Suppose that P is $v_0, e_1, v_1, \ldots, e_k, v_k$, where $v_0 = r$ and $v_k = v$. Then

$$c(P) = \sum_{i=1}^{k} c_{e_i} \geq \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} = y_v.$$

∎

Here is another simple but useful observation. Since we want dipaths from r to many other nodes, it may seem that the paths might use many arcs altogether. In fact, however, all the shortest paths can be assumed to use just one arc having head v for each node $v \neq r$. The reason is that *subpaths of shortest paths are shortest paths*, that is, if v is on the least-cost dipath P from r to w, then P splits into a dipath P_1 from r to v and a dipath P_2 from v to w. Obviously, if P_1 is not a least-cost dipath from r to v, then replacing it by a better one would also lead to a better dipath to w. Hence, the only arc having head v that we really need is the last arc of *one* least-cost dipath to v. Moreover, because there will be exactly $n - 1$ such arcs, and the corresponding subgraph contains a path from r to every other node, it is the arc-set of a spanning tree of G. So, just as in the connector problem of Section 2.1, the solution takes the form of a spanning tree of G. However, there are two crucial differences. First, not every spanning tree provides a feasible solution to the shortest path problem: We need a *directed spanning tree rooted at r*, meaning that it contains a dipath from r to v for every $v \in V$. Second, our objective here, in terms of the spanning tree, is not to minimize the sum of the costs of its arcs; see Exercise 2.18.

Ford's Algorithm

Proposition 2.9 provides a stopping condition for a shortest path algorithm. Namely, if we have a feasible potential y and, for each $v \in V$, a dipath from r to v of cost y_v, we know that each dipath is of least cost. Moreover, we have already described the essence of a "descent" algorithm — if y describes dipath costs and we find an arc vw violating (2.12), we replace y_w by $y_v + c_{vw}$. We can initialize such an algorithm with $y_r = 0$ and $y_v = \infty$ for $v \neq r$. Here $y_v = \infty$ simply means that we do not yet know a dipath to v, and ∞ satisfies $a + \infty = \infty$ and $a < \infty$ for all real numbers a. Since we wish, at termination of the algorithm with an optimal y, to obtain also the optimal dipaths, we add one more refinement to the algorithm. The arcs vw of a least-cost dipath

will satisfy $y_v + c_{vw} = y_w$, so the last arc of the optimal path to w will be the arc vw such that y_w was most recently lowered to $y_v + c_{vw}$. Moreover, the least-cost dipath to w must consist of a least-cost dipath to v with the arc vw appended, so knowing this "last-arc" information at each node allows us to trace (in reverse) the optimal dipath from r (because G is simple). For this reason, we keep a "predecessor" $p(w)$ for each $w \in V$ and set $p(w)$ to v whenever y_w is set to $y_v + c_{vw}$. Let us call an arc vw violating (2.12) *incorrect*. To *correct* vw means to set $y_w = y_v + c_{vw}$ and to set $p(w) = v$. To *initialize* y and p means to set $y_r = 0$, $p(r) = 0$, $y_v = \infty$ and $p(v) = -1$ for $v \in V \backslash \{r\}$. (We are using $p(v) = -1$ to mean that the predecessor of v is not (yet) defined, but we want to distinguish r from such nodes. We are assuming that $0, -1 \notin V$.) The resulting algorithm is due to Ford [1956].

Ford's Algorithm

Initialize y, p;
While y is not a feasible potential
    Find an incorrect arc vw and correct it.

On the digraph of Figure 2.3, Ford's Algorithm might execute as indicated in Table 2.1. At termination, we do have a feasible potential y and paths of

	Start		$vw = ra$		$vw = r$		$vw = ad$		$vw = ba$		$vw = ad$	
	y	p	y	p	y	p	y	p	y	p	y	p
r	0	0	0	0	0	0	0	0	0	0	0	0
a	∞	-1	3	r	3	r	3	rb	2	b	2	b
b	∞	-1	∞	-1	1	r	1	r	1	r	1	r
d	∞	-1	∞	-1	∞	-1	5	a	5	a	4	a

Table 2.1. Ford's Algorithm applied to the first example

cost y_v given (in reverse) by tracing the values of p back to r, and so we have solved this (trivial) example. Notice that we must have $y_v = y_{p(v)} + c_{p(v)v}$ at termination, but that this need not be true at all times — consider $v = d$ after the fourth iteration. In fact, $y_v \geq y_{p(v)} + c_{p(v)v}$ holds throughout. (Proof: It held with equality when y_v and $p(v)$ were assigned their current values and after that $y_{p(v)}$ can only decrease.)

Figure 2.4 shows a second example and Table 2.2 represents the first few iterations of Ford's Algorithm on that instance. It shows a situation in which the algorithm goes very badly wrong.

It is not hard to see that vw can be chosen as

$$ab,\ bd,\ da,\ ab,\ bd,\ da,\ ab, \ldots$$

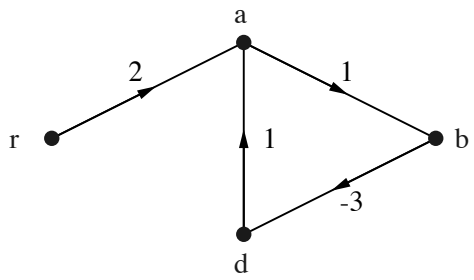


Figure 2.4. A second example

	Start		$vw = ra$		$vw = ab$		$vw = bd$		$vw = da$		$vw = ab$	
	y	p	y	p	y	p	y	p	y	p	y	p
r	0	0	0	0	0	0	0	0	0	0	0	0
a	∞	-1	2	r	2	r	2	r	1	d	1	d
b	∞	-1	∞	-1	3	a	3	a	3	a	2	a
d	∞	-1	∞	-1	∞	-1	0	b	0	b	0	b

Table 2.2. Ford's Algorithm applied to second example

indefinitely, that the algorithm will not terminate, and that certain y_v will become arbitrarily small (that is, will go to $-\infty$). This should not be such a surprise, since if we are asked to find a least-cost dipath from r to a we can repeat the sequence a, b, d as many times as we like before stopping at a. That is, there are arbitrarily cheap dipaths to a, so *there is no* least-cost one. It is apparent that if (G, c) has a negative-cost closed dipath (as in this example), then there exist nodes to which no least-cost dipath exists. In addition to wanting an algorithm that always terminates (quickly, we hope!), we want the algorithm to recognize when a negative-cost dicircuit exists. (Exercise: There is a negative-cost closed dipath if and only if there is a negative-cost dicircuit.)

In fact, not only are there many important applications in which negative costs really arise, but there are several in which a negative-cost dicircuit is actually the object of interest. As an example, consider the common situation of currency exchange rates where, for each pair v, w of currencies, we are quoted a rate r_{vw}, the number of units of currency w that can be purchased for one unit of currency v. Notice that if we convert a unit of currency 1 into currency 2, and then convert all of that into currency 3, we shall have $r_{12}r_{23}$ units of currency 3. This suggests looking for a sequence $v_0, v_1, v_2, \ldots, v_k$ of currencies with $v_0 = v_k$ such that $\prod_{i=1}^{k} r_{v_{i-1}v_i} > 1$, for on the associated sequence of exchanges we would make money. We form a digraph G whose nodes are the currencies, with an arc vw of cost $c_{vw} = -\log r_{vw}$ for each pair v, w. Then such a sequence is money-making if and only if the associated

closed dipath of G has cost $-\sum_{i=1}^{k} \log r_{v_{i-1}v_i} = -\log\left(\prod_{i=1}^{k} r_{v_{i-1}v_i}\right) < 0$. So we can check for the existence of a money-making sequence by checking for the existence of a negative-cost dicircuit in G.

Validity of Ford's Algorithm

We shall show that, provided no negative-cost dicircuit exists, Ford's Algorithm does terminate and that it terminates with least-cost dipaths. This algorithm is itself too crude to be directly useful. But all of the algorithms we treat later are refinements of it, so it is worthwhile (but a bit tedious) to establish its properties. The main step is the following.

Proposition 2.10 *If (G, c) has no negative-cost dicircuit, then at any stage of the execution of Ford's Algorithm we have:*

(i) If $y_v \neq \infty$, then it is the cost of a simple dipath from r to v;

(ii) If $p(v) \neq -1$, then p defines a simple dipath from r to v of cost at most y_v.

Proof: Let y_v^j be the value of y_v after j iterations. We know that y_v^j is the cost of a dipath (if $y_v^j \neq \infty$), so suppose that the dipath is not simple. Then there is a sequence $v_0, v_1, \ldots, v_k$ of nodes with $v_0 = v_k$ and iteration numbers $q_0 < q_1 < \ldots < q_k$ such that

$$y_{v_{i-1}}^{q_{i-1}} + c_{v_{i-1}v_i} = y_{v_i}^{q_i}, \quad 1 \leq i \leq k.$$

The cost of the resulting closed dipath is

$$\sum c_{v_{i-1}v_i} = \sum \left(y_{v_i}^{q_i} - y_{v_{i-1}}^{q_{i-1}}\right) = y_{v_k}^{q_k} - y_{v_0}^{q_0}.$$

But y_{v_k} was lowered at iteration q_k, so this dipath has negative cost, a contradiction, and (i) is proved. Notice that it follows from (i) that $y_r = 0$.

The proof of (ii) is similar. If p does not define a simple dipath from r to v, then there is a sequence $v_0, v_1, \ldots, v_k$ with $v_0 = v_k$ and $p(v_i) = v_{i-1}$ for $1 \leq i \leq k$. The cost of the resulting closed dipath is ≤ 0 since $c_{p(v)v} \leq y_v - y_{p(v)}$ always holds. But consider the most recent predecessor assignment on the dipath; say $y_{p(v)}$ was lowered. Then the above inequality is strict, so we have a negative-cost closed dipath, a contradiction.

Finally, we need to show that the simple dipath to v has cost at most y_v. Let the dipath be $v_0, e_1, v_1, \ldots, e_k, v_k$ where $v_0 = r$, $v_k = v$ and $p(v_i) = v_{i-1}$ for $1 \leq i \leq k$. Then its cost is $\leq \sum(y_{v_i} - y_{v_{i-1}}) = y_v - y_r = y_v$, as required. ∎

Theorem 2.11 *If (G, c) has no negative-cost dicircuit, then Ford's Algorithm terminates after a finite number of iterations. At termination, for each $v \in V$, p defines a least-cost dipath from r to v of cost y_v.*

Proof: There are finitely many simple dipaths in G. Therefore, by Proposition 2.10, there are a finite number of possible values for the y_v. Since at each step one of them decreases (and none increases), the algorithm terminates. At termination, for each $v \in V$, p defines a simple dipath from r to v of cost $\leq y_v$. But no dipath to v can have smaller cost than y_v by Proposition 2.9. ∎

A consequence of the correctness of Ford's Algorithm is the following fundamental fact. Notice that it applies even without any assumption about the existence of dipaths.

Theorem 2.12 *(G, c) has a feasible potential if and only if it has no negative-cost dicircuit.*

Proof: We have already observed that if G has a feasible potential, then it can have no negative-cost dicircuit. Now suppose that G has no negative-cost dicircuit. Add a new node r to G with arcs from r to v of cost zero for every $v \in V$. Where G' is the new digraph and c' is the new cost vector, (G', c') has no negative-cost dicircuit, because no dicircuit of G' goes through r. Now we can apply Ford's Algorithm to (G', c'), and since there is a dipath from r to all other nodes, it will terminate with a feasible potential, which clearly gives a feasible potential for (G, c). ∎

If there is no least-cost dipath to some node v, it is because there are arbitrarily cheap *nonsimple* dipaths to v. So it is natural to ask why we do not try to find a least-cost simple one. (One exists, because the number of simple dipaths is finite.) However, this problem is difficult (unless there is no negative-cost dicircuit) in the same sense that the traveling salesman problem is difficult. In fact, a solution to it could be used quite directly to solve the Euclidean traveling salesman problem (Exercise 2.26).

We shall see that Ford's Algorithm, although it can be modified to recognize the existence of negative-cost dicircuits and hence made finite in all cases (Exercise 2.20), does not have acceptable efficiency. (See Exercise 2.25.) We shall discuss a number of refinements that have better efficiency, although several of them work only in special cases. All of them specify more narrowly the order in which the arcs are considered in the basic step of the algorithm. However, there is one simple observation that can be made for the case in which the arc-costs are integers. Then each step of Ford's Algorithm decreases some y_v by at least 1, since all of these values are integer or ∞. We let C denote $2\max(|c_e| : e \in E) + 1$. Then we can prove the following.

Proposition 2.13 *If c is integer-valued, C is as defined above, and G has no negative-cost dicircuit, then Ford's Algorithm terminates after at most Cn^2 arc-correction steps.* ∎

The proof of Proposition 2.13 is left to Exercise 2.27. Several of the other exercises investigate better bounds that can be obtained via arguments that assume integral arc-costs and work with the size of the numbers. We shall see in the text that there are good bounds that do not depend on the size of the costs.

Feasible Potentials and Linear Programming

We have seen that feasible potentials provide lower bounds for dipath costs. But in fact at termination of Ford's Algorithm we have a feasible potential and dipaths for which equality holds. One possible statement of this fact is the following.

Theorem 2.14 *Let G be a digraph, $r, s \in V$ and $c \in \mathbf{R}^E$. If there exists a least-cost dipath from r to v for every $v \in V$, then*

$$\min\{c(P) : P \text{ a dipath from } r \text{ to } s\} = \max\{y_s : y \text{ a feasible potential}\}.$$

∎

We wish to point out the connection between this statement and linear-programming duality. The maximization in the theorem statement is obviously a linear-programming problem. It is convenient to drop the requirement that $y_r = 0$ and write that linear-programming problem as:

$$\begin{gathered} \text{Maximize } y_s - y_r \\ \text{subject to} \\ y_w - y_v \leq c_{vw}, \text{ for all } vw \in E. \end{gathered} \tag{2.13}$$

Where b_v is defined to be 1 if $v = s$, -1 if $v = r$ and 0 otherwise, the dual linear-programming problem of (2.13) is

$$\begin{gathered} \text{Minimize } \sum(c_e x_e : e \in E) \\ \text{subject to} \\ \sum(x_{wv} : w \in V,\ wv \in E) - \sum(x_{vw} : w \in V,\ vw \in E) = b_v, \text{ for all } v \in V \\ x_{vw} \geq 0, \text{ for all } vw \in E. \end{gathered} \tag{2.14}$$

The Duality Theorem says that if one of the optimal values in (2.13), (2.14) exists, then they both do, and they are equal. Notice that any dipath P from r to s provides a feasible solution to (2.14), as follows. Define $(x_e^P : e \in E)$ by: x_e^P is the number of times that arc e is used in P. (In particular, if P is simple, then x^P is $\{0,1\}$-valued, and is the characteristic vector of P.) Then the objective function of (2.14) for $x = x^P$ is just the cost of P. Therefore, Theorem 2.14 implies that, when shortest paths exist, (2.14) has an optimal

solution that is the characteristic vector of a simple dipath. As we shall see (Chapter 7), this result is equivalent to the statement that the vertices of the polyhedron of feasible solutions to (2.14) are characteristic vectors of simple dipaths.

Since we have solved the linear-programming problem (2.14) with Ford's Algorithm, one might wonder whether there is any connection between that algorithm and the simplex algorithm. The simplex algorithm keeps a set T of "basic" arcs (corresponding to the variables in (2.14) that are basic), a feasible solution x of (2.14), and a vector $y \in \mathbf{R}^V$ satisfying

$$x_e = 0, \text{ for all } e \notin T \tag{2.15}$$

$$y_v - y_w = c_{vw}, \text{ for all } vw \in T. \tag{2.16}$$

In each iteration it proceeds to a new such set by replacing one of the arcs in the set by one outside the set. The set T of basic arcs must correspond to a maximal linearly independent set of columns (that is, a column basis) $\{a_e : e \in T\}$ of the constraint matrix $A = \{a_e : e \in E\}$ of the equality constraints of (2.14). This matrix is called the *incidence matrix* of G. Its column bases can be characterized in a very nice way. We state the result here and leave the proof to Exercise 2.28.

Proposition 2.15 *Let G be a connected digraph and $A = \{a_e : e \in E\}$ be its incidence matrix. A set $\{a_e : e \in T\}$ is a column basis of A if and only if T is the arc-set of a spanning tree of G.* ∎

Ford's Algorithm, once it has found paths to all nodes, does have such a set T, namely, $\{p(v)v : v \in V \backslash \{r\}\}$. (It is possible to require that the simplex method for (2.14) keep such a directed spanning tree.) Moreover, the dipath from r to s determined by p uses only arcs from T, so its characteristic vector x^P satisfies (2.15). However, for this T, (2.16) becomes $y_{p(v)} + c_{p(v)v} = y_v$, a relation that is generally *not* enforced by Ford's Algorithm. Notice that enforcing this (and $y_r = 0$) would mean that the dipath to v determined by p would have cost exactly y_v. A spanning tree encountered by the simplex method need not have the property that every node other than r is the head of exactly one arc, but if it does encounter such a tree, then there is always a choice of the arc to delete (namely, the arc of the tree having the same head as the incoming arc) so that the property is kept. So (a version of) the simplex method moves from spanning tree to spanning tree, as does Ford's Algorithm, but the former method keeps the path costs determined by the current tree. In fact Ford's Algorithm may do a correction step on an arc of the form $p(v)v$, so that the tree does not change, but y does. In this sense, each step of the simplex algorithm could be regarded as a sequence of steps of Ford's Algorithm, one ordinary step followed by several steps that do not change the tree, until y "catches up" to the tree. We will learn more about such "network" versions of the simplex method in Chapter 4.

Refinements of Ford's Algorithm

The basic step of Ford's Algorithm could be written as

> Choose an arc e;
> If e is incorrect, then correct it;

Notice that, assuming that we store the values of y and p appropriately, we can perform each basic step in constant time. But the number of basic steps depends crucially on the order in which arcs e are chosen. Suppose that arcs are chosen in the sequence $f_1, f_2, f_3, \ldots, f_\ell$, which we denote by $\mathcal{S}$. (In general, there will be repetitions.) There are choices for $\mathcal{S}$ that result in very bad performance of the algorithm. (For example, see Exercise 2.25.) The basic idea for looking for good choices for $\mathcal{S}$ is simple. Denote by P the dipath $v_0, e_1, v_1, \ldots, e_k, v_k$ from $r = v_0$ to $v = v_k$. After the first time that Ford's Algorithm considers the arc e_1 we will have $y_{v_1} \leq y_r + c_{e_1} \leq c_{e_1}$. After the first subsequent time that the algorithm considers e_2, we will have $y_{v_2} \leq y_{v_1} + c_{e_2} \leq c_{e_1} + c_{e_2}$. Continuing, once $e_1, e_2, \ldots, e_k$ have been considered *in that order*, we will have $y_v \leq c(P)$. We say that P is *embedded* in $\mathcal{S}$ if its arcs occur (in the right order, but not necessarily consecutively) as a subsequence of $\mathcal{S}$. Our discussion can be summarized as follows.

Proposition 2.16 *If Ford's Algorithm uses the sequence $\mathcal{S}$, then for every $v \in V$ and for every path P from r to v embedded in $\mathcal{S}$, we have $y_v \leq c(P)$.* ∎

It follows that, if $\mathcal{S}$ has the property that for every node v there is a least-cost dipath to v embedded in $\mathcal{S}$, then $\mathcal{S}$ will bring the algorithm to termination. We want $\mathcal{S}$ to have this property, and we also want $\mathcal{S}$ to be short, since its length will be the running time of the algorithm.

The Ford-Bellman Algorithm

A simple way to use Proposition 2.16 is to observe that *every* simple dipath in G is embedded in $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{n-1}$, where for each i, $\mathcal{S}_i$ is an ordering of E. When we use such an ordering in Ford's Algorithm we speak of a sequence of "passes" through E. Since each arc is handled in constant time per pass, we get a shortest path algorithm that runs in time $O(mn)$. We call it the Ford-Bellman Algorithm, because Bellman [1958] seems to have been the first to prove a polynomial bound for such an algorithm. We want the algorithm also to recognize whether there exists a negative-cost dicircuit. We know that, if there is no negative-cost dicircuit, then $n - 1$ passes are sufficient to determine a feasible potential. Therefore, if y is not a feasible potential after $n - 1$ passes, then there exists a negative-cost dicircuit.

Ford-Bellman Algorithm

Initialize y, p;
Set $i = 0$;
While $i < n$ and y is not a feasible potential
    Replace i by $i + 1$;
    For $e \in E$
        If e is incorrect
            Correct e.

Theorem 2.17 *The Ford-Bellman Algorithm correctly computes a least-cost dipath from r to v for all $v \in V$ (if $i < n$ at termination), or correctly detects that there is a negative-cost dicircuit (if $i = n$ at termination). In either case it runs in time $O(mn)$.* ∎

We shall see that the running time of $O(mn)$ can be improved if special assumptions are made about G or c. However, in the general case, no better bound is currently known. Here we mention some further refinements that speed up the algorithm in practice. Most of them are based on the natural idea of scanning nodes, that is, considering consecutively all the arcs having the same tail. (We point out that the simplex method is not based on scanning. For another such example, see Exercise 2.42.)

A usual representation of a digraph is to store all the arcs having tail v in a list L_v. To *scan* v means to do the following:

For $vw \in L_v$
    If vw is incorrect
        Correct vw;

A natural further refinement of the Ford-Bellman Algorithm is to replace the last three lines in its statement by

For $v \in V$
    Scan v;

It is obvious that, if y_v has not decreased since the last time v was scanned, then v need not be scanned. Taking advantage of this observation saves time. One way to do that is to keep a set Q of nodes to be scanned, adding a node v to Q when y_v is decreased (if $v \notin Q$) and choosing the next node to be scanned from Q (and deleting it from Q). Initially $Q = \{r\}$, and the algorithm terminates when Q becomes empty. We keep Q both as a list and

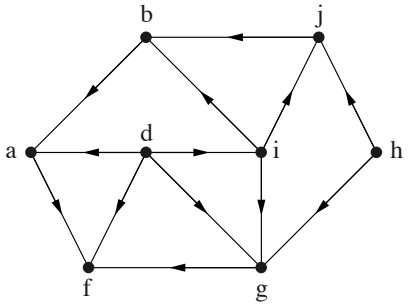


Figure 2.5. Digraph having a topological sort

a characteristic vector, so that we can add, delete, and check membership in Q in constant time. In order to do the first two operations in constant time, we add and delete at the ends of Q. If we add at one end, the "back," and delete at the other, the "front," we are keeping Q as a "first in, first out" list or a *queue*. In this case it can be checked (Exercise 2.32) that the algorithm is equivalent to a refinement of the Ford-Bellman Algorithm and so has a running time of $O(mn)$. This refinement also works well in practice, but there are some variants that are even faster. The paper of Gallo and Pallottino [1986] contains more information.

Acyclic Digraphs

Suppose that the nodes of G can be ordered from left to right so that all arcs go from left to right. More precisely, suppose that there is an ordering $v_1, v_2, \ldots, v_n$ of V so that $v_i v_j \in E$ implies $i < j$. We call such an ordering a *topological sort*. In the digraph of Figure 2.5, d, h, i, g, j, b, a, f is a topological sort.

If we order E in the sequence $\mathcal{S}$ so that $v_i v_j$ precedes $v_k v_\ell$ if $i < k$, then *every* dipath of G is embedded in $\mathcal{S}$. It follows that Ford's Algorithm will solve the shortest path problem in just one pass through E. There is a simple description of the class of digraphs for which this observation works. It is obvious that if G has a topological sort, then it has no dicircuit at all (and hence no negative-cost dicircuit); in other words, G is *acyclic*. Conversely, we claim that every acyclic digraph has a topological sort. To see this, first observe that each acyclic digraph has a candidate for v_1, that is, a node v such that $uv \in E$ for *no* $u \in V$. (Why?) Moreover, since $G \backslash v$ is acyclic, this can be repeated. This idea can be turned into an $O(m)$ algorithm to find a topological sort (Exercise 2.33). Notice that, if $r = v_i$ with $i > 1$, then there can be no dipath from r to $v_1, \ldots, v_{i-1}$, so these can be deleted. Hence we may assume that $v_1 = r$.

Shortest Paths in an Acyclic Digraph

Find a topological sort $v_1, \ldots, v_n$ of G with $r = v_1$;
Initialize y, p;
For $i = 1$ to n
    Scan v_i.

Theorem 2.18 *The shortest path problem on an acyclic digraph can be solved in time $O(m)$.* ∎

Nonnegative Costs

In many applications of the shortest path problem we know that $c \geq 0$. In fact, probably this is the situation more often than not, so this is an extremely important special case. Again it is possible to design a correct "one-pass" algorithm. Moreover, the ordering is determined from an ordering of the nodes as in the acyclic case. However, this ordering is computed during the course of execution. Namely, if $v_1, v_2, \ldots, v_i$ have been determined and scanned, then v_{i+1} is chosen to be the unscanned node v for which y_v is minimum. In this situation we have the following result.

Proposition 2.19 *For each $w \in V$, let y'_w be the value of y_w when w is chosen to be scanned. If u is scanned before v, then $y'_u \leq y'_v$.*

Proof: Suppose $y'_v < y'_u$ and let v be the earliest node scanned for which this is true. When u was chosen to be scanned, we had $y'_u = y_u \leq y_v$, so y_v was lowered to a value less than y'_u after u was chosen to be scanned but before v was chosen. So y_v was lowered when some node w was scanned, and it was set to $y'_w + c_{wv}$. By choice of v, $y'_w \geq y'_u$ and since $c_{wv} \geq 0$, we have $y'_v \geq y'_u$, a contradiction. ∎

We claim that after all nodes are scanned, we have $y_v + c_{vw} \geq y_w$ for all $vw \in E$. Suppose not. Since this was true when v was scanned, it must be that y_v was lowered after v was scanned, say while q was being scanned. But then $y_v = y'_q + c_{qv} \geq y'_v$ since q was scanned later than v and $c_{qv} \geq 0$, a contradiction. So the following algorithm, due to Dijkstra [1959], is valid.

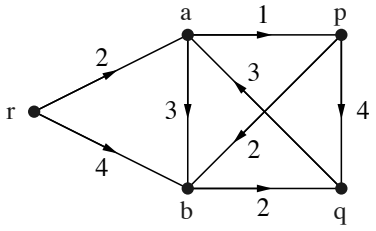


Figure 2.6. Example for Dijkstra's Algorithm

Dijkstra's Algorithm

Initialize y, p;
Set $S = V$;
While $S \neq \emptyset$
    Choose $v \in S$ with y_v minimum;
    Delete v from S;
    Scan v.

For example, in the digraph of Figure 2.6, the nodes will be scanned in the order r, a, p, b, q.

Actually, one can slightly improve the algorithm by observing that, for $w \notin S$, $y_v + c_{vw} \geq y_w$ follows from $y_v \geq y_w$. So the test that $y_v + c_{vw} < y_w$ could be done only for $w \in S$. The running time of the algorithm is $O(m)$ plus the time to find v. But this simple step requires considerable time: $k-1$ comparisons when $|S| = k$ and $n-1+n-2+\ldots+1 = O(n^2)$ comparisons in all. So the running time of a straightforward implementation is $O(n^2)$.

Theorem 2.20 *If $c \geq 0$, then the shortest path problem can be solved in time $O(n^2)$.* ∎

A number of improvements are discussed in Tarjan [1983].

Shortest path problems with nonnegative costs arise frequently in applications, so it is convenient to have a notation for the time required to solve them. We use $S(n, m)$ for the time needed to solve a nonnegative-cost shortest path problem on a digraph having n nodes and m arcs.

Feasible Potentials and Nonnegative Costs

If we happen to know a feasible potential y, we can use it to transform the cost vector c to a nonnegative one c'. Namely, put $c'_{vw} = c_{vw} + y_v - y_w$. This does not affect the least-cost dipaths, since any (r, s)-dipath P satisfies $c'(P) = c(P) + y_r - y_s$. Hence Dijkstra's Algorithm can be used.

There are several applications of this simple idea. We shall see an important one in Chapter 4. Meanwhile, here is another useful one. The "all pairs" shortest path problem is to find a least-cost dipath from r to v for every choice of r and v. There are direct algorithms for this problem but, from the point of view of running time, it seems to be better just to use a standard algorithm n times. Hence we get a running time of $O(nS(n,m))$ in the case of nonnegative costs, and $O(n^2m)$ in general. But the latter time can be improved. We find a feasible potential in time $O(nm)$ with Ford-Bellman, then transform to nonnegative costs, and then use Dijkstra n (or $n-1$) times, resulting in an overall running time of $O(nS(n,m))$.

Unit Costs and Breadth-First Search

The problem of finding a dipath from r to v having as few arcs as possible is, of course, a kind of shortest path problem, namely, it is the case where all arc-costs are 1. It is interesting to see how Dijkstra's Algorithm can be improved in this situation.

Proposition 2.21 *If each $c_e = 1$, then in Dijkstra's Algorithm the final value of y_v is the first finite value assigned to it. Moreover, if v is assigned its first finite y_v before w is, then $y_v \leq y_w$.*

Proof: Notice that these statements are obviously true for $v = r$. If $v \neq r$, the first finite value assigned to y_v is $y'_w + 1$, where y'_w is the final value of y_w. Moreover, any node j scanned later than w has $y'_j \geq y'_w$ by Proposition 2.19, so y_v will not be further decreased. Similarly, any node q assigned its first finite y_q after v, will have $y_q = y'_j + 1 \geq y'_w + 1 = y_v$. ∎

When picking $v \in S$ such that y_v is minimum, we choose among the set Q of those unscanned nodes v having y_v finite. Proposition 2.21 tells us that we can simply choose the element of Q that was added to Q first, that is, we can keep Q as a queue, and v can be found in constant time. So Dijkstra's Algorithm has a running time of $O(m)$ in this case. Notice that we no longer need to maintain the y_v (although we may want to). This algorithm is often called *breadth-first search*.

Breadth-first Search

```
Initialize p;
Set Q = {r};
While Q ≠ ∅
    Delete v from the front of Q;
    For vw ∈ L(v)
        If p(w) = −1
            Add w to the back of Q;
            Set p(w) = v.
```

Exercises

2.18. Show by an example that a spanning directed tree rooted at r can be of minimum cost but not contain least-cost dipaths to all nodes. Also show the converse, that it may contain least-cost dipaths but not be of minimum cost.

2.19. Show by an example that a subpath of a shortest simple dipath need not be a shortest simple dipath, if a negative-cost dicircuit exists.

2.20. Modify Ford's Algorithm (in a simple way) so that it always terminates, recognizing the existence of a negative-cost dicircuit if there is one.

2.21. Solve the shortest path problem for the digraph described by the following lists, using (a) Ford-Bellman using node-scanning and a queue; (b) the acyclic algorithm; (c) Dijkstra. $V = \{r, a, b, d, f, g, h, j, k\}$, and for each $v \in V$ the elements of the list L_v are the pairs (w, c_{vw}) for which $vw \in E$. $L_r : (a, 2), (k, 7), (b, 5)$. $L_a : (d, 8), (f, 4)$. $L_b : (k, 3), (f, 2)$. $L_d : (h, 5)$. $L_f : (g, 3), (j, 7)$. $L_g : (h, 4), (j, 3)$. $L_j : (k, 4), (h, 3)$. $L_k : (d, 2), (h, 9), (g, 6), (f, 1)$. ($L_h$ is empty.)

2.22. We are given a digraph $G = (V, E)$, $c \in \mathbf{R}^E$, and disjoint sets $R,\ S \subseteq V$. The problem is to find a least-cost dipath joining a node in R to a node in S. Show that this problem can be reduced to an ordinary shortest path problem.

2.23. Suppose that we are given a shortest path problem on a digraph G such that a node w is incident with exactly two arcs. Explain how the solution of a shortest path problem on a smaller digraph yields the solution to the given problem.

2.24. There are certain street corners in Bridgetown such that the street on which a car leaves the intersection may depend on the street on which it entered (for example, "no left turn"). How can a digraph, and arc costs, be defined so that the dipaths correspond to legal routes?

2.25. (Edmonds) Consider the digraph G_k of Figure 2.7. Show that Ford's Algorithm (in fact, the simplex method) can take more than 2^k steps to solve the shortest path problem on G_k. Hint: Use induction. Try to make the algorithm solve the problem on G_{k-1} twice.

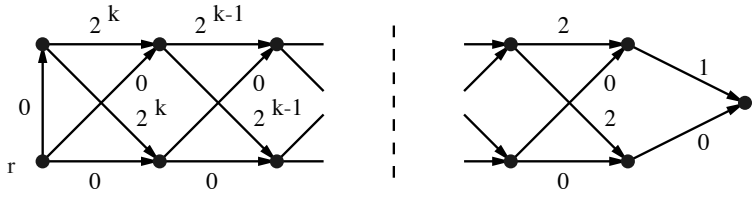


Figure 2.7. A bad example for the Ford and simplex algorithms

2.26. Prove that the problem of finding a least-cost simple dipath joining two fixed nodes in a digraph is hard, assuming only that the traveling salesman problem is hard.

2.27. Prove Proposition 2.13.

2.28. Prove Proposition 2.15. Hint: To prove that if T does not contain the arc-set of a circuit, then the corresponding columns are linearly independent, use the fact that if a forest has at least one arc, then there is a node incident to just one of its arcs.

2.29. Generalize Proposition 2.9 in the following way. Suppose that we have dipath costs $y_v,\ v \in V$ such that for every arc vw we have $y_w - c_{vw} - y_v \leq K$. Prove that for each node v, y_v is within Kn of being optimal.

2.30. A "scaling" approach to improving the complexity of Ford's Algorithm for integral arc-costs might work as follows. For some integer p suppose that in "stage p," we do correction steps only for arcs vw that satisfy $y_w - y_v - c_{vw} \geq 2^p$. If there are no more arcs of this sort, then we decrease p by 1. When we get to $p = 0$ we are just doing Ford's Algorithm. Use Exercise 2.29 to prove a bound on the number of steps in each stage after the first one. Then choose the first value of p so that the bound also applies for the first stage. What is the overall running time?

2.31. Here is a variant on the approach of doing only large correction steps in Ford's Algorithm. Suppose that at each step we choose to correct the arc that maximizes $y_w - c_{vw} - y_v$. (Of course, this requires some extra work to identify the arc.) Let $gap(k)$ denote the difference between the value after k iterations of $\sum(y_v \ : \ v \in V)$ and its minimum value. What will happen in the first $n-1$ iterations? To analyze the number of subsequent iterations, use the result of Exercise 2.29 to prove that $gap(k+1) \leq gap(k)(1-1/n^2)$, and hence prove a bound on the number of steps. (You may need the inequality $1 - x \leq e^{-x}$.)

2.32. Prove that the version of Ford's Algorithm that uses node-scanning and a queue to store the nodes to be scanned, has a running time of $O(nm)$.

2.33. Give an $O(m)$ algorithm to find in a digraph either a dicircuit or a topological sort.

2.34. We are given numbers $a_1, \ldots, a_n$. We want to find i and j, $1 \leq i \leq j \leq n+1$, so that $\sum_{k=i}^{j-1} a_k$ is minimized. Give an $O(n)$ algorithm.

2.35. Suppose that we are given tasks $t_1, t_2, \ldots, t_k$. Each task t_i has a processing time p_i. For certain pairs (i,j), t_i *must precede* t_j, that is, the processing of t_j cannot begin until the processing of t_i is completed. We wish to schedule the processing of the tasks so that all of the tasks are completed as soon as possible. Solve this problem as a maximum feasible potential problem on an acyclic digraph.

2.36. Give an example to show that Dijkstra's Algorithm can give incorrect results if negative costs are allowed.

2.37. Consider the least cost path problem for undirected graphs. Show that if the costs can be assumed to be nonnegative, then this problem can be solved by reducing it to a digraph problem. When costs are allowed to be negative, what difficulty arises?

2.38. Consider the problem of finding a minimum cost dipath with an odd (even) number of arcs from r to s in a digraph G having nonnegative arc costs. Notice that the dipath may not be simple. Show how to solve this problem by solving a shortest path problem in a digraph having two nodes for each node different from r and s.

2.39. Consider the *minmax path problem*: Given a digraph G with arc-costs and nodes r and s, find an (r,s) dipath P whose maximum arc-cost is as small as possible. Show how the idea of Exercise 2.17 can be applied to solve this problem. What is the running time?

2.40. Try to adapt Dijkstra's Algorithm to solve the minmax path problem. Prove that your algorithm works and give the running time.

2.41. Describe a direct all-pairs shortest-path algorithm based on the following idea. Given a set $S \subseteq V$, let y_{vw}, for $v,\ w \in V$, denote the least cost of a (v,w)-dipath whose internal nodes are from S. Compute this information beginning with $S = \emptyset$, adding one node at a time to S until $S = V$.

2.42. Consider the following refinement of Ford's Algorithm. Let $v_1, \ldots, v_n$ be an ordering of V, with $r = v_1$. Split E into E_1 and E_2, where $E_1 = \{v_i v_j : i < j\}$. Now order E_1 into a sequence $\mathcal{S}_1$, so that $v_i v_j$ precedes $v_k v_\ell$ if $i < k$ and order E_2 into a sequence $\mathcal{S}_2$ so that $v_i v_j$ precedes $v_k v_\ell$ if $i > k$. Now use the sequence $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_1, \mathcal{S}_2, \ldots$ in Ford's Algorithm. How does the running time compare to that of Ford-Bellman?

Bibliography

[1974] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1974.

[1987] R.P. Anstee, "A polynomial algorithm for b-matchings: An alternative approach," *Information Processing Letters* 24 (1987) 153–157.

[1995] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "Finding cuts in the TSP," *DIMACS Technical Report* 95-05, 1995.

[1993] D. Applegate and W. Cook, "Solving large-scale matching problems," in: *Algorithms for Network Flows and Matching* (D.S. Johnson and C.C. McGeoch, eds.), American Mathematical Society, 1993, pp. 557–576.

[1983] J. Aráoz, W.H. Cunningham, J. Edmonds, and J. Green-Krótki, "Reductions to 1-matching polyhedra," *Networks* 13 (1983) 455–483.

[1983] M.O. Ball and U. Derigs, "An analysis of alternative strategies for implementing matching algorithms," *Networks* 13 (1983) 517–549.

[1989] F. Barahona and W.H. Cunningham, "On dual integrality in matching problems," *Operations Research Letters* 8 (1989) 245–249

[1958] R.E. Bellman, "On a routing problem," *Quarterly of Applied Mathematics* 16 (1958) 87–90.

[1957] C. Berge, "Two theorems in graph theory," *Proceedings of the National Academy of Sciences* (U.S.A.) 43 (1957) 842–844.

[1958] C. Berge, "Sur le couplage maximum d'un graphe," *Comptes Rendus de l'Académie des Sciences Paris, series 1, Mathématique* 247 (1958), 258–259.

[1995] D. Bertsimas, C. Teo, and R. Vohra, "Nonlinear relaxations and improved randomized approximation algorithms for multicut problems," in: *Proceedings of the 4th IPCO Conferences* (E. Balas and J. Clausen, eds.), *Lecture Notes in Computer Science* 920, Springer, 1995, pp. 29–39.

[1946] G. Birkhoff, "Tres observaciones sobre el algebra lineal," *Revista Facultad de Ciencias Exactas, Puras y Aplicadas Universidad Nacional de Tucuman, Serie A (Matematicas y Fisica Teorica)* 5 (1946) 147–151.

[1961] R.G. Busacker and P.J. Gowen, "A procedure for determining a family of minimal cost network flow patterns," *Technical Paper* 15, Operations Research Office, 1961.

[1983] R.W. Chen, Y. Kajitani, and S.P. Chan, "A graph theoretic via minimization algorithm for two-layer printed circuit boards," *IEEE Transactions on Circuits and Systems* 30 (1983) 284–299.

[1997] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine, and C. Stein, "Experimental study of minimum cut algorithms," in: *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 324–333.

[1989] J. Cheriyan and S.N. Maheshwari, "Analysis of preflow push algorithms for maximum network flow," *SIAM Journal on Computing* 18 (1989) 1057–1086.

[1976] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1976.

[1973] V. Chvátal, "Edmonds polytopes and a hierarchy of combinatorial problems," *Discrete Mathematics* 4 (1973) 305–337.

[1973a] V. Chvátal, "Edmonds polytopes and weakly hamiltonian graphs," *Mathematical Programming* 5 (1973) 29–40.

[1983] V. Chvátal, *Linear Programming*, Freeman, New York, 1983.

[1985] V. Chvátal, "Cutting planes in combinatorics," *European Journal of Combinatorics* 6 (1985) 217–226.

[1989] V. Chvátal, W. Cook and M. Hartmann, "On cutting-plane proofs in combinatorial optimization," *Linear Algebra and its Applications* 114/115 (1989) 455–499.

[1997] W. Cook and A. Rohe, "Computing minimum-weight perfect matchings," Report Number 97863, Research Institute for Discrete Mathematics, Universität Bonn, 1997.

[1976] W.H. Cunningham,"A network simplex method," *Mathematical Programming* 11 (1976) 105–116.

[1979] W.H. Cunningham, "Theoretical properties of the network simplex method," *Mathematics of Operations Research* 4 (1979) 196–208.

[1978] W.H. Cunningham and A.B. Marsh III, "A primal algorithm for optimum matching," *Mathematical Programming Study* 8 (1978) 50–72.

[1954] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Operations Research* 2 (1954) 393–410.

[1991] U. Derigs and A. Metz, "Solving (large scale) matching problems combinatorially," *Mathematical Programming* 50 (1991) 113–122.

[1959] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik* 1 (1959) 269–271.

[1970] E.A. Dinits, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Mathematics Doklady* 11 (1970) 1277–1280.

[1987] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.

[1965] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics* 17 (1965) 449–467.

[1965a] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of Research of the National Bureau of Standards* (B) 69 (1965) 125–130.

[1970] J. Edmonds, "Matroids, submodular functions, and certain polyhedra," in: *Combinatorial Structures and Their Applications* (R.K. Guy, H. Hanani, N. Sauer, and J. Schönheim, eds.), Gordon and Breach, New York, 1970, pp. 69–87.

[1971] J. Edmonds, "Matroids and the greedy algorithm," *Mathematical Programming* 1 (1971) 127–136.

[1965] J. Edmonds and D.R.Fulkerson, "Transversal and matroid partition," *Journal of Research of the National Bureau of Standards* B 69 (1965) 147–153.

[1977] J. Edmonds and R. Giles, "A min-max relation for submodular functions on graphs," in: *Studies in Integer Programming* (P.L. Hammer, *et al.* eds.), *Annals of Discrete Mathematics* 1 (1977) 185–204.

[1970] J. Edmonds and E.L. Johnson, "Matching: A well-solved class of integer linear programs," in: *Combinatorial Structures and their Applications* (R.K. Guy, H. Hanani, N. Sauer, and J. Schönheim, eds.), Gordon and Breach, New York, 1970, pp. 89–92.

[1973] J. Edmonds and E.L. Johnson, "Matching, Euler tours, and the Chinese postman," *Mathematical Programming* 5 (1973) 88–124.

[1969] J. Edmonds, E.L. Johnson, and S.C. Lockhart, "Blossom I, a code for matching," unpublished report, IBM T.J. Watson Research Center, Yorktown Heights, New York.

[1972] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the Association for Computing Machinery* 19 (1972) 248–264.

[1956] L.R. Ford, Jr., "Network flow theory," Paper P-923, RAND Corporation, Santa Monica, California, 1956.

[1956] L.R. Ford, Jr. and D.R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics* 8 (1956) 399–404.

[1957] L.R. Ford, Jr. and D.R. Fulkerson, "A primal-dual algorithm for the capacitated Hitchcock problem," *Naval Research Logistics Quarterly* 4 (1957) 47–54.

[1958] L.R. Ford, Jr. and D.R. Fulkerson, "Suggested computation for maximal multi-commodity network flows," *Management Science* 5 (1958) 97–101.

[1962] L.R. Ford, Jr. and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey, 1962.

[1981] A. Frank, "A weighted matroid intersection algorithm," *Journal of Algorithms* 2 (1981) 328–336.

[1986] S. Fujishige, "A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework for the Tardos algorithm," *Mathematical Programming* 35 (1986) 298–308.

[1990] H. Gabow, "Data structures for weighted matching and nearest common ancestors," in: *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York, 1990, pp. 434–443.

[1957] D. Gale, "A theorem on flows in networks," *Pacific Journal of Mathematics* 7 (1957) 1073–1082.

[1986] G. Gallo and S. Pallottino, "Shortest path methods: A unifying approach," *Mathematical Programming Study* 26 (1986) 38–64.

[1979] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

[1995] A.M.H.Gerards, "Matching," in: *Network Models* (M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, eds.), North Holland, Amsterdam, 1995.

[1986] A.M.H. Gerards and A. Schrijver, "Matrices with the Edmonds-Johnson property," *Combinatorica* 6 (1986) 403–417.

[1979] F.R. Giles and W.R. Pulleyblank, "Total dual integrality and integer polyhedra," *Linear Algebra and its Applications* 25 (1979) 191–196.

[1995] M.X. Goemans and D.P. Williamson, "A general approximation technique for constrained forest problems," *SIAM Journal on Computing* 24 (1995) 296–317.

[1985] A.V. Goldberg, "A new max-flow algorithm," *Technical Report* MIT/-LCS/TM 291, Laboratory for Computer Science, M.I.T. (1985).

[1988] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *Journal of the Association for Computing Machinery* 35 (1988) 921–940.

[1989] A.V. Goldberg and R.E. Tarjan, "Finding minimum-cost circulations by canceling negative cycles," *Journal of the Association for Computing Machinery* 33 (1989) 873–886.

[1960] R.E. Gomory, "Solving linear programming problems in integers," in: *Combinatorial Analysis* (R. Bellman and M. Hall, eds.), *Proceedings of Symposia in Applied Mathematics* X, American Mathematical Society, Providence, 1960, pp. 211–215.

[1961] R.E. Gomory and T.C. Hu, "Multi-terminal network flows," *SIAM Journal on Applied Mathematics* 9 (1961) 551–556.

[1988] M. Grötschel and O. Holland, "Solution of large-scale symmetric travelling salesman problems," *Mathematical Programming* 51 (1991) 141–202.

[1988] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.

[1979] M. Grötschel and M.W. Padberg, "On the symmetric travelling salesman problem II: Lifting theorems and facets," *Mathematical Programming* 16 (1979) 282–302.

[1985] M. Grötschel and M.W. Padberg, "Polyhedral theory," in: *The Traveling Salesman Problem* (E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D. Shmoys, eds.), Wiley, Chichester, 1985, pp. 251–306.

[1986] M. Grötschel and W.R. Pulleyblank, "Clique tree inequalities and the symmetric travelling salesman problem," *Mathematics of Operations Research* 11 (1986) 537–569.

[1990] D. Gusfield, "Very simple methods for all pairs network flow analysis," *SIAM Journal on Computing* 19 (1990) 143–155.

[1975] F.O. Hadlock, "Finding a maximum cut in a planar graph in polynomial time," *SIAM Journal on Computing* 4 (1975) 221–225.

[1992] X. Hao and J.B. Orlin, "A faster algorithm for finding the minimum cut in a graph," *Proceedings of the 3rd SIAM-ACM Symposium on Discrete Algorithms*, 1992, pp. 165–174.

[1970] M. Held and R.M. Karp, "The traveling-salesman problem and minimum spanning trees," *Operations Research* 18 (1970) 1138–1162.

[1971] M. Held and R.M. Karp, "The traveling-salesman problem and minimum spanning trees: Part II," *Mathematical Programming* 1 (1971) 6–25.

[1974] M. Held, P. Wolfe, and H.P. Crowder, "Validation of subgradient optimization," *Mathematical Programming* 6 (1974) 62–88.

[1960] A.J. Hoffman, "Some recent applications of the theory of linear inequalities to extremal combinatorial analysis," *Proceedings of Symposia on Applied Mathematics* 10 (1960) 113–127.

[1974] A.J. Hoffman, "A generalization of max flow-min cut," *Mathematical Programming* 6 (1974) 352–359.

[1956] A.J. Hoffman and J.B. Kruskal, "Integral boundary points of convex polyhedra," in: *Linear Inequalities and Related Systems* (H.W. Kuhn and A.W. Tucker, eds.), Princeton University Press, Princeton, 1956, pp. 223–246.

[1987] O. Holland, *Schnittebenenverfahren für Travelling-Salesman- und verwandte Probleme*, Ph.D. Thesis, Universität Bonn, Germany, 1987.

[1973] J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *SIAM Journal on Computing* 2 (1973) 225–231.

[1930] V. Jarníik, "O jistém problému minimálníim," *Práce Moravské Přírodovědecké Společnosti* 6 (1930) 57–63.

[1997] D.S. Johnson, J.L. Bentley, L.A. McGeoch, and E.E. Rothberg, in preparation.

[1997] D.S. Johnson and L.A. McGeoch, "The traveling salesman problem: A case study in local optimization," in: *Local Search in Combinatorial Optimization* (E.H.L. Aarts and J.K. Lenstra, eds.), Wiley, New York, 1997, pp. 215–310.

[1993] M. Jünger and W. Pulleyblank, "Geometric duality and combinatorial optimization," in: *Jahrbuck Überblicke Mathematik* (S.D. Chatterji, B. Fuchssteiner, U. Kluish, and R. Liedl, eds.), Vieweg, Brunschweig/Wiesbaden, 1993, pp. 1–24.

[1995] M. Jünger, G. Reinelt, and G. Rinaldi, "The traveling salesman problem," in: *Handbook on Operations Research and Management Sciences: Networks* (M. Ball, T. Magnanti, C.L. Monma, and G. Nemhauser, eds.), North-Holland, 1995, pp. 225–330.

[1994] M. Jünger, G. Reinelt, and S. Thienel, "Provably good solutions for the traveling salesman problem," *Zeitschrift für Operations Research* 40 (1994) 183–217.

[1942] L.V. Kantorovich, "On the translocation of masses", CR de l'Academie des Sciences de l'URSS, 1942.

[1993] D.R. Karger, "Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm," in: *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, 1993, pp. 84–93.

[1993] D.R. Karger and C. Stein, "An $\tilde{\mathrm{O}}(n^2)$ algorithm for minimum cuts," in: *Proceedings of the 25th ACM Symposium on the Theory of Computing*, ACM Press, 1993. pp. 757–765.

[1972] R.M. Karp, "Reducibility among combinatorial problems," in: *Complexity of Computer Computations* (R.E. Miller and J.W. Thatcher, eds.), Plenum Press, New York, 1972, pp. 85–103.

[1931] D. König, "Graphok és matrixok," *Matematikai és Fizikai Lapok* 38 (1931) 116–119.

[1990] B. Korte, L. Lovász, H.J. Prömel, and A. Schrijver, eds., *Paths, Flows, and VLSI-Layout*, Springer, Berlin, 1990.

[1956] A. Kotzig, "Súvislost' a Pravideliná Súvislost' Konečných Grafor," Bratislava: *Vysoká Škola Ekonomická* (1956).

[1956] J.B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society* 7 (1956) 48–50.

[1955] H.W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly* 2 (1955) 83–97.

[1975] E.L. Lawler, "Matroid intersection algorithms," *Mathematical Programming* 9 (1975) 31–56.

[1985] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D. Shmoys, *The Traveling Salesman Problem*, Wiley, Chichester, 1985.

[1973] S. Lin and B.W. Kernighan, "An effective heuristic algorithm for the traveling salesman problem," *Operations Research* 21 (1973) 498–516.

[1979] L. Lovász, "Graph theory and integer programming," in: *Discrete Optimization I* (P.L. Hammer, E.L. Johnson, and B.H. Korte, eds), *Annals of Discrete Mathematics* 4 (1979) 146–158.

[1986] L. Lovász and M. Plummer, *Matching Theory*, North Holland, Amsterdam, 1986.

[1993] K.-T. Mak and A.J. Morton, "A modified Lin-Kernighan traveling-salesman heuristic," *Operations Research Letters* 13 (1993) 127–132.

[1979] A.B. Marsh III, *Matching Algorithms*, Ph.D. Thesis, Johns Hopkins University, Baltimore.

[1996] O. Martin and S.W. Otto, "Combining simulated annealing with local search heuristics," *Annals of Operations Research* 63 (1996) 57–75.

[1992] O. Martin, S.W. Otto, and E.W. Felten, "Large-step Markov chains for the TSP incorporating local search heuristics," *Operations Research Letters* 11 (1992) 219–224.

[1980] S. Micali and V.V. Vazirani, "An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs," *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, IEEE, 1980, pp. 17–27.

[1995] D.L. Miller and J.F. Pekney, "A staged primal-dual algorithm for perfect b-matching with edge capacities," *ORSA Journal on Computing* 7 (1995) 298–320.

[1957] J. Munkres, "Algorithms for the assignment and transportation problems," *SIAM Journal on Applied Mathematics* 5 (1957) 32–38.

[1990] D. Naddef, "Handles and teeth in the symmetric traveling salesman polytope," in: *Polyhedral Combinatorics* (W. Cook and P.D. Seymour, eds.), American Mathematical Society, 1990, pp. 61–74.

[1992] H. Nagamochi and T. Ibaraki, "Computing edge connectivity in multigraphs and capacitated graphs," *SIAM Journal on Discrete Mathematics* 5 (1992) 54–66.

[1966] C.St.J.A. Nash-Williams, "An application of matroids to graph theory," in: *Theory of Graphs* (P. Rosenstiehl, ed.) Dunod, Paris, 1966, pp. 263–265.

[1988] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, New York, 1988.

[1985] J.B. Orlin, "On the simplex algorithm for networks and generalized networks," *Mathematical Programming Study* 24 (1985) 166–178.

[1988] J.B. Orlin, "A faster strongly polynomial minimum cost flow algorithm," *Proceedings of the 20th ACM Symposium on Theory of Computing*, ACM Press, 1988, pp. 377–387.

[1992] J.G. Oxley, *Matroid Theory*, Oxford University Press, Oxford, 1992.

[1982] M.W. Padberg and M.R. Rao, "Odd minimum cut-sets and b-matchings," *Mathematics of Operations Research* 7 (1982) 67–80.

[1990] M. Padberg and G. Rinaldi, "An efficient algorithm for the minimum capacity cut problem," *Mathematical Programming* 47 (1990) 19–36.

[1991] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM Review* 33 (1991) 60–100.

[1977] C. Papadimitriou and K. Steiglitz, "On the complexity of local search for the traveling salesman problem," *SIAM Journal on Computing* 6 (1977) 76–83.

[1976] J.-C. Picard, "Maximal closure of a graph and applications to combinatorial problems," *Management Science* 22 (1976) 1268–1272.

[1984] R.Y. Pinter, "Optimal layer assignment for interconnect," *Journal VLSI Comput. Syst.* 1 (1984) 123–137.

[1900] H. Poincaré, "Second complément à l'analysis situs," *Proceedings of the London Mathematical Society* 32 (1900) 277–308.

[1967] B.T. Polyak, "A general method of solving extremum problems" (in Russian), Doklady Akademmi Nauk SSSR 174 (1) (1967) 33–36.

[1957] R.C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal* 36 (1957) 1389–1401.

[1973] W.R. Pulleyblank, *Faces of Matching Polyhedra*, Ph.D. Thesis, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, 1973.

[1974] W.R. Pulleyblank and J. Edmonds, "Facets of 1-matching polyhedra," in: *Hypergraph Seminar* (C. Berge and D. Ray-Chaudhuri, eds.), Springer, Berlin, 1974, pp. 214–242.

[1942] R. Rado, "A theorem on independence relations," *Quarterly Journal of Mathematics Oxford* 13 (1942) 83–89.

[1957] R. Rado, "A note on independence functions," *Proceedings of the London Mathematical Society* 7 (1957) 300–320.

[1991] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA Journal on Computing* 3 (1991) 376–384.

[1994] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications*, Springer-Verlag, Berlin, 1994.

[1970] J. Rhys, "A selection problem of shared fixed costs and network flows," *Management Science* 17 (1970) 200–207.

[1977] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II, "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal on Computing* 6 (1977) 563–581.

[1980] A. Schrijver, "On cutting planes," in: *Combinatorics 79, Part II* (M. Deza and I.G. Rosenberg, eds.), *Annals of Discrete Mathematics* 9 (1980) 291–296.

[1983] A. Schrijver, "Short proofs on the matching polytope," *Journal of Combinatorial Theory (Series B)* 34 (1983) 104–108.

[1983a] A. Schrijver, "Min-max results in combinatorial optimization," in: *Mathematical Programming, the State of the Art: Bonn 1982* (A. Bachem, M. Grötschel, and B. Korte, eds.), Springer-Verlag, Berlin, 1983, pp. 439–500.

[1984] A. Schrijver, "Total dual integrality from directed graphs, crossing families, and sub- and supermodular functions," in: *Progress in Combinatorial Optimization* (W.R. Pulleyblank, ed.), Academic Press, Toronto, 1984, pp. 315–361.

[1986] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, Chichester, 1986.

[1980] P.D. Seymour, "Decomposition of regular matroids," *Journal of Combinatorial Theory (Series B)* 28 (1980) 305–359.

[1981] P.D. Seymour, "On odd cuts and plane multicommodity flows," *Proceedings of the London Mathematical Society* (1981) 178–192.

[1977] T.H.C. Smith and G.L. Thompson, "A LIFO implicit enumeration search algorithm for the symmetric traveling salesman problem using Held and Karp's 1-tree relaxation," in: *Studies in Integer Programming* (P.L. Hammer , *et al.* eds.), *Annals of Discrete Mathematics* 1 (1977) 479–493.

[1994] M. Stoer and F. Wagner, "A simple min cut algorithm," to appear.

[1987] R. Tamassia, "On embedding a graph in the grid with the minimum number of bends," *SIAM Journal on Computing* 16 (1987) 421–444.

[1985] E. Tardos, "A strongly polynomial minimum cost circulation algorithm," *Combinatorica* 5 (1985) 247–255.

[1983] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.

[1994] L. Tunçel, "On the complexity of preflow-push algorithms for the maximum flow problem," *Algorithmica* 11 (1994) 353–359.

[1947] W.T. Tutte, "The factorization of linear graphs," *Journal of the London Mathematical Society* 22 (1947) 107–111.

[1954] W.T. Tutte, "A short proof of the factor theorem for finite graphs," *Canadian Journal of Mathematics* 6 (1954) 347–352.

[1965] W.T. Tutte, "Lectures on matroids," *Journal of Research of the National Bureau of Standards* (B) 69 (1965) 1–47.

[1968] A.F. Veinott and G.B. Dantzig, "Integral extreme points," *SIAM Review* 10 (1968) 371–372.

[1976] D.J.A. Welsh, *Matroid Theory*, Academic Press, London, 1976.

[1996] D.B. West, *Introduction to Graph Theory*, Prentice Hall, Upper Saddle River, 1996

[1973] N. Zadeh, "A bad network problem for the simplex method and other minimum cost flow algorithms," *Mathematical Programming* 5 (1973) 255–266.