13

MINIMUM SPANNING TREES

Chapter Outline

13.1 Introduction
13.2 Applications
13.3 Optimality Conditions
13.4 Kruskal’s Algorithm
13.5 Prim’s Algorithm
13.6 Solin’s Algorithm
13.7 Minimum Spanning Trees and Matroids
13.8 Minimum Spanning Trees and Linear Programming
13.9 Summary

13.1 INTRODUCTION

As we have seen repeatedly throughout earlier chapters, spanning trees play a central role within the field of network flows. In solving the shortest path problem in Chapters 4 and 5, we constructed (shortest path) spanning trees rooted at a source node. The simplex method for solving minimum cost flow problems that we discussed in Chapter 11 is a spanning tree manipulation algorithm that iteratively moves from one spanning tree to another, at each step introducing one arc into the spanning tree in place of another. As we have also seen in Chapter 11 (Theorem 11.3), minimum cost flow problems always have spanning tree solutions; therefore, in principle to solve any minimum cost network flow problem, including shortest path problems and maximum flow problems, we can always restrict our attention to spanning tree solutions. Since any network has only a finite number of spanning trees, we can view any network flow problem as a discrete optimization model and solve it in a finite number of iterations.

In this chapter we consider another spanning tree model, known as the minimum spanning tree problem. Recall that a spanning tree $T$ of a connected acyclic subgraph that spans all the nodes. Every spanning tree of $G$ has $n-1$ arcs (see Property 2.2). Given an undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ arcs and with a length or cost $c_e$ associated with each arc $(i, j) \in E$, we wish to find a spanning tree, called a minimum spanning tree, that has the smallest total cost (or length) of its constituent arcs, measured as the sum of costs of the arcs in the spanning tree. Note that minimal spanning trees differ from the shortest path tree that we have considered in Chapters 4 and 5 in the following two respects:

1. For the minimum spanning tree problem, the arcs are undirected. [Since the network is undirected, we refer to the arc between the node pair $i$ and $j$ as either $(i, j)$ or $(j, i).$ For the version of the shortest path problems that we considered previously, the networks were directed. This distinction is unimportant in one sense: We could easily have developed our prior results for shortest path problems using undirected graphs as well as directed graphs (see Section 2.4). Viewed in another way, however, this distinction is important: Finding a minimum spanning tree on a directed network with all paths directed away from a given root node (this structure is known as a rooted arborescence) is a much more difficult problem than the undirected minimum spanning tree problem.

2. Our objective functions for the minimum spanning tree problem and for the shortest path tree problem are quite different. For the minimal spanning tree problem, we count the cost of each arc exactly once; for the shortest path tree problem, we typically count the cost of some arcs several times: equal to the number of paths from the root node that pass through that arc (i.e., the number of shortest paths in the tree that contain that arc).

The minimum spanning tree problem arises in a number of applications, both as a stand-alone problem and as a subproblem in a more complex problem setting. We begin this chapter by describing several such applications. We next consider combinatorial based optimality conditions for assessing whether a given spanning tree is a minimum spanning tree. We consider two such optimality conditions. The first condition is based on comparing the cost of any tree arc with the other arcs contained in the cut defined by removing that arc from the tree. The other is based on comparing the cost of a non-tree arc with the tree arcs in the path that connects the endpoints of the non-tree arc. These two cut and path optimality conditions are easy to state and to develop, yet they quite naturally motivate several algorithms for solving the minimum spanning tree problem.

The resulting algorithms are all very simple, although implementing them efficiently requires considerable care and ingenuity. The three algorithms we consider in this chapter—Kruskal’s algorithm, Prim’s algorithm, and Solin’s algorithm—all share one characteristic: They are “greedy” algorithms in the sense that at each step they add an arc of minimum cost from a candidate list, as long as the added arc does not form a cycle with the arcs already chosen. All three algorithms maintain a forest containing arcs already chosen and then they add one or more arcs to enlarge the size of the forest. For Kruskal’s algorithm, the candidate list is the entire network; for Prim’s algorithm, the forest is a single tree plus a set of isolated nodes and the candidate list contains all the arcs between the single tree and the nodes not in the tree; Solin’s algorithm is a hybrid approach that maintains several components in the forest, as in Kruskal’s algorithm, but then adds several arcs at each iteration, choosing (like Prim’s algorithm) the minimum cost arc connecting each component of the forest to the nodes not in that component.

Since greedy algorithms, such as Kruskal’s, Prim’s, and Solin’s, arise in many...
other problem contexts in discrete optimization, in Section 13.7 we show how a
generalization of Kruskal's algorithm will solve a broad class of abstract combi-
natorial optimization problems known as matroid optimization problems. This dis-
cussion not only permits us to show how to solve a new class of combinatorial
optimization problems, but also provides additional insight concerning the com-
natorial structure of spanning trees that underlies the validity of the greedy solution
approach.

Mathematical programming has another useful way to view the minimum span-
ing tree problem. In Section 13.8 we formulate the minimal spanning tree problem
as an integer programming model and use linear programming arguments to establish
yet another proof of the validity of Kruskal's algorithm. This discussion serves sev-
eral purposes: (1) it gives another useful view of minimum spanning trees; (2) it
illustrates a proof technique, via linear programming, that has proved to be very
powerful in the field of combinatorial optimization; and (3) it provides a bridge
between the minimum spanning tree problem and an important topic in discrete
optimization, polyhedral combinatorics (i.e., the study of integer polyhedra).

In closing this section we might note that we can also define and study the
maximum spanning tree problem, which as its name implies, seeks the spanning tree
with the largest total costs of its constituent arcs. Since we can find a maximum
spanning tree by multiplying all the arc costs by -1 and then solving a minimum
spanning tree, the algorithms and theory of the maximum spanning tree problem are
essentially the same as those of the minimum spanning tree problem.

13.2 APPLICATIONS

Minimum spanning tree problems generally arise in one of two ways, directly or
indirectly. In some direct applications, we wish to connect a set of points using the
least cost or least length collection of arcs. Frequently, the points represent physical
entities such as components of a computer chip, or users of a system who need to
be connected to each other or to a central service such as a central processor in a
computer system. In indirect applications, we either (1) wish to connect some set
of points using a measure of performance that on the surface bears little resemblance
to the minimum spanning tree objective (sum of arc costs), or (2) the problem itself
bears little resemblance to an "optimal tree" problem—in these instances, we often
need to be creative in modeling the problem so that it becomes a minimum spanning
tree problem. In this section we consider several direct and indirect applications.

Application 13.1 Designing Physical Systems

The design of physical systems can be a complex task involving an interplay between
performance objectives (such as throughput and reliability), design costs and op-
erating economics, and available technology. In many settings, the major criterion
is fairly simple: We need to design a network that will connect geographically dis-
persed system components or that will provide the infrastructure needed for users
to communicate with each other. In many of these settings, the system need not
have any redundancy, so we are interested in the simplest possible connection,
namely, a spanning tree. This type of application arises in the construction (or in-
stallation) of numerous physical systems: highways, computer networks, leased-line
telephone networks, railroads, cable television lines, and high-voltage electrical
power transmission lines. For example, this type of minimum spanning tree problem
arises in the following problem settings:

1. Connect terminals in cabling the panels of electrical equipment. How should we
wire the terminals to use the least possible length of the wire?
2. Constructing a pipeline network to connect a number of towns using the small-
est possible total length of pipeline.
3. Linking isolated villages in a remote region, which are connected by roads but
not yet by telephone service. In this instance we wish to determine along which
branches of roads we should place telephone lines, using the minimum possible
total miles of the lines, to link every pair of villages.
4. Constructing a digital computer system, composed of high-frequency circuitry,
when it is important to minimize the length of wires between different compo-
nents to reduce both capacitance and delay line effects. Since all components
must be connected, we obtain a spanning tree problem.
5. Connecting a number of computer sites by high-speed lines. Each line is avail-
ble for leasing at a certain monthly cost, and we wish to determine a con-
figuration that connects all the sites at the least possible cost.

Each of these applications is a direct application of the minimum spanning tree
problem. We next describe several indirect applications.

Application 13.2 Optimal Message Passing

An intelligence service has n agents in a nonfriendly country. Each agent knows
some of the other agents and has in place procedures for arranging a rendezvous
with anyone he knows. For each such possible rendezvous, say between agent i and
agent j, any message passed between these agents will fall into hostile hands with
a certain probability pu. The group leader wants to transmit a confidential message
among all the agents while minimizing the total probability that the message is
intercepted.

If we represent the agents by nodes, and each possible rendezvous by an arc,
then in the resulting graph G we would like to identify a spanning tree T that mini-
mizes the probability of interception given by the expression \(1 - \Pi_{i,j \in T} (1 - p_{ij})\). Alternatively, we would like to find a tree T that maximizes \(\Pi_{i,j \in T} (1 - p_{ij})\). We can identify such a tree by defining the length of an arc (i, j) as \(\log(1 - p_{ij})\) and solving a maximum spanning tree problem.

Application 13.3 All-Pairs Minimax Path Problem

The minimax path problem is a variant of the maximum capacity path problem that
we discussed in Exercise 4.37. In a network G = (N, A) with arc costs \(c_{ij}\), we define
the value of a path P from node k to node l as the maximum cost arc in P. The all-
pairs minimax path problem requires that we determine, for every pair (k, l) of nodes,
a minimum value path from node k to node l. We show how to solve the all-pairs

Sec. 13.2 Applications

513
minimax path problem on an undirected graph by solving a single minimum spanning tree problem.

The minimax path problem arises in a variety of situations. As an example, consider a spacecraft that is about to enter the earth’s atmosphere. The craft passes through different pressure and temperature zones that we can represent by arcs of a network. It needs to fly along a trajectory that will bring the craft to the surface of the earth while keeping the maximum temperature to which the surface of the craft is exposed as low as possible. As an alternative, we might wish to select a path that will minimize the maximum deceleration during the descent. Other examples of the minimax path problem arise when (1) in traveling through a desert, we want to minimize the length of the longest stretch between rest areas; and (2) in traveling in a wheelchair, a person might wish to minimize the maximum ascent along the path segments.

To transform the all-pairs minimax path problem into a minimum spanning tree problem, let $T^*$ be a minimum spanning tree of $G$. Let $P$ denote the unique path in $T^*$ between a node pair $[p, q]$ and let $(i, j)$ denote the maximum cost arc in $P$. Observe that the value of the path $P$ is $c_{ij}$. By deleting arc $(i, j)$ from $T^*$, we partition the node set $N$ into two subsets and therefore define a cut $[S, \bar{S}]$ with $i \in S$ and $j \in \bar{S}$ (see Figure 13.1). We later show in Theorem 13.1 that this cut satisfies the following property:

$$c_{ij} \leq c_{kl} \quad \text{for each arc } (k, l) \in [S, \bar{S}], \quad (13.1)$$

for otherwise by replacing the arc $(i, j)$ by an arc $(k, l)$ we can obtain a spanning tree of smaller cost. Now, consider any path $P'$ from node $p$ to node $q$. This path must contain at least one arc $(k, l)$ in $[S, \bar{S}]$. Property (13.1) implies that the value of the path $P'$ will be at least $c_{ij}$. Since $c_{ij}$ is the value of the path $P$, $P$ must be a minimum value path from node $p$ to node $q$. This observation establishes the fact that the unique path between any pair of nodes in $T^*$ is the minimum value path between that pair of nodes.

**Application 13.4 Reducing Data Storage**

In several different application contexts, we wish to store data specified in the form of a two-dimensional array more efficiently than storing all the elements of the array.

![Figure 13.1 Cut formed by deleting the arc $(i, j)$ from a spanning tree.](image)

(to save memory space). We assume that the rows of the array have many similar entries and differ only at a few places. One such situation arises in the sequence of amino acids in a protein found in the mitochondria of different animals and higher plants.

Since the entities in the rows are similar, one approach for saving memory is to store one row, called the reference row, completely, and to store only the differences between some of the rows so that we can derive each row from these differences and the reference row. Let $c_{ij}$ denote the number of different entries in rows $i$ and $j$; that is, if we are given row $i$, then by making $c_{ij}$ changes to the entries in this row we can obtain row $j$, and vice versa. Suppose that the array contains four rows, represented by $R_1, R_2, R_3$, and $R_4$, and we decide to treat $R_1$ as a reference row. Then one plausible solution is to store the differences between $R_1$ and $R_2$ and $R_3$ and $R_4$. Clearly, from this solution, we can obtain rows $R_2$ and $R_3$ by making $c_{12}$ and $c_{13}$ changes to the elements in row $R_1$. Having obtained row $R_2$, we can make $c_{24}$ changes to the elements of this row to obtain $R_4$.

It is easy to see that it is sufficient to store differences between those rows that correspond to arcs of a spanning tree. These differences permit us to obtain each row from the reference row. The total storage requirement for a particular storage scheme will be the length of the reference row (which we can take as the row with the least amount of data) plus the sum of the differences between the rows. Therefore, a minimum spanning tree would provide the least cost storage scheme.

**Application 13.5 Cluster Analysis**

The essential issue in cluster analysis is to partition a set of data into “natural groups”; the data points within a particular group of data, or a cluster, should be more “closely related” to each other than the data points not in that cluster. Cluster analysis is important in a variety of disciplines that rely on empirical investigations. Consider, for example, an instance of a cluster analysis arising in medicine. Suppose that we have data on a set of 350 patients, measured with respect to 18 symptoms. Suppose, further, that a doctor has diagnosed all of these patients as having the same disease, which is not well understood. The doctor would like to know if he can develop a better understanding of this disease by categorizing the symptoms into smaller groupings that can be detected through cluster analysis. Doing so might permit the doctor to find more natural disease categories to replace or subdivide the original disease.

In this section we describe the use of spanning tree problems to solve a class of problems that arise in the context of cluster analysis. Suppose that we are interested in finding a partition of a set of $n$ points in two-dimensional Euclidean space into clusters. A popular method for solving this problem is by using Kruskal's algorithm for solving the minimum spanning tree problem (we describe this method in Section 13.4). As we will show, at each intermediate iteration, Kruskal's algorithm maintains a forest (i.e., a collection of node-disjoint trees) and adds arcs in non-decreasing order of their lengths. We can regard the nodes spanned by the trees at intermediate steps as different clusters. These clusters are often excellent solutions for the clustering problem, and moreover, we can obtain them very efficiently. Kruskal's algorithm can be thought of as providing $n$ partitions: The first partition contains
\( n \) clusters, each cluster containing a single point, and the last partition contains just one cluster containing all the points. Alternatively, we can obtain \( n \) partitions by starting with a minimum spanning tree and deleting tree arcs one by one in nonincreasing order of their lengths. We illustrate the latter approach using an example. Consider a set of 27 points shown in Figure 13.2(a). Suppose that the network in Figure 13.2(b) is a minimum spanning tree for these points. Deleting the three largest length arcs from the minimum spanning tree gives a partition with four clusters shown in Figure 13.2(c).

![Figure 13.2 Identifying clusters by finding a minimum spanning tree.](image)

Analysts can use the information obtained from the preceding analysis in several ways. The procedure we have described yields \( n \) partitions. Out of these, we might select the "best" partition by simple visualization or by defining an appropriate objective function value. A good choice of the objective function depends on the underlying features of the particular clustering application. We might note that this analysis is not limited to points in two-dimensional space; we can easily extend it to multidimensional space if we define interpoint distances appropriately.

### 13.3 Optimality Conditions

As in our earlier discussion of network flow algorithms, optimality conditions for the minimum spanning tree problem play a central role in developing algorithms and establishing their validity. For the minimum spanning tree problem, we can formulate the optimality conditions in two important ways: cut optimality conditions and path optimality conditions. Needless to say, both optimality conditions are equivalent. Before considering these conditions, let us establish some further notation and illustrate some basic concepts.

The subgraphs shown in Figures 13.3(b) and 13.3(c) are spanning trees for the network shown in Figure 13.3(a). However, the subgraph shown in Figure 13.3(d) is not a spanning tree because it is not connected, and the subgraph shown in Figure 13.3(e) is not a spanning tree because it contains a cycle 1–3–4–1. We refer to those arcs contained in a given spanning tree as tree arcs and to those arcs not contained in a given spanning tree as non-tree arcs. The following two elementary observations will arise frequently in our development in this chapter.

1. For every non-tree arc \((k, l)\), the spanning tree \(T\) contains a unique path from node \(k\) to node \(l\). The arc \((k, l)\) together with this unique path defines a cycle [see Figure 13.4(a)].
2. If we delete any tree arc \((i, j)\) from a spanning tree, the resulting graph partitions the node set \(N\) into two subsets [see Figure 13.4(b)]. The arcs from the un-

![Figure 13.3 Illustrating spanning trees: (a) underlying graph; (b) and (c) two spanning trees; (d) non-spanning tree (doesn’t span all nodes); (e) another non-spanning tree (cyclic graph).](image)

1. For every non-tree arc \((k, l)\), the spanning tree \(T\) contains a unique path from node \(k\) to node \(l\). The arc \((k, l)\) together with this unique path defines a cycle [see Figure 13.4(a)].
2. If we delete any tree arc \((i, j)\) from a spanning tree, the resulting graph partitions the node set \(N\) into two subsets [see Figure 13.4(b)]. The arcs from the un-

![Figure 13.4 Illustrating properties of a spanning tree: (a) adding arc \((3, 9)\) to the spanning tree forms the unique cycle 3–4–5–9–3; (b) deleting arc \((4, 5)\) forms the cut \(\{5, 8\}\) with \(S = \{1, 2, 3, 4\}\).](image)
deriving graph $G$ whose two endpoints belong to the different subsets constitute a cut.

We next prove the two optimality conditions.

**Theorem 13.1 (Cut Optimality Conditions).** A spanning tree $T^*$ is a minimum spanning tree if and only if it satisfies the following cut optimality conditions: For every tree arc $(i, j) \in T^*$, $c_{ij} \leq c_{kl}$ for every arc $(k, l)$ contained in the cut formed by deleting arc $(i, j)$ from $T^*$.

**Proof.** It is easy to see that every minimum spanning tree $T^*$ must satisfy the cut optimality condition. For, if $c_{ij} > c_{kl}$ and arc $(k, l)$ is contained in the cut formed by deleting arc $(i, j)$ from $T^*$, then introducing arc $(k, l)$ into $T^*$ in place of arc $(i, j)$ would create a spanning tree with a cost less than $T^*$, contradicting the optimality of $T^*$.

We next show that if any tree $T$ satisfies the cut optimality conditions, it must be optimal. Suppose that $T^*$ is a minimum spanning tree and $T \neq T^*$. Then $T^*$ contains an arc $(i, j)$ that is not in $T$ (the reader might find it helpful to refer to Figure 13.5 while reading the rest of the proof). Deleting arc $(i, j)$ from $T^*$ creates a cut, say $\{S, \overline{S}\}$. Now notice that if we add the arc $(i, j)$ to $T^*$, we create a cycle $W$ that must contain an arc $(k, l)$ [other than arc $(i, j)$] with $k \in S$ and $l \in \overline{S}$. Since $T^*$ satisfies the cut optimality conditions, $c_{ij} \leq c_{kl}$. Moreover, since $T^*$ is an optimal spanning tree, $c_{ij} \geq c_{kl}$; otherwise we could improve on its cost by replacing arc $(k, l)$ by arc $(i, j)$. Therefore, $c_{ij} = c_{kl}$. Now if we introduce arc $(k, l)$ in the tree $T^*$ in place of arc $(i, j)$, we produce another minimum spanning tree and it has one more arc in common with $T^*$. Repeating this argument several times, we can transform $T^*$ into the minimum spanning tree $T^*$. This construction shows that $T^*$ is also a minimum spanning tree and completes the proof of the theorem.

The cut optimality conditions imply that every arc in a minimum spanning tree is a minimum cost arc across the cut that is defined by removing it from the tree. In fact, the cut optimality conditions also imply that we can always include any minimum arc cost in any cut in some minimum spanning tree, which we state in the following somewhat stronger form.

**Property 13.2.** Let $F$ be a subset of arcs in some minimum cost spanning tree and let $S$ be a set of nodes of some component of $F$. Suppose that $(i, j)$ is a minimum cost arc in the cut $\{S, \overline{S}\}$. Then some minimum spanning tree contains all the arcs in $F$ as well as the arc $(i, j)$.

**Proof.** Suppose that $F$ is a subset of the minimum cost tree $T^*$. If $(i, j) \notin T^*$, we have nothing to prove. So suppose that $(i, j) \in T^*$. Adding $(i, j)$ to $T^*$ creates a cycle $C$, and $C$ contains at least one arc $(p, q) \neq (i, j)$ in $\{S, \overline{S}\}$. By assumption, $c_{ij} \leq c_{pq}$. Also, since $T^*$ satisfies the cut optimality conditions, $c_{ij} \geq c_{pq}$. Consequently, $c_{ij} = c_{pq}$, so adding arc $(i, j)$ to $T^*$ and removing arc $(p, q)$ produces a minimum spanning tree containing all the arcs in $F$ as well as the arc $(i, j)$.

The cut optimality conditions provide us with an "external" characterization of a minimum spanning tree that rests on the relationship between a single arc in the tree and many arcs outside the tree, that is, those in the cut that we produce by removing the arc from the tree. The following related path optimality conditions provide an alternative "internal" characterization that considers the relationship between a single non-tree arc and several arcs in the tree, that is, those in the path formed by adding the non-tree arc to the spanning tree.

**Theorem 13.3 (Path Optimality Conditions).** A spanning tree $T^*$ is a minimum spanning tree if and only if it satisfies the following path optimality conditions: For every non-tree arc $(k, l)$ of $G$, $c_{kl} \leq c_{kl}$ for every arc $(i, j)$ contained in the path in $T^*$ connecting nodes $k$ and $l$.

**Proof.** It is easy to show the necessity of the path optimality conditions. Suppose $T^*$ is a minimal spanning tree satisfying these conditions and arc $(i, j)$ is contained in the path in $T^*$ connecting nodes $k$ and $l$. If $c_{ij} > c_{kl}$, introducing arc $(k, l)$ into $T^*$ in place of arc $(i, j)$ would create a spanning tree with a cost less than $T^*$, contradicting the optimality of $T^*$.

We establish the sufficiency of the path optimality conditions by using the sufficiency of the cut optimality conditions. This proof technique highlights the equivalence between these conditions. We will show that if a tree $T^*$ satisfies the path optimality conditions, it must also satisfy the cut optimality conditions; Theorem 13.1 would then imply that $T^*$ is an optimal tree. Let $(i, j)$ be any tree arc in $T^*$, and let $S$ and $\overline{S}$ be the two sets of connected nodes produced by deleting arc $(i, j)$ from $T^*$. Suppose $i \in S$ and $j \in \overline{S}$. Consider any arc $(k, l) \in \{S, \overline{S}\}$ (see Figure 13.5). Since $T^*$ contains a unique path joining nodes $k$ and $l$, and since arc $(i, j)$ is the only arc in $T^*$ joining a node in $S$ and a node in $\overline{S}$, arc $(i, j)$ must belong to this path. The path optimality condition implies that $c_{ij} = c_{kl}$; since this condition must be valid for every non-tree arc $(k, l)$ in the cut $\{S, \overline{S}\}$ formed by deleting any tree arc $(i, j)$, $T^*$ satisfies the cut optimality conditions and so it must be a minimum spanning tree.

In the preceding discussion we have established two optimality conditions for the minimum spanning tree problem. The following optimality conditions for the maximum spanning tree problem are similar. We leave their proofs as an exercise (see Exercise 13.9).

---

**Figure 13.5 Proving cut and path optimality conditions.**

---

**Minimum Spanning Trees** Chap. 13

**Sec. 13.3 Optimality Conditions**

Page 519
Theorem 13.4 (Maximum Spanning Tree Optimality Conditions).
(a) A spanning tree $T^*$ is a maximum spanning tree if and only if it satisfies the following cut optimality conditions: For every tree arc $(i, j) \in T^*$, $c_{ij} \leq c_{kl}$ for every arc $(k, l)$ contained in the cut formed by deleting arc $(i, j)$ from $T^*$.
(b) A spanning tree is a maximum spanning tree $T^*$ if and only if it satisfies the following path optimality conditions: For every non-tree arc $(k, l)$ of $G$, $c_{ij} \geq c_{kl}$ for every arc $(i, j)$ contained in the tree path in $T^*$ connecting nodes $k$ and $l$.

13.4 KRUSKAL'S ALGORITHM

The path optimality conditions immediately suggest the following straightforward algorithm for solving the minimum spanning tree problem. We start with any arbitrary spanning tree $T$ and test the path optimality conditions. If $T$ satisfies this condition, it is an optimal tree; otherwise, $c_{ij} > c_{kl}$ for some non-tree arc $(k, l)$ and some tree arc $(i, j)$ contained in the unique path in $T$ connecting nodes $k$ and $l$. In this case, adding arc $(k, l)$ to $T$ in place of arc $(i, j)$ gives us a spanning tree with a lower cost. Repeating this step will give us a minimum spanning tree within a finite number of iterations. Although this algorithm is strikingly simple, its running time cannot be polynomially bounded in the size of the problem data.

Simple Version of Kruskal's Algorithm

To derive an alternative and more efficient algorithm, known as Kruskal's algorithm, from the path optimality conditions, we consider an algorithm that builds an optimal spanning tree from scratch by adding one arc at a time. We first sort all the arcs in non-decreasing order of their costs and define a set, LIST, that is the set of arcs we have chosen as part of a minimum spanning tree. Initially, the list LIST is empty. We examine the arcs in the sorted order one by one and check whether adding the arc we are currently examining to LIST creates a cycle with the arcs already in LIST. If it does not, we add the arc to LIST; otherwise, we discard it. We terminate when $|\text{LIST}| = n - 1$. At termination, the arcs in LIST constitute a minimum spanning tree $T^*$.

The correctness of Kruskal's algorithm follows from the fact that we discarded each non-tree arc $(k, l)$ with respect to $T^*$ at some stage because it created a cycle with the arcs already in LIST. But observe that the cost of arc $(k, l)$ is greater than or equal to the cost of every arc in that cycle because we examined the arcs in the non-decreasing order of their costs. Therefore, the spanning tree $T^*$ satisfies the path optimality conditions, so it is an optimal tree.

To illustrate Kruskal's algorithm on a numerical example, we consider the network shown in Figure 13.6(a). Sorted in the order of their costs, the arcs are $(2, 4), (3, 5), (2, 3), (4, 5), (2, 1)$, and $(3, 1)$. In the first three iterations, the algorithm adds the arcs $(2, 4), (3, 5)$, and $(4, 5)$ to LIST (see Figures 13.6(b) to (d)). In the next two iterations, the algorithm examines arcs $(2, 3)$ and $(4, 5)$ and discards them because the addition of each arc to LIST creates a cycle [see Figure 13.6(e) and (f)]. Then the algorithm adds arc $(2, 1)$ to LIST and terminates. Figure 13.6(g) shows the minimum spanning tree.

We might view the running time of Kruskal's algorithm as being composed of the time for sorting the arcs and the time for detecting cycles. For a network with arbitrarily large arc costs, sorting requires $O(m \log m) = O(m \log n^2) = O(m \log n)$ time. The time to detect a cycle depends on the method we use for this step. One naive method would work as follows. The set LIST at any stage of the algorithm is a forest (i.e., a collection of subtrees). For example, the set LIST corresponding to Figure 13.6(c) consists of three trees containing the nodes $(1), (2, 4)$, and $(3, 5)$, respectively. We denote these sets of nodes for a collection of trees by $N_1, N_2, N_3, \ldots$. We can store these sets as different singly linked lists. While examining an arc $(k, l)$, we scan through these linked lists and check whether both the nodes $k$ and $l$ belong to the same list. If so, adding arc $(k, l)$ to LIST creates a cycle and we discard this arc. If nodes $k$ and $l$ belong to different lists, we add arc $(k, l)$ to LIST, which requires merging the lists containing nodes $k$ and $l$ into a single list. Clearly, this data structure requires $O(n)$ time for each arc that we examine, so if we use this data structure, Kruskal's algorithm runs in $O(mn)$ time.

Improved Implementation of Kruskal's Algorithm

We now describe a more efficient implementation of Kruskal's algorithm that runs in $O(m + n \log n)$ time plus the time taken for sorting the arcs. This implementation is similar to the preceding one: We store the collection of trees, denoted by the sets.
implementation has a running time of $O(m \alpha(n, m))$ for a function $\alpha(n, m)$ that grows so slowly that for all practical purposes it can be viewed as a constant less than 6 (see the reference notes).

13.5 PRIM’S ALGORITHM

Just as the path optimality conditions allowed us to develop Kruskal’s algorithm, the cut optimality conditions permit us to develop another simple algorithm for the minimum spanning tree problem, known as Prim’s algorithm. This algorithm builds a spanning tree from scratch by fanning out from a single node and adding arcs one at a time. It maintains a tree spanning on a subset $S$ of nodes and adds a nearest neighbor to $S$. The algorithm does so by identifying an arc $(i, j)$ of minimum cost in the cut $[S, \bar{S}]$. It adds arc $(i, j)$ to the tree, node $j$ to $S$, and repeats this basic step until $S = N$. The correctness of the algorithm follows directly from Property 13.2 since this result implies that each arc that we add to the tree is contained in some minimum spanning tree with the arcs that we have selected in the previous steps.

We illustrate Prim’s algorithm on the same example, shown in Figure 13.7(a), that we used earlier to illustrate Kruskal’s algorithm. Suppose, initially, that $S = \{1\}$. The cut $[S, \bar{S}]$ contains two arcs, $(1, 2)$ and $(1, 3)$, and the algorithm selects the arc $(1, 2)$ [see Figure 13.7(b)]. At this point $S = \{1, 2\}$ and the cut $[S, \bar{S}]$ contains the arcs $(1, 3), (2, 3)$, and $(2, 4)$. The algorithm selects arc $(2, 4)$ since it has the minimum cost among these three arcs [see Figure 13.7(c)]. In the next two iterations, the algorithm adds arc $(4, 3)$ and then arc $(3, 5)$; Figure 13.7(d) and (e) show the details of these iterations. Figure 13.7(f) shows the minimum spanning tree produced by the algorithm.

To analyze the running time of Prim’s algorithm, we consider each of the $n − 1$ iterations that the algorithm performs as it adds one arc at a time to the tree until it has a spanning tree with $n − 1$ arcs. In each iteration, the algorithm selects
the minimum cost arc in the cut \( \{ S, \bar{S} \} \). If we scan the entire arc list to identify the minimum cost arc, this operation requires \( O(m) \) time, giving us an \( O(nm) \) time bound for the algorithm. Therefore, this implementation of Prim’s algorithm runs in \( O(nm) \) time. However, we can improve upon it substantially, as we now show.

The bottleneck step in the \( O(nm) \) implementation of Prim’s algorithm is the identification of a minimum cost arc in the cut \( \{ S, \bar{S} \} \). We can improve the efficiency of this step by maintaining two indices for each node \( j \) in \( \bar{S} \): (1) a distance label \( d(j) \), which represents the minimum cost of arcs in the cut incident to a node \( j \) not in \( S \) (i.e., \( d(j) = \min \{c_{ij} : i \in \{ S, \bar{S} \} \} \)), and (2) a predecessor label \( \text{pred}(j) \), which represents the other endpoint of the minimum cost arc in the cut incident to node \( j \). For example, in Figure 13.7(d), three arcs, \( (1, 3), (2, 3), \) and \( (4, 3) \), in the cut are incident to node 3. Among these arcs, arc \( (4, 3) \) has the minimum cost of 20. Therefore, \( d(3) = 20 \) and \( \text{pred}(3) = 4 \). For the same figure, \( d(5) = 30 \) and \( \text{pred}(5) = 4 \). If we maintain these indices, we can easily find the minimum cost of an arc in the cut; we simply compute \( \min \{d(j) : j \in \bar{S} \} \). If node \( i \) achieves this minimum, \( \text{pred}(i) \) is a minimum cost arc in the cut. Observe that if we move node \( i \) from \( \bar{S} \) to \( S \), we need to update the distance and predecessor labels only for the nodes adjacent to node \( i \).

Notice the similarity between this implementation of Prim’s algorithm and the implementation of Dijkstra’s algorithm that we discussed in Section 4.5. Just as in Dijkstra’s algorithm, the basic operations are finding the minimum distance label \( d(i) \) among the nodes in the set \( \bar{S} \), moving the corresponding node into the set \( S \), and updating the distance labels of those nodes in \( \bar{S} \) that are adjacent to node \( i \). Indeed, we can implement Prim’s algorithm using the various types of heaps (or priority queues) that we used in our implementations of Dijkstra’s algorithm in Section 4.7. Recall from Appendix A that a heap is a data structure that permits us to perform the following operations on a collection \( H \) of objects: each having an associated number called its key.

\begin{itemize}
  \item \text{create-heap}(H).
  \item \text{find-min}(H).
  \item \text{insert}(i, H).
  \item \text{delete-min}(H).
  \item \text{decrease-key}(i, \text{value}, H).
\end{itemize}

Observe that if we implement Prim’s algorithm using a heap, \( H \) would be the collection of nodes in \( \bar{S} \) and the key of a node would be its distance label. Prim’s algorithm would be implemented as described in Figure 13.8. As always, we let \( C \) denote the maximum arc cost in the graph \( G \).

As is clear from its description, Prim’s algorithm performs the operations \text{find-min}, \text{delete-min}, and \text{insert} at most \( n \) times and the operation \text{decrease-key} at most \( m \) times. When implemented with different heaps, the algorithm would have the running times shown in Figure 13.9.

Our discussion of the binary heap, \( d \)-heap, and Fibonacci heap data structures in Appendix A permits us to justify these time bounds. In that discussion we showed that the \( d \)-heap data structure performs each delete-min operation in \( O(d \log d) \) time and every other heap operation in \( O(\log d) \) time. If we select \( d = \min \), this result gives us a running time of \( O(m \log \Delta + nd \log n) = O(m \log n) \) for Prim’s algorithm implemented using the \( d \)-heap data structure. The binary heap is a special case of \( d \)-heap with \( d = 2 \), so its time bound is \( O(m \log n) \). The Fibonacci heap data structure performs each delete-min operation in \( O(\log n) \) time and every other heap operation in \( O(1) \) time. Consequently, the Fibonacci heap implementation of Prim’s algorithm runs in \( O(m + n \log n) \) time.

**Theorem 13.6.** Fibonacci heap implementation of Prim’s algorithm solves the minimum spanning tree problem in \( O(m + n \log n) \) time.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Heap type} & \textbf{Running time} \\
\hline
Binary heap & \( O(m \log n) \) \\
\hline
\( d \)-heap & \( O(m \log_{d+1} n) \) \quad \text{with} \quad d = \max(2, \min) \\
\hline
Fibonacci heap & \( O(m + n \log n) \) \\
\hline
Johnson’s data structure & \( O(m \log \log C) \) \\
\hline
\end{tabular}
\caption{Running time of various heap implementations of Prim’s algorithm.}
\end{table}

**Figure 13.9** Running time of various heap implementations of Prim’s algorithm.

Sec. 13.5 Prim’s Algorithm
10.6 SOLLIN’S ALGORITHM

We can use the cut optimality conditions to derive another novel algorithm for the minimum spanning tree problem, known as Sollin’s algorithm. We can view this algorithm as a hybrid version of Kruskal’s and Prim’s algorithm. As in Kruskal’s algorithm, Sollin’s algorithm maintains a collection of trees spanning the nodes $N_1$, $N_2$, $N_3$, . . . and adds arcs to this collection. However, at every iteration, it adds a minimum cost arc emanating from these trees, an idea borrowed from Prim’s algorithm. As a result, we obtain a fairly simple algorithm that uses elementary data structures and runs in $O(m \log n)$ time. As pointed out in the reference notes, a more clever implementation of this approach runs in $O(m \log \log n)$ time.

Sollin’s algorithm repeatedly performs the following two basic operations:

- **nearest-neighbor** ($N_v$, $i_k$, $j_k$). This operation takes as an input a tree spanning the nodes $N_v$ and determines an arc $(i_k, j_k)$ with the minimum cost among all arcs emanating from $N_v$ [i.e., $C_{ij} = \min(C_{ij} : (i, j) \in A, i \in N_v$ and $j \in N_v$)].

To perform this operation we need to scan all the arcs in the adjacency lists of nodes in $N_v$, and find a minimum cost arc among those arcs that have one endpoint not belonging to $N_v$.

- **merge**($i_k$, $j_k$). This operation takes as an input two nodes $i_k$ and $j_k$, and if the two nodes belong to two different trees, then merges these two trees into a single tree.

Using these two basic operations, we state Sollin’s algorithm as shown in Figure 13.10.

We illustrate Sollin’s algorithm on the same numerical example that we have used to illustrate Kruskal’s and Prim’s algorithms. As shown in Figure 13.11(b), Sollin’s algorithm starts with a forest containing five trees: Each tree is a singleton node. This figure also shows the least cost arc emanating from each tree. We next perform mergings, reducing the number of trees to only two [see Figure 13.11(c)]. The least cost arc emanating from these two trees is (3, 4), and when we add this arc, we obtain the spanning tree shown in Figure 13.11(d). The algorithm now terminates.

To analyze the running time of Sollin’s algorithm, we need to discuss the data structure needed to implement it. We will show that the algorithm performs $O(n \log n)$ executions of the while loop, and that we can perform all the nearest-neighbor and merge operations in $O(m)$ time. These results establish a time bound of $O(m \log n)$ for Sollin’s algorithm.

We store the nodes of a tree as a circular doubly linked list. The doubly linked list allows us to visit every node of the tree starting at any tree node. We assign a numerical label with every node in the network; the label satisfies the following two properties: (1) nodes of the same tree have the same label, and (2) nodes of different trees have different labels. At the beginning of the algorithm, we assign label $i$ to each node $i \in N$.

Using this data structure, we can easily check whether an arc $(i, j)$ has both of its endpoints in the same tree. We answer this question simply by checking the labels of nodes $i$ and $j$. This observation implies that we can perform the nearest-neighbor operation for each tree in the forest in a total time of $O(\sum_{i \in N} |A(i)|) = O(m)$.

We perform merge operations in a while loop using the following iterative scheme. In each iteration we select an unexamined tree, say $N_i$, and consider the minimum cost arc $(i_1, j_1)$ emanating from $N_i$. (Node $i_1$ is in $N_i$ and $j_1$ might or might not be in $N_i$.) Suppose that nodes in $N_i$ have the label $\alpha$. If node $j_1$ also has the label $\alpha$, the iteration ends. Otherwise, we scan through the nodes of the tree, say $N_{\alpha}$, containing node $j_1$ and assign them the label $\alpha$. Next, we consider the minimum cost arc $(i_2, j_2)$ emanating from $N_{\alpha}$. If node $j_2$ has label $\alpha$, the iteration ends; otherwise, we scan through the nodes of the tree, say $N_{\beta}$, containing node $j_2$ and assign them the label $\alpha$. We repeat this process until the iteration ends. Notice that within an iteration we might assign the nodes of several trees the label of the first tree. When an iteration ends, we initiate a new iteration by selecting another unexamined tree. We terminate this iterative process when we have examined all the trees. As is clear from this description, this method assigns a label to each node once and hence runs in $O(n)$ time.

Having proved that each execution of the while loop in Sollin’s algorithm re-
quires $O(m)$ time, we now obtain a bound on the number of executions of the loop. Each execution of the loop reduces the number of trees in the forest by a factor of at least two because we merge each tree into a larger tree. This observation implies that we will perform $O(\log n)$ executions of the loop. We have therefore established the following result.

**Theorem 13.7.** The execution of Sollin’s algorithm requires $O(m \log n)$ time.

### 13.7 Minimum Spanning Trees and Matroids

In keeping with the orientation of this book, we have examined the minimum spanning tree problem from a perspective of graph theory and the data structures needed to implement spanning tree algorithms efficiently. We could, instead, view the minimum spanning tree problem and develop several of the core ideas of this chapter from at least two other perspectives: (1) broader notions in combinatorial optimization, and (2) linear programming. These two alternative viewpoints are instructive because they help show the connection between network optimization and other important topics in discrete optimization. Indeed, the minimum spanning tree problem and network flows have inspired the development of many other problem domains in discrete optimization. Consequently, it is useful to pause at this point and briefly delineate these connections.

**Matroids and the Greedy Algorithm**

Suppose that we view a spanning tree in the following way: We have a finite collection of objects $E$, the arcs of a network, and we define a subset $I$ of objects to be independent if they do not form a cycle in the network. If each object (i.e., arc) $e$ has an associated weight $w_e$, the minimal spanning tree problem seeks an $n - 1$ element independent set $I$ with the smallest total weight $w(I) = \sum_{e \in I} w_e$.

Let us now describe this and related problems in a more abstract setting. A subset system $(E, S)$ is a finite set of objects $E$ and a nonempty collection $\mathcal{S}$ of subsets of the object set $E$, called independent sets, that satisfies the hereditary property that whenever $I$ is an independent set (i.e., belongs to $\mathcal{S}$) and $I'$ is a subset of $I$, then $I'$ also is an independent set.

Suppose that we associate a weight $w_e$ with each element $e$ of $E$ and define the weight $w(S)$ of any subset $S$ of $E$ as the sum of the weights of its elements; that is, $w(S) = \sum_{e \in S} w_e$. As a generalization of the minimum spanning tree problem, we might consider the following independence (or subset) system optimization problem: Find a maximal independent set of the subset system $(E, \mathcal{S})$ with the minimum weight.

At this level of generality, the independence system optimization problem appears to be hopelessly difficult to solve efficiently. Therefore, we need to impose additional structure on the problem so that it becomes tractable. We would like to do so, however, by imposing the least amount of additional structure so that the results remain as general as possible. The following type of auxiliary structure appears to be just right for this purpose.

A subset system $(E, \mathcal{S})$ is a **matroid** if it satisfies the property that if $I_p$ and $I_{p+1}$ are independent sets containing $p$ and $p + 1$ elements, we always can find an element $e \in I_{p+1} \setminus I_p$, satisfying the property that $I_p \cup \{e\}$ is an independent set.

Note that this definition implies if $I$ and $I'$ are any two independent sets and $|I| > |I'|$, we can add certain elements of $I'$ to $I$ and obtain another independent set $I''$ so that $I''$ contains $I$ and has as many elements as $I'$. That is, the sets $I_p$ and $I_{p+1}$ in the definition need not differ in cardinality by one. (We establish this extended growth property by applying the growth property to $I$ and the first $|I| + 1$ elements of $I'$, which are independent by the definition, and then repeating the operation.)

Let us illustrate the definition of a matroid with a few examples.

**Graph or forest matroid.** Note that forests in a network satisfy these definitions if we let $E$ equal the arcs in a network and let $\mathcal{S}$ denote the collection of arc sets that contain no cycles (i.e., the arcs define a forest). In this case the system $(E, \mathcal{S})$ is an independent system because removing arcs from a forest always produces another forest. Moreover, if $I_p$ and $I_{p+1}$ are two forests containing $p$ and $p + 1$ arcs, the forest $I_{p+1}$ must contain an arc $e$ that we can add to $I_p$ and produce another forest $I_p \cup \{e\}$ (see Exercise 13.41). Consequently, the system $(E, \mathcal{S})$ is a matroid.

**Partition matroid.** Let $E = E_1 \cup E_2 \cup \cdots \cup E_k$ be a union of $K$ disjoint finite sets and let $u_1, u_2, \ldots, u_k$ be given positive integers. Let $\mathcal{S}$ be the family of subsets $I$ of $E$ that satisfy the property that for all $k = 1, 2, \ldots, K$, $I$ contains no more than $u_k$ elements of $E_k$. The system is a matroid. Note that if we consider a bipartite graph $(N_1 \cup N_2, A)$ and let $E_k$ for all nodes in $N_k$, be the set of arcs incident to node $k$, then if all the $u_k$ are equal to 1, the matroid defines “half” of an assignment problem. Another partition matroid defined on the nodes $N_2$ defines the other half of the matroid, so any feasible solution to the assignment problem is an independent set in both partition matroids.

**Matroid.** Let $M$ be a real-valued matrix, let $E$ be the columns of $M$, and let $\mathcal{S}$ be sets of columns of $M$ that are linearly independent. Since removing columns from a linearly independent set of columns produces another independent set, the system $(E, \mathcal{S})$ is a subset system. By elementary results in linear algebra, this system also satisfies the growth property and so is a matroid.

Let us make one further observation about matroids. A maximal independent set is an independent set $I$ satisfying the property that we cannot add any other element $e$ to $I$ and produce another independent set. The (extended) growth property implies that every maximal independent set of a matroid contains the same number of elements (since we can always add elements of one maximal independent set to another if they contain a different number of elements). Borrowing notation from linear algebra, we refer to any maximal independent set as a basis of the matroid. In this terminology, the matroid optimization problem seeks a basis with the smallest possible total weight.

We can attempt to solve this problem by using a greedy algorithm (Figure 13.12) which is a direct generalization of Kruskal’s algorithm.

Note that for the minimal spanning tree problem, the test condition “$\text{LIST} \cup \{e\}$ is independent” from this algorithm is just the test condition from Kruskal’s
algorithm greedy; begin
    order the elements of \( E = \{e_1, e_2, \ldots, e_k\} \) so that \( w_1 \leq w_2 \leq \cdots \leq w_k \);
    set \( \text{LIST} = \emptyset \);
    for \( j = 1 \) to \( k \) do
        if \( \text{LIST} \cup \{e_j\} \) is independent then \( \text{LIST} = \text{LIST} \cup \{e_j\} \);
    end; \( \text{LIST} \) is a minimum weight basis;

Figure 13.12 Greedy algorithm.

algorithm, namely, that the network defined by the arcs in \( \text{LIST} \) and \( e_j \) contains no cycle.

**Theorem 13.8. The greedy algorithm solves the matroid optimization problem.**

Proof. Let \( I^* \) be any optimal solution to the matroid optimization problem and let \( \text{LIST} = \{e_1, e_2, \ldots, e_h\} \) be the solution generated by the greedy algorithm. We will show that \( w(\text{LIST}) = w(I^*) \) and therefore that \( \text{LIST} \) is an optimal basis as well. If \( \text{LIST} = I^* \), we have nothing to prove. So assume that \( \text{LIST} \neq I^* \). Suppose that we order the elements of \( I^* \) in the order of increasing indices from the set \( E = \{e_1, e_2, \ldots, e_k\} \) as \( e_{i_1}, e_{i_2}, \ldots, e_{i_h} \), and assume that \( e_1 \) is the first element of \( I^* \) not in \( \text{LIST} \). Since the set \( \{e_1, e_2, \ldots, e_h\} \) is independent, the steps of the greedy algorithm imply that \( j \geq n+1 \) and therefore that \( w_{j+1} \leq w_j \). Now, since both the sets \( I = \{e_1, e_2, \ldots, e_h, e_{j+1}\} \) and \( I^* \) are independent, the growth property implies that we can add elements to \( I \) to obtain another basis \( I' \). Since this basis contains the elements \( I^* \cup \{e_{j+1}\} = I^* \cup \{e_{j+1}\} \) for some \( e_{j+1} \in I^* \) and \( p \geq j+1 \), \( w(I') = w(I^*) \); consequently, \( I' \) is also an optimal basis. Note that this basis has a greater cardinality of elements in common with \( \text{LIST} \) (at least \( k+1 \)). But now if we apply the same argument to the sets \( I' \) and \( \text{LIST} \), we will obtain another optimal basis with at least one more lead element in common with \( \text{LIST} \). If we continue in this fashion, eventually \( I' \) will equal \( \text{LIST} \), therefore establishing that \( \text{LIST} \) is a minimum-basis matroid.

Note that this discussion not only gives an alternative proof of Kruskal’s algorithm for the minimum spanning tree problem, but also shows that two underlying combinatorial properties—indepenence and the growth property—are the essential ingredients necessary to ensure that the greedy algorithm solves the minimum spanning tree problem. (In Exercise 13.45 we show that the greedy algorithm will solve the minimum weight independent set problem for any choice of the element weights if and only if the subset system is a matroid.) Therefore, any other property of a graph is irrelevant for ensuring that Kruskal’s algorithm works correctly. That is, we have now identified the combinatorial postulates that drive the algorithm.

### 13.8 MINIMUM SPANNING TREES AND LINEAR PROGRAMMING

Linear programming provides yet another proof of Kruskal’s algorithm. Moreover, the development of a linear programming-based approach permits us to make some elementary connections between network optimization and an important topic in applied mathematics, polyhedral combinatorics, which is the study of integer polyhedra (i.e., polyhedra with integer extreme points). As shown in Section 11.12, the minimum cost flow problem provides another connection between these topics.

Let \( A(S) \) denote the set of arcs contained in the subgraph of \( G = (N, A) \) induced by the node set \( S \) (i.e., \( A(S) \) is the set of arcs of \( A \) with both endpoints in \( S \)). Consider the following integer programming formulation of the minimum spanning tree problem:

\[
\text{Minimize} \quad \sum_{(i,j) \in E} c_{ij}x_{ij} \quad \text{(13.2a)}
\]

subject to
\[
\sum_{(i,j) \in A(s)} x_{ij} = n - 1, \quad \text{(13.2b)}
\]
\[
\sum_{(i,j) \in A(s)} x_{ij} \leq |S| - 1 \quad \text{for any set of nodes,} \quad \text{(13.2c)}
\]
\[
x_{ij} \geq 0 \text{ and integer.} \quad \text{(13.2d)}
\]

In this formulation, the \( 0 \)–\( 1 \) variable \( x_{ij} \) indicates whether we select arc \((i,j)\) as part of the chosen spanning tree (note that the second set of constraints with \(|S| = 2\) implies that each \(x_{ij} \in \{0,1\}\). The constraint (13.2b) is a cardinality constraint implying that we choose exactly \( n - 1 \) arcs, and the "packing" constraint (13.2c) implies that the set of chosen arcs contains no cycles (if the chosen solution contained a cycle, and \( S \) were the set of nodes on a chosen cycle, the solution would violate this constraint). Note that as a function of the number of nodes in the network, this model contains an exponential number of constraints. Nevertheless, as we will show, we can solve it very efficiently by applying Kruskal’s algorithm. We might note that any formulation in the variables \( x_{ij} \) always requires an exponential number of constraints; that is, we cannot replace the given constraints by some polynomial set of constraints and still have a valid formulation of the problem. Nevertheless, it is possible to give a polynomial formulation of the problem if we introduce new (multicommodity flow) variables (see the reference notes).

Suppose that we consider the linear programming relaxation of this integer programming model. That is, we drop the restriction that the variables be integer. As we noted in Section 9.4 (also see Appendix C), we can formulate a set of reduced cost and complementary slackness optimality conditions for every linear programming problem and use these conditions, as we use the reduced costs and complementary slackness conditions of network flows, to assess when a given feasible solution is optimal. Recall that for network flow problems, we used node potentials to define the reduced costs and the complementary slackness conditions; each node in a minimum cost flow problem corresponds to one equation or inequality for any set \( S \) of nodes (the one equation in the model corresponds to the node set \( S = N \)), and for the minimum spanning tree problem we associate a potential \( \mu_S \) with every set \( S \) of nodes. The potential \( \mu_S \) is unrestricted in sign and the other potentials \( \mu_S \) must be nonnegative. We then define the reduced cost \( c_{ij}' = c_{ij} + \sum_{(i,j) \in A(s)} \mu_S \).
With this definition of the reduced costs, we have the following complementary slackness optimality conditions.

**Minimum spanning tree complementary slackness optimality conditions.** A solution $x$ of the minimum spanning tree problem is an optimal solution to the linear programming relaxation of the integer programming formulation (13.2) if and only if we can find node potentials $\mu_S$ defined on node sets $S$ so that the reduced costs satisfy the following conditions:

$$c^T_f = 0 \quad \text{if} \quad x_f > 0,$$

$$c^T_f \geq 0 \quad \text{if} \quad x_f = 0.$$

We can use this fundamental result to give yet another proof that Kruskal’s algorithm solves the minimum spanning tree problem.

**Theorem 13.9.** If $x$ is the solution generated by Kruskal’s algorithm, $x$ solves both the integer program (13.2) and its linear programming relaxation.

Rather than giving a formal proof of this theorem, let us illustrate the proof technique on the five-node example that we have already considered in Figure 13.6. For this problem, Kruskal’s algorithm chooses the arcs of the minimum spanning tree in the order $(2, 4), (3, 5), (3, 4), (1, 2)$. So we set $x_{24} = x_{35} = x_{34} = x_{12} = 1$ and $x_{13} = x_{23} = x_{45} = 0$. Note that during the course of applying Kruskal’s algorithm, we form several connected node components: first $(2, 4)$, then $(3, 5)$, then $(2, 3, 4, 5)$, and finally, the entire node set $(1, 2, 3, 4, 5)$. We will associate a nonzero potential with these sets and a zero potential with every other set of nodes. We define these potentials in the reverse order that Kruskal’s algorithm formed the node components. We first set $\mu_{\{1,2,3,4,5\}} = -35$, the negative of the cost of the final arc added to the tree. Now we note that each arc $(i, j)$ that we add to the tree defines a node component $S(i, j)$. For example, $S(3, 4) = \{2, 3, 4, 5\}$. Moreover, at some later stage in the algorithm, we combine the node component $S(i, j)$ with one or more other nodes to define a large component by adding another arc $(p, q)$ to the tree. We now set the potential of the node component $S(i, j)$ to be the difference between the cost of arc $(p, q)$ and the cost of arc $(i, j)$. Therefore, we set $\mu_{\{2,3,4,5\}} = c_{12} - c_{34} = 35 - 20 = 15$, $\mu_{\{2,3\}} = c_{34} - c_{35} = 20 - 15 = 5$, and $\mu_{\{2,4\}} = c_{34} - c_{24} = 20 - 10 = 10$.

Now checking the reduced cost of every arc, we find that

$$c^T_f = 35 - 35 = 0,$$

$$c^T_f = 40 - 35 = 5,$$

$$c^T_f = 25 - 35 + 15 = 5,$$

$$c^T_f = 10 - 35 + 15 + 10 = 0,$$

$$c^T_f = 20 - 35 + 15 = 0,$$

$$c^T_f = 15 - 35 + 15 + 5 = 0,$$

$$c^T_f = 30 - 35 + 15 = 10.$$

Note that with these choices of the potentials, the reduced cost of every arc chosen by Kruskal’s algorithm is zero and the cost of every other arc $(i, j)$ is the difference between the cost of arc $(i, j)$ and the cost of the most expensive arc on the path formed by adding arc $(i, j)$ to the tree found by Kruskal’s algorithm. It is fairly easy to use an induction argument to extend this proof technique for any problem and thus to give a formal proof of Theorem 13.9 (see Exercise 13.42).

The proof technique we have just illustrated establishes one of the most important core results in combinatorial optimization. Since a linear program always has an extreme point solution (see Appendix C for this result and for linear programming definitions), if we can show that for every choice of the coefficients of its objective function, a linear programming formulation has at least one integer solution, then the extreme points of the polyhedron defined by that linear program are integer valued. Since we have just established this property for the linear programming relaxation of the integer program (13.2), we have proven the following fundamental result.

**Theorem 13.10.** The polyhedron defined by the linear programming relaxation of the packing formulation of the minimum spanning tree problem has integer extreme points.

This theorem is just one example of an important meta rule that seems to lie at the core of combinatorial optimization; namely, for essentially most optimization problems that can be solved in polynomial time, it is possible to define a linear program with integer extreme points that contains the incident vectors of the solution to the combinatorial optimization problem. The minimum cost flow problem and the minimal spanning tree problem were two of the first notable examples of this result discovered in the combinatorial optimization literature; these results have inspired many streams of investigation within discrete optimization, such as the study of matroids that we introduced in Section 13.7. For example, it is possible to specify a linear programming formulation of the matroid optimization problem so that the extreme points of the linear programming formulation are exactly the set of bases of the underlying matroid (see Exercise 13.44).

### 13.9 Summary

The minimum spanning tree problem is perhaps the simplest, and certainly one of the most central, models in the field of combinatorial optimization. In this chapter, after describing several applications of minimum spanning trees, we proved two (equivalent) necessary and sufficient conditions—the cut and path optimality conditions—for characterizing the optimality of minimum spanning trees. The cut optimality conditions state that a spanning tree $T^*$ is a minimum spanning tree if and only if the cost of the tree $T^*$ is less than or equal to the cost of every non-tree arc in the cut formed by deleting arc $(i, j)$ from $T^*$. The path optimality conditions are closely related to these conditions (in a sense, they are dual conditions); they state that a spanning tree $T^*$ is a minimum spanning tree if and only if the cost of every non-tree arc $(k, l)$ is greater than or equal to the cost of every tree arc in the path in $T^*$ between nodes $k$ and $l$.
In this chapter we described three algorithms for solving the minimum spanning tree problem: Kruskal’s, Prim’s, and Sollin’s. All these algorithms are easy to implement, have excellent running times, and are very efficient in practice. Figure 13.13 summarizes the basic features of these algorithms.

The minimum spanning tree problem is important not only because it is a core model in network optimization, but also because it serves as a valuable prototype model in combinatorial optimization that has stimulated many lines of inquiry. In this chapter we have considered two ways in which minimum spanning trees relate to general issues in combinatorial optimization. If we consider Kruskal’s algorithm as a greedy procedure that chooses the minimum cost feasible arc at each step, we might ask whether a similar type of greedy algorithm is able to solve other combinatorial optimization problems. We have answered this question affirmatively by showing that the greedy algorithm also solves a broad class of problems known as matroid optimization problems.

Studying specialized structures, such as matroids, is one very important stream of inquiry in combinatorial optimization. Another is the use of linear programming as a tool for understanding and solving combinatorial optimization problems. In Section 13.8 we showed how to characterize the incidence vectors of spanning trees as solutions to a linear programming formulation of the problem; we also showed how to interpret Kruskal’s algorithm as a method for solving this linear program. This development illustrates the use of linear programming in combinatorial optimization and is indicative of the type of investigations that analysts conduct in the important subspecialty of combinatorial optimization known as polyhedral combinatorics (i.e., the study of integer polyhedra).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running time</th>
<th>Features</th>
</tr>
</thead>
</table>
| Kruskal’s       | $O(m + n \log n)$ plus time needed to sort m arc lengths | 1. Examines arcs in nondecreasing order of their lengths and include them in the minimum spanning tree if the added arc does not form a cycle with the arcs already chosen.  
2. The proof of the algorithm uses the path optimality conditions.  
3. Attractive algorithm if the arcs are already sorted in increasing order of their lengths. |
| Prim’s          | $O(m + n \log n)$ | 1. Maintains a tree spanning a subset S of nodes and adds a minimum cost arc in the cut $\{S, \bar{S}\}$.  
2. The proof of the algorithm uses the cut optimality conditions.  
3. Can be implemented using a variety of heap structures; the stated time bound is for the Fibonacci heap data structure. |
| Sollin’s        | $O(m \log n)$ | 1. Maintains a collection of node-disjoint trees; in each iteration, adds the minimum cost arc emanating from each such tree.  
2. The proof of the algorithm uses the cut optimality conditions. |

Reference Notes

Algorithms for the minimum spanning tree problem, developed as early as 1926, are among the earliest network algorithms. The paper by Graham and Hell [1985] presents an excellent survey of the historical developments of minimum spanning tree algorithms. Borůvka [1926] and Jarník [1930] independently formulated and solved the minimum spanning tree problem. Later, other researchers rediscovered these algorithms. Kruskal [1956] and Loberman and Weinberger [1957] independently discovered Kruskal’s algorithm discussed in Section 13.4. Prim [1957] developed the algorithm described in Section 13.5. Sollin presented his algorithm, discussed in Section 13.6, in a seminar in 1961; it was never published. Claude Berge was present at this seminar and reported this algorithm in his book, Berge and Ghoulia-Houri [1962]. Later, researchers discovered that Sollin’s algorithm is similar to Borůvka’s algorithm and that Prim’s algorithm is similar to Jarník’s algorithm.

Our description of Kruskal’s algorithm runs in the time required to sort m numbers plus $O(m + n \log n)$. The use of improved union-find data structures leads to a faster implementation of Kruskal’s algorithm. This implementation, as developed by Tarjan [1984], runs in the time required to sort m numbers plus $O(m \log(n, m); \alpha(n, m)$ is the Ackermann function which, for all practical purposes, is smaller than 6. In this chapter we reported an $O(m + n \log n)$ implementation of Prim’s algorithm; this implementation appears to be new. Gabow, Galil, Spencer, and Tarjan [1986] presented a variant of this algorithm that runs in $O(m \log \beta(m, n))$ time with the function $\beta(m, n)$ defined as $\beta(m, n) = \min\{\log^* \log(m/n) < 1\}$. In this expression, $\log^x = \log \log \log \cdots \log x$ with the log iterated i times. So $\beta(m, n)$ is a very slowly growing function. For example, if $\min = 2^{\log \log \log \log \cdots \log n}$ then $\beta(m, n) = 6$. Yao [1975] developed an improved implementation of Sollin’s algorithm running in $O(m \log \log n)$ time. Currently, the fastest algorithm for solving the minimum spanning tree algorithm is Tarjan’s [1984] implementation of Kruskal’s algorithm if the arcs are already sorted, and Gabow et al. [1986] variant of Prim’s algorithm, otherwise. Gabow et al. [1986] also give efficient algorithms that (1) solve the minimum spanning tree problem in a directed network (i.e., the arborescence problem), and (2) solve the minimum spanning tree problem with a single degree constraint.

Chin and Houch [1978], Davish and Srikant [1979], and Tarjan [1982] have developed techniques for minimizing the minimum spanning tree problem when we change arc costs. Haymond, Jarvis, and Shier [1980] have described data structures for implementing Kruskal’s, Prim’s, and Sollin’s algorithm and have presented computational results for these algorithms. Jarvis and White [1983] described the results of another computational study. These studies indicate that Prim’s and Sollin’s algorithms are consistently superior to Kruskal’s algorithm. They show that Sollin’s algorithm is better than Prim’s algorithm for sparse networks, and is worse for dense networks. These studies find that the best implementation of Prim’s algorithm uses a variant of Dijkstra’s implementation of Djikstra’s algorithm that we described in Section 4.6.

Our presentation of matroids in Section 13.7 and of a linear programming formulation of the minimum spanning tree in Section 13.8 merely touches upon two very important topics in combinatorial optimization. Although the concept of matroids is quite old, dating from their introduction by Whitney [1935], their use in...

The description of the polyhedral structure of combinatorial optimization problems via linear programming has become a very fertile field in combinatorial optimization that has shed theoretical light on many problems and led to effective algorithms for solving many important applications. The comprehensive text by Nemhauser and Wolsey [1988] gives an instructive account of this field, known as polyhedral combinatorics. The linear programming description of the minimum spanning tree problem, and the interpretation of Kruskal's algorithm as a method for solving the linear programming formulation of the problem, has served as an important stimulus for developments of this field. As but one example, this approach has proven very fruitful in developing algorithms for solving the nonbipartite matching problems that we considered in Chapter 12 from a purely combinatorial approach. For a polynomial formulation of the minimal spanning tree problem using multicommodity flow variables, see the survey by Magnanti, Wolsey, and Wong [1992].

The applications of the minimum spanning tree problem that we presented in Section 13.2 are adapted from the following papers:

1. Designing physical systems (Borůvka [1926], Prim [1957], Loberman and Weinberger [1957], and Dijkstra [1959]).
2. Optimal message passing (Prim [1957]).
3. All pairs minimax path problem (Hu [1961]).
4. Reducing data storage (Kang, Lee, Chang, and Chang [1977]).
5. Cluster analysis (Gower and Ross [1969], and Zahn [1971]).

In Application 1.7 we described another application of the spanning tree problem that arises in measuring the homogeneity of bimetallic objects (Shier [1982] and Filliben, Kafadar, and Shier [1983]). Additional applications of the minimum spanning tree problem arise in (1) solving a special case of the traveling salesman problem (Gillmore and Gomory [1964]), (2) chemical physics (Stillinger [1967]), (3) Lagrangian relaxation techniques (Held and Karp [1970]), (4) network reliability analysis (Van Slyke and Frank [1972]), (5) pattern classification (Dude and Hart [1973]), (6) picture processing (Osteen and Lin [1974]), and (7) network design (Magnanti and Wong [1984]). The survey paper of Graham and Hell [1985] provides references for additional applications of the minimum spanning tree problem.

**EXERCISES**

13.1. Suppose that you want to determine a spanning tree \( T \) that minimizes the objective function \( \sum_{e \in \text{tree}} (c_e)^{1/2} \). How would you solve this problem?

13.2. In the network shown in Figure 13.14, the bold lines represent a minimum spanning tree.

(a) By listing each nontree arc \((i, j)\) and the minimum length arc on the tree path from node \( k \) to node \( i \), verify that this tree satisfies the path optimality conditions.

(b) By listing each tree arc \((i, j)\) and the minimum length arc in the cut defined by the arc \((i, j)\), verify that the tree satisfies the cut optimality conditions.

13.3. Using Kruskal's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

(a) 

(b) 

**Figure 13.15 Examples for Exercises 13.3 to 13.5.**

13.4. Using Prim's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

13.5. Using Sollin's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

13.6. Think of the network shown in Figure 13.16 as a highway map, and the number recorded next to each arc as the maximum elevation encountered in traversing the arc. A traveler plans to drive from node 1 to node 12 on this highway. This traveler dislikes high altitudes and so would like to find a path connecting node 1 to node 12.
that minimizes the maximum altitude. Find the best path for this traveler using a minimum spanning tree algorithm.

13.7. Can you generalize the approach outlined in Application 13.3 to solve the all-pairs maximum capacity path problem in directed networks? If yes, describe your algorithm; if not, why not?

13.8. In Theorem 13.3 we proved the sufficiency of the path optimality conditions using the cut optimality conditions. Give a direct proof of this sufficiency condition that does not use the cut optimality conditions.

13.9. Prove the maximum spanning tree optimality conditions stated in Theorem 13.4.

13.10. Let \((p, q)\) be a minimum cost arc in \(G\). Show that \((p, q)\) belongs to some minimum spanning tree of \(G\). Does every minimum spanning tree of \(G\) contain the arc \((p, q)\)?

13.11. Show that a maximum weight acyclic subgraph in an undirected graph \(G\) with strictly positive arc weights \(c_{ij}\) must be a spanning tree.

13.12. How would you modify Kruskal’s and Prim’s algorithms to solve the maximum spanning tree problem?

13.13. In an undirected network, we define a tree of shortest paths as a spanning tree in which the unique path from a specified node \(s\) to every other node is a shortest path. Is a minimum spanning tree of \(G\) also a tree of shortest paths? Either prove this result or construct an example to show that the trees could be different.

13.14. Tree minimax result. Let \(G = (N, A)\) be an undirected network with a capacity \(u_{ij}\) associated with every arc \((i, j) \in A\). For any spanning tree \(T\) of \(G\), we define its capacity as \(\text{min}(u_{ij} : (i, j) \in T)\), and for any cut \(Q\) of \(G\), we define its value as \(\text{max}(u_{ij} : (i, j) \in Q)\). Show that the capacity of any spanning tree is a lower bound on the value of every cut. Next show that the maximum capacity of any spanning tree equals the minimum value of any cut.

13.15. We say that two spanning trees \(T^*\) and \(T^*\) are adjacent if they have all but one arc in common. Show that for any two spanning trees \(T^*\) and \(T^*\), we can find a sequence of spanning trees \(T^*\), \(T^*\), \(T^*\) with \(T^* = T^*\) and with a unique adjacents for every \(i = 1\) to \(k - 1\).

13.16. Suppose that you are given a graph with each arc colored either red or blue.
   (a) Show how to find a spanning tree with the maximum number of red arcs.
   (b) Suppose that some spanning tree has \(k^*\) red arcs and another spanning tree has \(k^* > k^*\) red arcs. Show that for every \(k, k^* \leq k^*\), some spanning tree has \(k^*\) red arcs.

13.17. Let \(T\) be a spanning tree. For any pair \((i, j)\) of nodes, let \(\beta(i, j)\) denote the least cost arc among the arcs in the tree path joining node \(i\) and node \(j\). Show how to compute \(\beta(i,j)\) for every pair of nodes in a total of \(O(n^2)\) time.

13.18. In a class of undirected networks, suppose that all arc costs are small (i.e., they lie in the interval \([1, k]\) for some small integer \(k\), say \(k = 10\)). How would you implement Kruskal’s and Prim’s algorithms for solving the minimum spanning tree problem in this class of networks?

13.19. Consider the following reverse greedy algorithm:

\[
\text{begir} \\
\text{let the arcs } (i, j), (k, l), \ldots, (m, n) \text{ be arranged in nonincreasing order of their lengths; } \\
G = G; \\
\text{for } k = 1 \text{ to } m \text{ do} \\
\quad \text{if } G' = (i, j, k) \text{ is a connected graph} \\
\quad \quad G = G' = (i, j, k); \\
\text{end}; \\
\text{end}; \\
\text{Show that at the termination of this algorithm, the graph } G' \text{ is a minimum spanning tree.}
\]

13.20. Consider the following algorithm. Arrange the arcs in \(A\) in any arbitrary order and start with a null tree \(T\). Examine each arc \((i, j)\) in \(A\), one by one, and perform the following steps: add arc \((i, j)\) to \(T\) and if \(T\) contains a cycle \(W\), delete from \(T\) an arc of maximum cost from the cycle \(W\). Show that when this algorithm has examined all the arcs, the final tree \(T\) is a minimum spanning tree. Is it possible to implement this algorithm as efficiently as Kruskal’s algorithm? Why or why not?

13.21. Can you use the data structure of Dial’s implementation of Dijkstra’s shortest path algorithm (discussed in Section 4.6) to implement Prim’s algorithm? If so, is the running time of Prim’s algorithm better than the running time of the shortest path algorithm?

13.22. In Section 13.5 we observed a striking resemblance between Prim’s algorithm and Dijkstra’s algorithm. This observation might lead us to conjecture that we can use a radix heap data structure (discussed in Section 4.8) to implement Prim’s algorithm in \(O(m + n \log n/c)\) time. However, this conjecture is not valid. What are the difficulties we would encounter if we attempted to implement Prim’s algorithm using radix heaps?

13.23. The first implementation of Kruskal’s algorithm that we discussed in Section 13.4 selects a nonroot arc \((k, l)\) violating its optimality condition and exchanges this arc with some tree arc of lower cost. Show that no matter which order we use to select the nonroot arcs violating their optimality conditions, we perform at most \(nm\) iterations. (Hint: Let \(f(i, j)\) be the number of arcs in the network whose cost is strictly greater than \(c_{ij}\). Consider the effect on the potential function \(\sum_{(i, j) \in T} f(i, j)\) as we change the spanning tree \(T\).

13.24. Let \(T\) be a minimum spanning tree of an undirected graph \(G = (N, A)\) and let \(Q\) be a set of nonroot arcs \((k, l)\) satisfying the following property: Some arc \((i, j)\) in the tree path from node \(k\) to node \(l\) has the same cost as arc \((k, l)\); that is, \(c_{ij} = c_{k,l}\). Professor May B. Wright claims that every spanning tree in the subgraph \(G^* = (N, T \cup Q)\) is a minimum spanning tree of \(G\). Construct a counterexample to show that Professor Wright’s claim is false.

13.25. Sensitivity analysis. Let \(T^*\) be a minimum spanning tree of a graph \(G = (N, A)\). For any arc \((i, j) \in A\), define its cost interval as the set of values of \(c_{ij}\) for which \(T^*\) continues to be a minimum spanning tree.
   (a) Describe an efficient method for determining the cost interval of a given arc \((i, j)\). (Hint: Consider two cases: When \((i, j) \in T^*\) and when \((i, j) \notin T^*\), and use the cut and path optimality conditions.)
   (b) Describe a method for determining the cost intervals of every arc in \(A\). Your method must be faster than determining the cost intervals of each arc one by one. (Hint: Use the result of Exercise 13.17.)

13.26. Suppose that we have in hand a minimum spanning tree \(T\) for the undirected graph \(G = (N, A)\). Suppose that we add a new node \((n + 1)\) to \(N\) and \(n\) new arcs to \(A\) incident to this node. How fast can you find a minimum spanning tree for the enlarged network \((n + 1)\)? Use the minimum spanning tree \(T\) of \(G\). (Hint: Use the cut optimality conditions.)

13.27. Arc additions and deletions. Let \(T^*\) be a minimum spanning tree for the undirected graph \(G = (N, A)\). Describe an algorithm for reoptimizing the minimum spanning tree when we delete an arc \((i, j) \in A\) from the network. Similarly, describe an algorithm for reoptimizing the problem when we add a new arc \((i, j) \in A\). Prove that your algorithms correctly find new minimum spanning trees and state their running times.

13.28. Spanning trees containing specific arcs. In an undirected graph \(G = (N, A)\), let \((p, q)\) be a specified arc. Describe a method for identifying a minimum spanning tree \(T\) such that the condition that the tree must contain the arc \((p, q)\). Prove that your method correctly solves this problem. Generalize the method for situations in which the minimum spanning tree must contain an acyclic set \(A'\) of arcs. (Hint: Assign appropriate costs to the arcs required to be in the optimal tree.)

13.29. Factored minimum spanning tree problem. Let \(G\) be a complete undirected graph. Suppose that we associate a positive real number \(a_i\) with each node \(i \in N\) and define
the cost of each arc \((i, j)\) as \(c_{ij} = \alpha_{ij}\). This specialized minimum spanning tree problem is known as the binary minimum spanning tree problem. We wish to describe an algorithm for solving this class of problems that is more efficient than the general minimum spanning tree algorithms.

(a) Consider a five-node network with \(a_i = i\). Find a minimum spanning tree in this network.

(b) Use the insight obtained from answering part (a) to develop an \(O(n)\) algorithm for solving the factored minimum spanning tree problem.

13.30. Most vital arcs. In the minimum spanning tree problem, we refer to an arc as a vital arc if its deletion strictly increases the cost of the minimum spanning tree. A most vital arc is a vital arc whose deletion increases the cost of the minimum spanning tree by the maximum amount. (a) Does a network always contain a vital arc?

(b) Suppose that a network contains a vital arc. Describe an \(O(n^3)\) algorithm for identifying a most vital arc. Can you develop an algorithm that runs faster than \(O(n^3)\) time? (Hint: Use the cut optimality conditions.)

13.31. Suppose that we arrange all the spanning trees of a graph \(G\) in nondecreasing order of their costs. We refer to a spanning tree \(T\) as a \(k\)th minimum spanning tree if it is at the \(k\)th position in this order. Describe an \(O(n^2)\) algorithm for finding the second minimum spanning tree. (Hint: Observe that the second minimum spanning tree must contain at least one arc that is not in the first minimum spanning tree. Then use the result of Exercise 13.17.)

13.32. Bottleneck spanning trees. A spanning tree \(T\) is a bottleneck spanning tree if the maximum arc cost in \(T\) is as small as possible from among all spanning trees. Show that a minimum spanning tree of \(G\) is also a bottleneck spanning tree of \(G\). Is the converse result also true (i.e., is a bottleneck spanning tree of \(G\) also a minimum spanning tree of \(G\))? Either prove this result or construct a counterexample.

13.33. Describe an \(O(m \log n)\) algorithm, using binary search, for solving the bottleneck spanning tree problem defined in Exercise 13.32.

13.34. Balanced spanning trees. A spanning tree \(T\) is a balanced spanning tree if from among all spanning trees the difference between the maximum arc cost in \(T\) and the minimum arc cost in \(T\) is as small as possible. Describe an \(O(m^2)\) algorithm for determining a balanced spanning tree of Theorem 13.9.

13.35. Parametric analysis of minimum spanning tree problem. In the parametric minimum spanning tree problem, each arc length \(c_{ij} = c_{ij} + \lambda\) is a linear function of a parameter \(\lambda\). Let \(T^\lambda\) denote a minimum spanning tree with arc lengths chosen as \(c_{ij} + \lambda\) for a specific value of \(\lambda\). (a) Show that for sufficiently large values of the constant \(k > 0\), \(T^{-k}\) and \(T^k\) are the maximum arc minimum spanning trees when the arc lengths are \(c_{ij}\).

(b) Show that \(T^\lambda\) is a minimum spanning tree for all of the values of \(\lambda\) in some interval \([\alpha, \beta]\). Moreover, show that at the lower and upper limits of this interval, at least two alternate minimum spanning trees are adjacent in the sense of Exercise 13.15.

(c) Describe an algorithm for determining a minimum spanning tree for all values of \(\lambda\) from \(-\infty\) to \(\infty\).

13.36. (a) Show that in the parametric minimum spanning tree problem, as we vary \(\lambda\) from \(-\infty\) to \(\infty\), we obtain at most \(m^2\) minimum spanning trees and every two consecutive minimum spanning trees are adjacent. (Hint: Use the fact that \(T^\lambda\) and \(T^\mu\) are two consecutive minimum spanning trees, we can obtain \(T^\mu\) from \(T^\lambda\) by replacing a tree arc \((i, j)\) by a non-tree arc \((k, l)\) satisfying the condition \(c_{ij} \leq c_{kl}\)).

(b) Consider a special case of the parametric minimum spanning tree problem in which each \(c_{ij} = 0\) or 1. Show that in this case, we vary \(\lambda\) from \(-\infty\) to \(\infty\), we obtain at most \(n^2\) minimum spanning trees.

Consider another special case of the parametric minimum spanning tree problem in which each \(c_{ij} = 1\) for a specific arc \((p, q)\) and is zero for all other arcs. Show how to find minimum spanning trees for all values of \(\lambda\) in time proportional to solving a single minimum spanning tree problem.

13.37. Minimum ratio spanning trees (Chandrasekar [1977]). In the minimum ratio spanning tree problem, we associate two numbers, \(c_{ij}\) and \(\gamma_{ij}\), with each arc \((i, j)\) in a network \(G\) and wish to determine a spanning tree \(T^*\) that minimizes \(\sum_{(i, j) \in T^*} c_{ij}/(\sum_{(i, j) \in T^*} \gamma_{ij})\) from among all spanning trees. We assume that \(\sum_{(i, j) \in T^*} \gamma_{ij} > 0\) for all spanning trees \(T^*\). Suggest a binary search algorithm for identifying a minimum ratio spanning tree of \(G\) that runs in polynomial time.

13.38. A \(k\)-tree of \(G\) is a spanning tree of \(G\) plus one arc. Show that the minimum spanning tree of \(G\) plus the least cost non-tree arc defines a minimum cost \(1\)-tree of \(G\). Suppose that the additional arc must be adjacent to a particular node \(x\) of \(G\). How would you find a minimum cost \(1\)-tree for this version of the problem?

13.39. Optimal 1-forest. A set of arcs is a 1-forest of an undirected graph \(G\) if some arc \((k, l)\) in \(F\) satisfies the condition that \(F - ((k, l))\) is a forest.

(a) Show that the collection of all 1-forests forms a matroid.

(b) Give a greedy algorithm for identifying a maximum weight 1-forest of \(G\).

(c) How would you modify your answers to parts (a) and (b) if we required that the arc \((k, l)\) be incident to a specific node \(s\) of the network?

13.40. Optimal \(k\)-forest. A set \(F\) of arcs is a \(k\)-forest of an undirected graph \(G\) if some subset \(F' \subseteq F\) containing \(k\) or fewer arcs satisfies the condition that \(F - F'\) is a forest. Show that the collection of all \(k\)-forests forms a matroid and give a greedy algorithm for identifying a maximum weight \(k\)-forest of \(G\). (Hint: Generalize the result in Exercise 13.39.)

13.41. Let \(F_0\) and \(F_{p+1}\) be forests in a graph containing \(p\) and \(p + 1\) arcs. Show that we can always add some arc in \(F_{p+1}\) to \(F_0\) to produce a forest with \(p + 1\) arcs.

13.42. Using the example we have considered in the text as motivation, give a formal proof of Theorem 13.9.

13.43. Linear programming proof of the greedy algorithm. Let \((E, S)\) be a matroid with an associated weight \(w_e\) for \(e \in E\). Let \(x_e\) be a zero–one vector indicating whether or not the element \(e\) is a member of a set \(I\) from \(E\); that is, \(x_e = 1\) if \(e \in I\) and \(x_e = 0\) if \(e \notin I\). For any subset \(S \subseteq E\), let \(r(S)\) denote its rank, defined as the number of elements of the largest independent set in \(S\). For example, the rank of a set of 3 arcs in a graph is the size of the largest forest defined by these arcs.

(a) Show that the incidence vectors \(x_e\) of a basis of the matroid satisfy the following conditions:

\[
\begin{align*}
\sum_{e \in E} x_e &= r(E), \\
\sum_{S \subseteq E} x_S &= r(S) & \text{for all } S \subseteq E, \\
x_e &= 0.
\end{align*}
\]

(b) Show that for the minimum spanning tree problem, the constraints in (13.3) contain all of the constraints in the formulation (13.2).

(c) Mimicking the proof of Theorem 13.9 (see Exercise 13.42), give a linear programming proof that the greedy algorithm solves the matroid optimization problem of finding a basis of the matroid \((E, S)\) with the smallest possible weight \(w(\delta(S))\).

13.44. Linear programming formulation of matroids. In Theorem 13.10 we showed that spanning trees of a graph correspond to the extreme points of the linear program (13.2).
Using the result of Exercise 13.43, show that the bases of a matroid correspond to the extreme points of the polyhedron defined by the constraints given in (13.3). (Hint: Use the result of Exercise 13.43 and the fact that each extreme point of a linear program is the unique optimal solution for some choice of the objective coefficients.)

13.45. In Exercise 13.43 we showed that the greedy algorithm solves the matroid optimization problem. Show that this property actually characterizes matroids. That is, show that the greedy algorithm will solve the minimum weight independent set problem for any choice of the element weights if and only if the subset system is a matroid. (Hint: Any subset system that is not a matroid contains two independent subsets \( I \) and \( I' \) satisfying the property that \(| I' | > | I |\) and no element in \( I' \) can be added to \( I \) to obtain an independent set. Define the weight function on \( E \) appropriately so that the greedy algorithm terminates with \( I \), but \( I' \) is optimal.)

13.46. (a) The set of minimum spanning trees \( T^1, T^2, \ldots, T^{k-1} \) that we determined in Exercise 13.36(d) as we varied the parameter from \( C + 1 \) to \(-\infty\) satisfy the "monotonicity" property that once an arc \((1, j)\) belongs to any tree \( T^p \), it also belongs to all of the trees \( T^q \) for \( q \geq p \). Suppose that the parametric cost of arc \((1, j)\) is \( c_{U} + \lambda d_{1j} \) for some constant \( d_{1j} \) and that the cost of arc \((i, j)\) is \( c_{U} \) for \( i \neq 1 \) and \( j \neq 1 \). Does the set of optimal spanning trees, as we vary \( \lambda \) from \( C + 1 \) to \(-\infty\), satisfy the monotonicity property?

(b) If possible, describe a polynomial time variant of the procedure discussed in Exercise 13.36(d) that will solve the parametric problem defined in part (a). If you cannot describe any such algorithm, explain the difficulties encountered.