CHAPTER 5

Spanning Trees

Chapter Goals

Present a classic application of trees to a problem in chemistry.

Define the concepts of spanning tree and spanning forest.

Present Cayley's Theorem for counting labeled trees.

Define Prüfer codes and their uses.

Examine how to find all of the spanning trees of a graph.

Introduce the notion of minimum cost spanning tree.

Describe KRUSKAL'S and PRIM'S algorithms.

Present several applications of spanning trees.

Examine degree-constrained spanning trees.

5.1 Introduction

Graph theory is full of many useful classes of trees. In this chapter we examine some of the most important ones. In §5.2 we describe Cayley's application of trees to a problem in chemistry. This is one of the earliest applications of trees. This practical application serves as a nice introduction to §5.3 and our discussion of spanning trees—another important notion in graph theory. We present some useful and practical methods for counting the number of spanning trees of a given graph. We also present Cayley's classic result on finding the number of spanning trees in the complete graph on n vertices. In §5.4 we present Cayley's Theorem for counting labeled trees. There we introduce *Prüfer codes* that provide us with another representation of trees. We explore how to find all the spanning trees of a graph in §5.5. In §5.6 we study weighted graphs and present a couple of classic algorithms for computing minimum cost spanning trees. We conclude this chapter in §5.7 by looking at degree-constrained spanning trees. These have important applications in electrical engineering and computer science.

5.2 Counting Hydrocarbons Using Trees

In 1857 Arthur Cayley discovered trees while he was trying to count the number of structural isomers of the saturated hydrocarbons C_kH_{2k+2}, where k is a natural number greater than or equal to one. He used a connected graph G to represent the C_kH_{2k+2} molecule. Corresponding to their chemical valences, a carbon atom was represented by a vertex of degree four and a hydrogen atom by a vertex of degree one. So, a hydrogen atom was represented by a leaf. The total number of vertices in such a graph is $n = 3k + 2$. So, by the Hand Shaking Theorem 1.24, the total number of edges is as follows:

$$\begin{aligned} |E(G)| &= \frac{1}{2}\sum_{u\in V(G)} d(u) \\ &= \frac{1}{2}((4\cdot k) + 1\cdot(2k+2)) \\ &= 3k+1. \end{aligned}$$

Since G is connected and the number of edges is one less than the number of vertices, G is a tree. Thus counting structural isomers of a given hydrocarbon is equivalent to counting trees having certain properties.

A graph in which each vertex is assigned an unique name or label is called a *labeled graph*. A subtree of such a graph is called a *labeled tree*. Cayley asked the following question: What is the number of different labeled trees that one can construct with n vertices? If n equals four for instance, we have 16 trees.

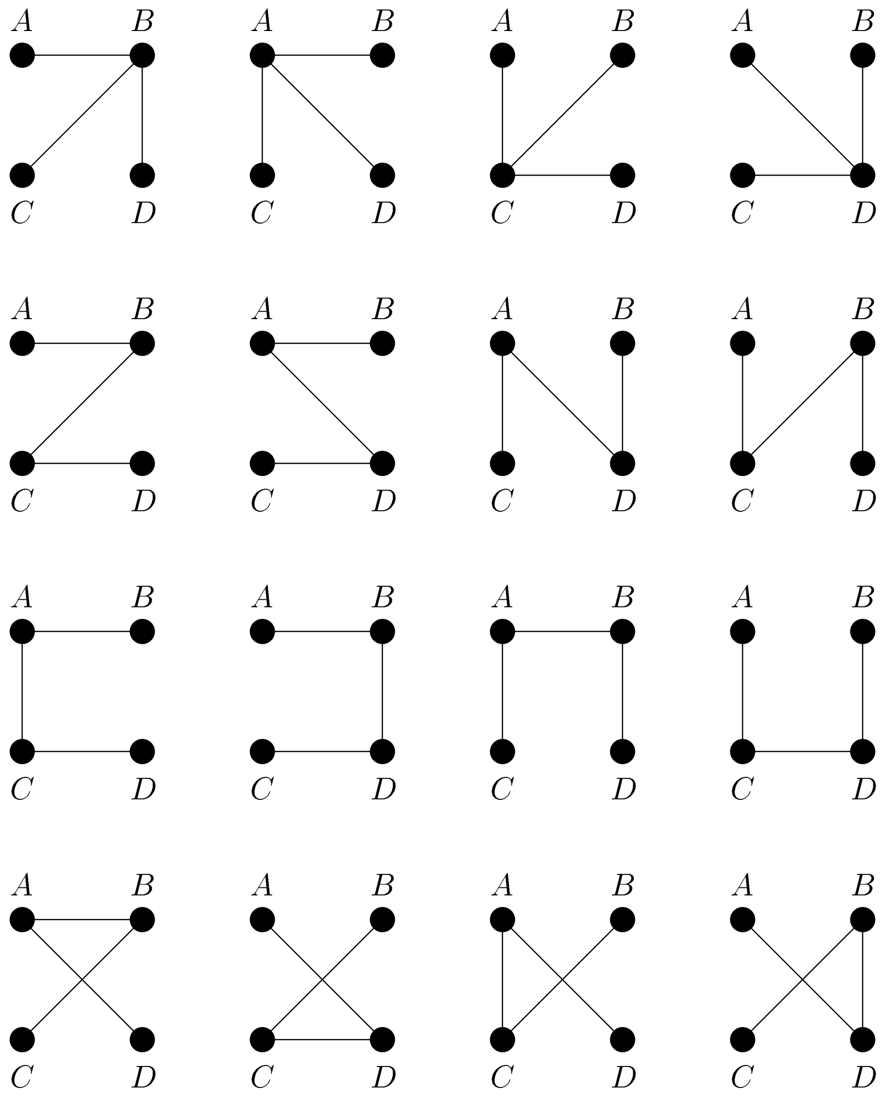

Figure 5.1: All 16 trees of four labeled vertices.

These labeled trees are shown in Figure 5.1. The reader can check that there are no more labeled trees of four vertices.

The distinction between a labeled and an unlabeled graph is very important when we are counting the number of different possible graphs. For instance, the four graphs in the first row of Figure 5.1 are counted as four different trees because the vertices are labeled differently even though the trees are isomorphic.

If there were no distinction made between A, B, C, or D, these four trees would be counted as one. A careful inspection of the graphs in Figure 5.1 reveals that there are only two unlabeled trees on four vertices with no distinction made between A, B, C, and D.

In the next section we examine an important class of trees called *spanning trees*. This notion is a helpful one in working towards a solution to Cayley's question. We use spanning trees in §5.4 where we present Cayley's Theorem for counting labeled trees. There we come back to Cayley's original question.

5.3 Spanning Trees

A fixed graph G has numerous subgraphs. In fact, a graph having $|E(G)|$ edges has $2^{|E(G)|}$ possible distinct subgraphs. Obviously, some of these subgraphs are trees. Out of these trees, we are particularly interested in certain types of trees called *spanning trees*.

Definition 5.1
Let G be a graph.

1. *A subtree T of G is called a* spanning tree *if T contains all of the vertices of G. That is, $V(T)$ equals $V(G)$.*
2. *A subforest F of G is called a* spanning forest *if $F \cap H$ is a spanning tree for each of the components H of G.*
3. *The number of distinct spanning forests in G is denoted $\tau(G)$.*

In Figure 5.2 we depict a seven-node graph. A spanning tree of the graph is shown in thick lines. The set of edges in the spanning tree is $\{b_i : 1 \leq i \leq 6\}$.

Remark 5.2
1. *If a graph G is connected, then $\tau(G)$ is the number of spanning trees in G.*
2. *If G' is the graph we get from G by removing all the loops from G, then $\tau(G)$ equals $\tau(G')$.*

The following observation is easy to prove.

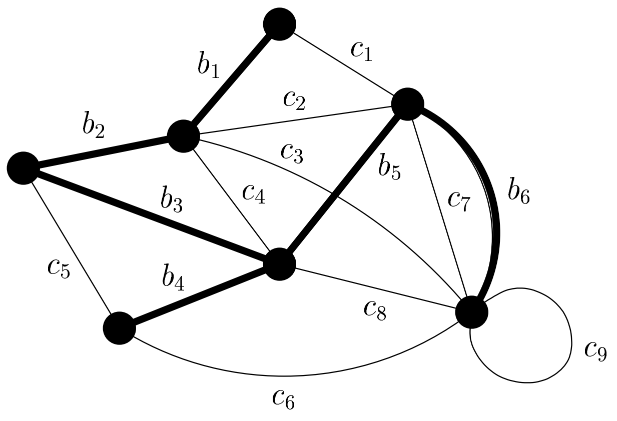

Figure 5.2: A spanning tree.

Observation 5.3
Let G be a graph.

1. *If G is a tree, then $\tau(G)$ equals one.*
2. *If $G_1, \ldots, G_k$ are the components of G, then*
$$\tau(G) = \tau(G_1) \times \cdots \times \tau(G_k).$$

Before we continue with our discussion of the spanning trees of a graph G, it is helpful to recall Definition 3.12. We defined the contraction of G by an edge $e \in E(G)$. This contraction is denoted $G \cdot e$. With this definition in mind, we can prove the following theorem. This theorem is useful for counting the number of spanning trees of a graph.

Theorem 5.4
Let G be a graph and $e \in E(G)$, where e is not a loop. We have that

$$\tau(G) = \begin{cases} \tau(G \cdot e) + \tau(G - e) & \text{if } e \text{ is not a cut-edge} \\ \tau(G \cdot e) & \text{if } e \text{ is a cut-edge} \end{cases}$$

Proof: By Observation 5.3 we can assume that G is connected. Let $\mathcal{T}(G)$ denote the set of all spanning trees of G. So, $\tau(G)$ equals $|\mathcal{T}(G)|$. By setting

$$\begin{aligned} \mathcal{T}_1 &= \{T \in \mathcal{T}(G) : e \in E(T)\} \text{ and} \\ \mathcal{T}_2 &= \{T \in \mathcal{T}(G) : e \notin E(T)\}, \end{aligned}$$

we get a partition of $\mathcal{T}(G)$ into two disjoint sets $\mathcal{T}(G) = \mathcal{T}_1 \cup \mathcal{T}_2$.

If e is not a cut-edge, then $\mathcal{T}_2 = \mathcal{T}(G-e)$. If on the other hand e is a cut-edge, then $\mathcal{T}_2 = \emptyset$. The map

$$\begin{aligned}\psi : \mathcal{T}_1 &\rightarrow \mathcal{T}(G \cdot e) \text{ defined by}\\ \psi(T) &= T \cdot e\end{aligned}$$

is clearly bijective. Therefore, $|\mathcal{T}_1| = |\mathcal{T}(G \cdot e)|$. Since $\tau(G) = |\mathcal{T}_1| + |\mathcal{T}_2|$, the theorem follows. $\square$

Note that $G-e$ has one less edge than G but the same number of vertices. So, the number $|V(G)| + |E(G)|$ decreases by one in $G-e$. However, $G \cdot e$ has one edge and one vertex less than G, so $|V(G)| + |E(G)|$ decreases by two in $G \cdot e$. This suggests a way to use Theorem 5.4 to prove various properties of $\tau(G)$ by induction on $|V(G)| + |E(G)|$. We also can use Theorem 5.4 to calculate the number of spanning trees of any graph G recursively as shown in Example 5.5. A different way to compute $\tau(G)$ for a given connected graph G is given by the Matrix-Tree Theorem 10.27.

☞ **Example 5.5**

We illustrate how Theorem 5.4 can be applied to compute the number of spanning trees of the graph G shown in Figure 5.3. Noting that e is not a cut-edge and applying Theorem 5.4, we get the following:

$$\begin{aligned}\tau(G) &= \tau(G \cdot e) + \tau(G-e)\\ &= \tau(G \cdot e) + 1\end{aligned}$$

since $G-e$ is a tree. Continuing along these lines, we get

$$\begin{aligned}\tau(G \cdot e) &= \tau((G \cdot e) \cdot e') + \tau((G \cdot e) - e'),\\ &= \tau((G \cdot e)) \cdot e') + 1.\end{aligned}$$

By Remark 5.2 we have that $\tau((G \cdot e)) \cdot e')$ equals one by removing the loop. Hence, we have that $\tau(G)$ equals three.

Calculations like those in Example 5.5 become cumbersome for large graphs. This technique does not provide an efficient method to calculate $\tau(G)$. However, there is one interesting corollary we get for any graph G by induction on $|V(G)| + |E(G)|$.

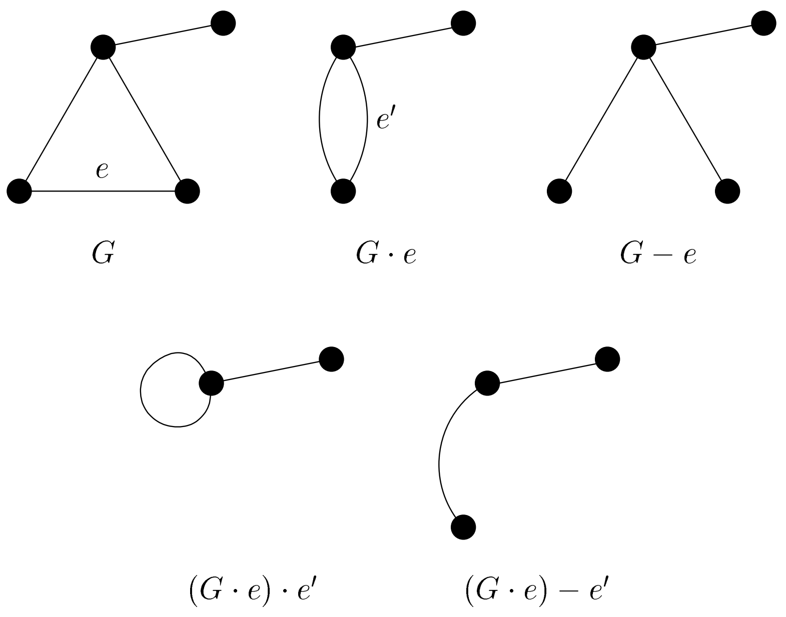

Figure 5.3: A recursive way to calculate $\tau(G)$ for any graph G.

Corollary 5.6
For any graph G, we have that $\tau(G)$ is greater than or equal to one. That is, any graph has a spanning forest. In particular, any connected graph has a spanning tree.

We return to Cayley's question in the next section armed with our knowledge of spanning trees.

5.4 Cayley's Theorem for Counting Labeled Trees

The following well-known theorem for counting labeled trees was first stated and proved by Cayley [5].

Theorem 5.7 (Cayley's Theorem)
Let n be a natural number greater than or equal to two. The number of labeled trees on n vertices is n^{n-2}.

PROOF: We sketch the proof using Prüfer's idea. His proof gives a one-to-one correspondence between the set of all labeled spanning trees of K_n and the set of all ordered $(n-2)$-tuples of the natural numbers $1, \ldots, n$. The encoding of each labeled tree into an ordered $(n-2)$-tuple is called a *Prüfer code*. Once this bijection has been established, we have that the number of labeled trees on n vertices is equal to the number of all possible $(n-2)$-tuples of the natural numbers $1, \ldots, n$. Hence, there are n^{n-2} labeled trees on n vertices.

Assume that we have a tree T on n greater than or equal to two vertices $u_1, \ldots, u_n$. We want to encode T into an ordered tuple $(a_1, \ldots, a_{n-2})$ of natural numbers $1, \ldots, n$. This procedure is called PRÜFER CODE:

```
▷ Let $T'$ equal $T$.
▷ For $i = 1$ to $n - 2$ do the following steps:
    1. Let $u$ equal the smallest numbered leaf in $T'$.
    2. Let $a_i$ be the index of $u$'s neighbor in $T'$.
    3. Let $T'$ be $T' - \{u\}$.
```

Assume we have a given ordered $(n-2)$-tuple $(a_1, \ldots, a_{n-2})$ of natural numbers $1, \ldots, n$. We can construct a tree on vertices numbered $1, \ldots, n$ by the procedure called TREE TO PRÜFER CODE:

```
▷ Let $A$ equal $(a_1, \ldots, a_{n-2})$.
▷ For $i = 1$ to $n - 2$ do the following steps:
    1. Let $b_i$ be the least number among $\{1, \ldots, n\}$ that does not occur in
       $A$.
    2. Connect the vertices numbered $a_i$ and $b_i$ with an edge.
    3. Remove $a_i$ from $A$ and add $b_i$ to $A$ to get the new $(n-2)$-tuple
       $(a_{i+1}, \ldots, a_{n-2}, b_1, \ldots, b_i)$.
▷ Connect the two vertices that are not indexed by any of $b_1, \ldots, b_{n-2}$.
```

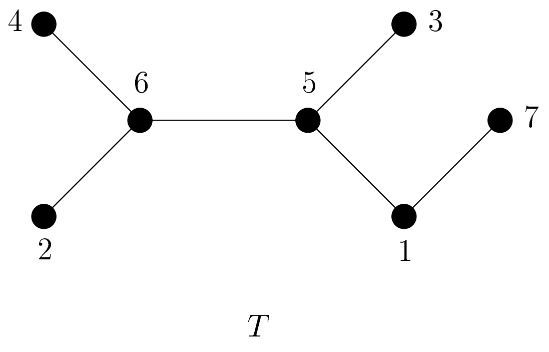

Figure 5.4: The labeled tree T has Prüfer code $(6, 5, 6, 5, 1)$.

We notice that the procedure TREE TO PRÜFER CODE adds edges in the same order that procedure PRÜFER CODE removed them. We also see that starting with an $(n-2)$-tuple, applying TREE TO PRÜFER CODE, and then applying PRÜFER CODE we end up with the original $(n-2)$-tuple. Hence, these procedures give a bijection between the spanning trees of K_n and the ordered $(n-2)$-tuples of the natural numbers $1, \dots, n$. Therefore, we have the theorem. $\square$

Figure 5.4 shows a seven node tree and its corresponding Prüfer code.

There are many different proofs of Cayley's Theorem (Theorem 5.7) with various approaches. An excellent summary of ten such proofs is given by Moon [23]. Other proofs are given in Example 10.28 and in Chapter 14.

Remark 5.8

In the actual counting of the isomers of C_kH_{2k+2}, Cayley's Theorem is not enough. This is because we are not interested in all *labeled trees, but merely all the trees that are not isomorphic. So, we really want to count the* unlabeled spanning trees *of K_n, where n equals $3k+2$ and where the vertices have degree either one or four. The problem of counting unlabeled trees of different types is discussed more thoroughly in Chapter 14. In the next section we look at finding all of the spanning trees of a graph.*

5.5 Finding all Spanning Trees of a Graph

In a given connected graph there are usually a large number of spanning trees. In many applications we need to find all spanning trees. In this section we discuss how to generate all of the spanning trees of a given graph G. Using Theorem 4.7 we get the following definition.

Definition 5.9
Let G be a connected graph and T a spanning tree of G. An edge in T is called a branch *of T in G. An edge in G that is not in T is called a* chord *of G with respect to T. When we add one chord to T in G, we get an unique cycle whose edges consist of edges from T plus the chord. This cycle is called a* fundamental cycle *of G with respect to T.*

In Figure 5.5 we show a graph G and three of its spanning trees T_i with $1 \leq i \leq 3$. One reasonable way to generate spanning trees of (any graph) G is to start with a given spanning tree T_1. If we add a chord, say h, to the tree T_1, this forms a fundamental cycle $F = (u_2, b, u_3, c, u_5, h, u_4, d, u_2)$. The removal of any branch of T_1, say c, from the fundamental cycle F creates a new spanning tree T_2. Additional spanning trees can be found by repeating this process. Note that T_3 is another spanning tree of T but was not found via this process from T_1. Our observation about how to form a new spanning tree motivates the following definition.

Definition 5.10
Let G be a connected graph and T a spanning tree of G. Let b be a branch of T and c a chord of G with respect to T. The operation of forming the new spanning tree

$$T' = (T - b) \cup c$$

from T is called a cyclic interchange.

Generating one spanning tree from another by adding a chord and deleting an appropriate branch is often iterated to solve certain transportation problems.

In the previous discussion for generating spanning trees, instead of initially deleting branch c, we could have deleted either branch b or d. In doing this we would have obtained two additional spanning trees. The tree in Figure 5.5 corresponding to the deletion of b from the fundamental cycle F has edges a, c, d, and h. The tree corresponding to the deletion of d has edges a, b, c, and h.

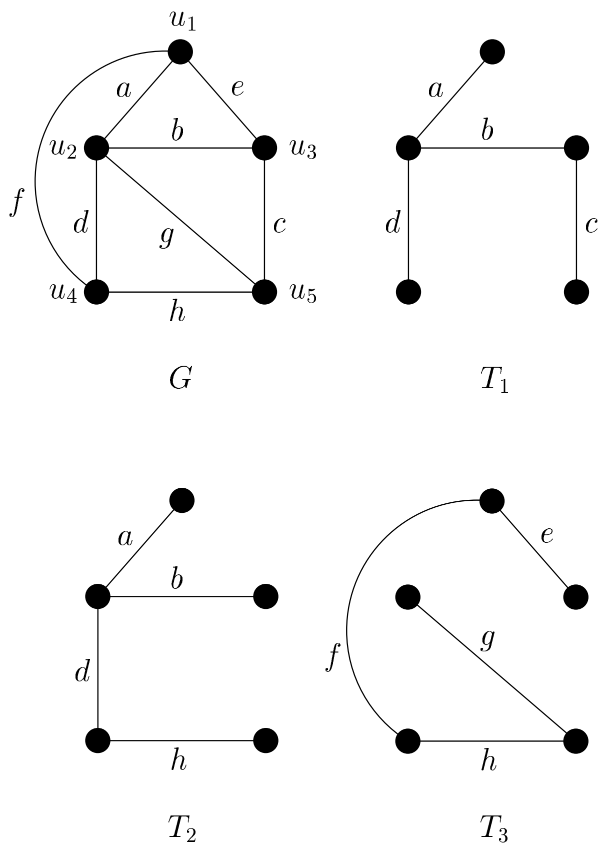

Figure 5.5: A graph and three of its spanning trees.

Moreover, after generating these three trees, each containing chord h, we could restart with T_1 and add a different chord (e, f, or g). We can then repeat the process of obtaining a different spanning tree each time a branch is deleted from the newly formed fundamental cycle. Thus we have a procedure for generating spanning trees for any given graph G.

The procedure raises many questions. Can we start from any spanning tree and get a desired spanning tree by a number of cyclic interchanges? Can we get all spanning trees of a given graph in this fashion? How long do we have to continue exchanging edges? Out of all possible spanning trees which is the best one for starting the procedure? How efficient is this procedure? We will try to answer some of these questions. It is helpful to recall Definition 3.5 of the symmetric difference for two subgraphs of a given graph. The following definition allows us to measure how closely two spanning trees are related.

Definition 5.11
Let G be a connected graph. Let T and T' be two spanning trees of G. Define the distance $D(T, T')$ *between the two trees T and T' by*

$$D(T, T') = \frac{1}{2}|E(T \vartriangle T')|.$$

For example, in Figure 5.5 we have that $D(T_2, T_3)$ equals three. The next remark addresses how many steps are required to convert one spanning tree into another.

Remark 5.12
If G has n vertices, then by Theorem 4.7 both T and T' in Definition 5.11 have $n-1$ edges. If they have k less than or equal to $n-1$ edges in common, then $|E(T \vartriangle T')| = 2(n-1-k)$ is an even number. Therefore, $D(T, T')$ is always a natural number. It is not hard to see that this number is the minimum number of cyclic interchanges needed to obtain T' from T.

For finite sets A, B, and C, we have the following result:

$$|A \vartriangle B| \leq |A \vartriangle C| + |C \vartriangle B|. \tag{5.1}$$

The proof of this inequality is Exercise 14. From Inequality 5.1, we get the following.

Theorem 5.13
If G is a connected graph and $\mathcal{T}(G)$ is the set of all spanning trees of G, then $D : \mathcal{T}(G) \times \mathcal{T}(G) \to \mathbb{N} \subseteq \mathbb{R}$ is a metric on $\mathcal{T}(G)$.

Assume that a given graph G has n vertices and m edges. If T and T' are two spanning trees of G, then $|E(T \vartriangle T')|$ can be at most

$$|E(T)| + |E(T')| \leq 2(n-1).$$

Hence,

$$\max(\{D(T, T') \mid T, T' \in \mathcal{T}(G)\}) \leq n-1.$$

Also, since T (or T' for that matter) has $n-1$ edges, no more than $m-(n-1)$ edges can be replaced in T to get T'. Hence,

$$\max(\{D(T, T') \mid T, T' \in \mathcal{T}(G)\}) \leq m-(n-1).$$

We summarize these observations in the following theorem.

Theorem 5.14
Let G be a connected graph with n vertices and m edges. Starting from any spanning tree, one can obtain every other spanning tree of G by cyclic interchanges. Moreover, if T and T' are two spanning trees, then one can form the tree T' starting from the tree T by at most $D(T,T')$ cyclic interchanges, where

$$D(T,T') \leq \min(\{n-1, m-n+1\}).$$

It is natural to ask if there is a *best* spanning tree to start with if one needs to obtain all other spanning trees of a connected graph G. For a spanning tree T of G, let

$$\text{maxdistance}(T) = \max(\{D(T,T') \mid T' \in mathcalT(G)\}),$$

where T' runs through $\mathcal{T}(G)$—the set of all spanning trees of G. The best spanning tree to start with is called a *central tree* of T and denoted T_c, where maxdistance(T_c) is a minimum. The concept of a central tree is also useful in enumerating all trees of G. In general, a central tree in a graph is not unique. For more on central trees the reader should see [1] and [8].

The *tree graph* of G is useful in a procedure for obtaining all spanning trees of G. It is defined as the graph with vertex set $\mathcal{T}(G)$, and edge set where we connect two spanning trees T and T' if and only if T' can be obtained from T by one cyclic interchange. Note that in this case one can also get T from T' by one cyclic interchange.

From Theorem 5.14 we know that starting from any spanning tree we can obtain all other spanning trees through cyclic interchanges. Therefore, the tree graph of any given connected graph is connected. For additional properties of tree graphs, the reader should refer to [7].

5.6 Spanning Trees in a Weighted Graph

We next discuss how to find the *minimum cost spanning tree* in a given graph G. After some preliminary definitions, we will present the problem formally.

Definition 5.15
Let G be a graph and $W : E(G) \to \mathbb{R}$ be a function.

1. *The ordered tuple* (G, W) *is called a* weighted graph. *The function* W *is called a* weight *or* weight function *of* (G, W).

2. *If* G' *is a subgraph of* G *then*

$$W(G') = \sum_{e \in E(G')} W(e)$$

is called the weight *of* G' *in* G.

When there is no possibility of ambiguity, we write about the weighted graph G instead of the tuple (G, W). In most applications we also assume that $W(e)$ is greater than or equal to zero for all $e \in E(G)$.

As we have seen, a spanning tree in a connected graph G is a minimal subgraph connecting all vertices of G. If the graph G is a weighted graph, then clearly different spanning trees of G may have different weights. Among all spanning trees of G, the one with the *smallest* weight is of practical significance in many applications. There may be several spanning trees with the smallest weight. For instance, in a graph on n vertices in which every edge has unit weight, each spanning tree has weight $n - 1$ units. These ideas motivate the following definition.

Definition 5.16
For a connected weighted graph (G, W), *the spanning tree* T *with the minimum weight* $W(T)$ *is called a* minimum cost spanning tree.

The next example illustrates the practical significance of minimum cost spanning trees.

☞ **Example 5.17**
Suppose that we are to connect n *cities* $u_1, \ldots, u_n$ *through a network of roads. Suppose further that the cost* C_{ij} *of building a direct road between* u_i *and* u_j *is given for all pairs of cities where roads can be built. The costs are all greater than zero. There may be pairs of cities between which no direct road can be built. In this case we set* C_{ij} *equal to infinity. (When programming a solution to this problem, a value such as* maxint *can be used in place of infinity.) The problem is to find the least expensive network that connects all* n *cities. It is immediately evident that this connected network must be a tree. If not, we could remove some edge(s) and get a connected graph with smaller weight.*

Thus the problem of connecting n cities with a least expensive network is the problem of finding a minimum cost spanning tree in a connected weighted graph of n vertices. One can easily see how minimum cost spanning trees can be used to solve many problems involving other types of networks including airline routes, wiring layouts, and so forth.

How do we actually compute a minimum cost spanning tree? The following algorithm, called KRUSKAL'S ALGORITHM, yields a minimum cost spanning tree in any connected weighted graph.

```
KRUSKAL'S ALGORITHM
Input: A connected weighted graph $(G, W)$.
Output: A minimum cost spanning tree $T$ of $G$.
begin
    $T_1 = \emptyset$;
    for $i = 1$ to $|V(G)| - 1$ do {
        let $e_i \in E(G) \setminus E(T_i)$ be a minimum weight edge and such
            that $T_i \cup e_i$ is a forest;
        $T_{i+1} = T_i \cup e_i$;
    }
    output $T = T_{|V(G)|}$;
end.
```

The following theorem shows that KRUSKAL'S ALGORITHM computes a minimum cost spanning tree.

Theorem 5.18
Let (G, W) be a connected weighted graph. KRUSKAL'S ALGORITHM *computes a minimum cost spanning tree T of G.*

PROOF: It is clear that KRUSKAL'S ALGORITHM builds a spanning tree T of G because it constructs a tree having $|V(G)| - 1$ edges. We need to show that if T_1 is another spanning tree of G, then $W(T)$ is less than or equal to $W(T_1)$.

Assume we have labelings $e_1, \ldots, e_m$ ($e'_1, \ldots, e'_m$) of the edges of T (respectively, T_1) such that $W(e_1) \leq \cdots \leq W(e_m)$ (respectively, $W(e'_1) \leq \cdots \leq W(e'_m)$). Since T_1 is not equal to T, there is a least number k such that e_k is not equal to e'_k. Notice that $T_1 \cup e_k$ has an unique cycle C that is not completely contained in T. Hence, there is an edge e' in C that is not in T. In this case we have that

$$T_2 = (T_1 \cup e_k) \setminus e'$$

is also a spanning tree of G. By the algorithm's choice of e_k, we have that $W(e_k)$ is less than or equal to $W(e')$. Hence, $W(T_2)$ is less than or equal to $W(T_1)$.

However, we note that $D(T, T_2)$ is less than $D(T, T_1)$ since T_2 has one more edge in common with T than T_1. If T_2 is not equal to T, we can continue in this fashion and get T_3 with $W(T_3)$ less than or equal to $W(T_2)$ and $D(T, T_3)$ less than $D(T, T_2)$. Continuing in this fashion, we end up with a sequence $T_1, \ldots, T_r$ of spanning trees of G satisfying $W(T_1) \geq \cdots \geq W(T_r)$, and more importantly

$$D(T, T_1) > D(T, T_2) > \cdots > D(T, T_r) = 0.$$

This means that T equals T_r. Hence, $W(T)$ is less than or equal to $W(T_1)$. □

From the proof of Theorem 5.18, we get an interesting corollary that provides a necessary and sufficient condition for a spanning tree to be a minimum cost spanning tree.

Corollary 5.19
Let (G, W) be a connected weighted graph. A spanning tree T is a minimum cost spanning tree if and only if there exists no other spanning tree T' of distance one from T and with weight $W(T')$ less than $W(T)$.

We now consider an example of KRUSKAL'S ALGORITHM.

☞ **Example 5.20**
A connected weighted graph G with six vertices and twelve edges is shown in Figure 5.6. The weights of the graph's edges are given in the symmetric 6×6 matrix in Table 5.1. The (i, j)th entry denotes the weight $W(\{u_i, u_j\})$.

KRUSKAL'S ALGORITHM *begins by selecting a minimum cost edge to include in the forest it is constructing. So, the first edge selected for inclusion is $\{u_1, u_5\}$. The algorithm next selects a minimum cost edge from the remaining edges. The next edge selected must not add a cycle to the forest already constructed. There are two edges of cost seven to choose from. For illustration purposes we select edge $\{u_3, u_4\}$ to include at this step. Notice the edge set $\{\{u_1, u_5\}, \{u_3, u_4\}\}$ forms a forest but not a tree.* KRUSKAL'S ALGORITHM *maintains a forest at each iteration. The next three edges the algorithm selects are $\{u_4, u_6\}$, $\{u_4, u_5\}$, and $\{u_2, u_3\}$ in that order. The resulting spanning tree of G is shown in thick lines in Figure 5.6. The cost of this spanning tree is $5 + 7 + 7 + 8 + 9.5 = 36.5$. This is a minimum.*

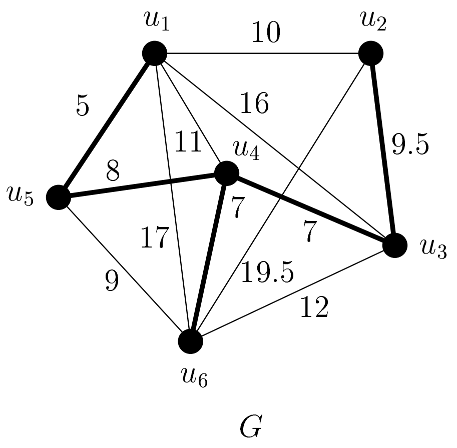

Figure 5.6: Minimum cost spanning tree in a weighted graph G.

	u_1	u_2	u_3	u_4	u_5	u_6
u_1	–	10	16	11	5	17
u_2	10	–	9.5	∞	∞	19.5
u_3	16	9.5	–	7	∞	12
u_4	11	∞	7	–	8	7
u_5	5	∞	∞	8	–	9
u_6	17	19.5	12	7	9	–

Table 5.1: The weights of the graph used in Examples 5.20 and 5.21.

The next algorithm, called PRIM'S ALGORITHM, yields a minimum cost spanning tree in any connected weighted graph.

```
PRIM'S ALGORITHM
Input: A connected weighted graph $(G, W)$ that has no loops.
Output: A minimum cost spanning tree $T$ of $G$.
begin
    $T_1 = (\{u_1\}, \emptyset)$;
    for $i = 1$ to $|V(G)| - 1$ do {
        let $e_i \in E(G) \setminus E(T_i)$ be a minimum weight edge and such
            that $|T_i(V) \cap e_i| = 1$;
        $T_{i+1} = T_i \cup e_i$;
    }
    output $T = T_{|V(G)|}$;
end.
```

We now informally argue the correctness of PRIM'S ALGORITHM. At each phase of the algorithm, the edges obtained thus far form a spanning tree of the subgraph of G induced by these edges. This tree is a minimum cost spanning tree in the corresponding weighted subgraph of G. Therefore, PRIM'S ALGORITHM computes a minimum cost spanning tree of G. We ask for a formal induction proof in Exercise 29.

Here is an example of PRIM'S ALGORITHM.

☞ **Example 5.21**

We use the graph of Figure 5.6 with the weights shown in Table 5.1 to illustrate this algorithm.

PRIM'S ALGORITHM *starts from vertex* u_1. *It then selects a minimum weight edge that has* u_1 *as one of its endpoints. Thus it picks a smallest entry from row one in Table 5.1. This entry from row one is five. The algorithm adds edge* $\{u_1, u_5\}$ *to the tree being constructed. The closest neighbor of the subgraph consisting of the edge* $\{u_1, u_5\}$ *and its endvertices is* u_4. *This can be seen by examining the unused entries in rows one and five. The three remaining edges selected by following the above procedure are* $\{u_4, u_6\}$, $\{u_3, u_4\}$, *and* $\{u_2, u_3\}$ *in this order. The resulting tree is a minimum cost spanning tree. The tree is shown in Figure 5.6 in thick lines. The cost of this spanning tree is* $5+7+7+8+9.5 = 36.5$. *This is a minimum.*

It is interesting to compare Examples 5.20 and 5.21 to see how KRUSKAL'S and PRIM'S algorithms relate. KRUSKAL'S ALGORITHM builds a forest that may consist of many components. On the other hand, PRIM'S ALGORITHM maintains a tree at each phase of its execution. In this case both algorithms computed the same spanning tree. In the exercises we ask the reader to design a connected weighted graph where the algorithms construct different minimum cost spanning trees.

5.7 Degree-Constrained Spanning Trees

We conclude this chapter by considering a *degree-constrained minimum cost spanning tree* problem. In this problem we want to find a spanning tree T of a connected graph G of total weight as small as possible, but where the degree of each node in T is not to exceed a certain bound.

In a minimum cost spanning tree T, resulting from preceding constructions, the degree of a vertex in T ranges from one to $n-1$. In some practical cases an upper limit on the degree of every vertex (of the resulting spanning tree) must be imposed. For instance, in an electrical wiring problem, one may be required to wire together n pins (using as little wire as possible) with no more than three wires wrapped around any individual pin. In this particular case, $d_T(u)$ is less than or equal to three for all $u \in V(T) = V(G)$. In general, the problem may be stated as follows.

Problem 5.22
Given a weighted connected graph (G, W) and a natural number Δ greater than or equal to two find a minimum cost spanning tree T of G such that

$$d_T(u) \leq \Delta$$

for every $u \in V(G)$. Such a spanning tree is called a degree-constrained minimum cost spanning tree.

If Δ equals two, this problem reduces to the problem of finding the minimum cost Hamiltonian path. It also reduces to the traveling salesman problem without the salesman returning to his home base. We discussed the traveling salesman problem at the end of Chapter 3. To date no efficient method of finding an arbitrarily degree-constrained minimum cost spanning tree has been found.

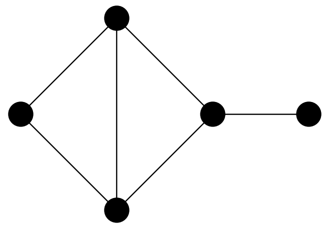

Figure 5.7: A simple graph used in Exercises 3 and 6.

5.8 Exercises

1. Draw all trees on n labeled vertices for n equals 1, 2, 3, and 5.

2. Draw all spanning trees of the graph G shown in Figure 5.6.

3. Use Theorem 5.4 repeatedly to determine the number of spanning trees of the graph shown in Figure 5.7.

4. What is the Prüfer code corresponding to the tree shown in Figure 5.4 if you swap numbers one and six?

5. Draw the tree whose Prüfer code is $(1, 1, 1, 1, 6, 5)$.

6. Draw all spanning trees of the graph G shown in Figure 5.7.

7. Show that a leaf of a connected graph G is contained in every spanning tree of G.

8. Prove that any subgraph G' of a connected graph G is contained in some spanning tree of G if and only if G' contains no cycle.

9. How many chords does K_n have with respect to any of its spanning trees?

10. What are $D(T_1, T_3)$ and $D(T_1, T_2)$, where the T_i's are as given in Figure 5.5?

11. Show that a Hamiltonian path of a graph is a spanning tree.

12. Let G be a connected graph and T a spanning tree of G. If C is a cycle of G, show that C has at least one chord of G with respect to T.

13. Find a spanning tree of distance four from the spanning tree with edges $b_1, \ldots, b_7$ in Figure 5.2. List all fundamental cycles with respect to this new spanning tree.

14. Prove the set-theoretic expression displayed in Inequality 5.1. Use it to prove Theorem 5.13.

15. Explain why the distance function (presented in Definition 5.11) between two trees always has a natural number as a value.

16. Can you construct the original graph G if you are given all of its spanning trees? Give reasons explaining when this is possible and when it is not.

17. Let G be a connected graph. Show that the number of chords of G with respect to any spanning tree does not change when you insert a new vertex in the middle of an edge, or remove a vertex of degree two by merging two edges incident to it.

18. Prove that any edge of a connected graph G is a branch of some spanning tree of G. Is it also true that any edge is a chord of G with respect to some spanning tree of G?

19. Let n be a natural number greater than one. Calculate $\tau(C_n)$.

20. Let n be a natural number greater than zero. Calculate $\tau(K_{2,n})$. (Can you calculate $\tau(K_{3,n})$? Do you see a possible pattern?)

21. Let n be a natural number greater than two. Let u be a fixed vertex of the complete graph K_n. Compute the number $\tau_u(n)$ of spanning trees of K_n that contain the vertex u as a leaf. Use this to show that the probability that a vertex in a tree on n vertices is a leaf is approximately $1/e$, where $e = 2.71828182\ldots$ is the base number for the natural logarithm.

22. Let T_1 and T_2 be two spanning trees of a connected graph G. If e is an edge in T_1 but not in T_2, prove that there exists another edge f in T_2, but not in T_1, such that the subgraphs $(T_1 - e) \cup f$ and $(T_2 - f) \cup e$ are also spanning trees of G.

23. Construct the tree graph of K_4.

24. Show that the tree graph of K_4 is Hamiltonian and that every edge is contained in some Hamiltonian cycle. (This holds for general tree graphs by a result of Cummins [7].)

25. Let (G, W) be a connected weighted graph. Assume that there is only one edge e_s of smallest weight in G. Show that every minimum cost spanning tree must contain edge e_s.

26. Let (G, W) be a connected weighted graph such that every edge belongs to some cycle of G. Assume that there is only one edge e_l of largest weight. Show that e_l is not contained in any minimum cost spanning tree of G.

27. By providing a counterexample, show that in Exercise 26 the same can not be said of the edge with the second largest weight.

28. Let (G, W) be a simple connected weighted graph. Let $e_s \in E(G)$ be as in Exercise 25 so that $W(e_s)$ is less than $W(e)$ for all edges e not equal to e_s. As we have seen, every minimum cost spanning tree of G must contain the edge e_s. Now let $e'_s \in E(G)$ be such that $W(e'_s)$ is less than $W(e)$ for all edges e not in $\{e_s, e'_s\}$. Must every minimum cost spanning tree contain the edge e'_s? If $e''_s \in E(G)$ is such that $W(e''_s)$ is less than $E(e)$ for all edges e not in $\{e_s, e'_s, e''_s\}$ must every minimum cost spanning tree contain the edge e''_s?

29. Using induction prove that Prim's Algorithm yields a minimum cost spanning tree for a given connected weighted graph G.

30. Pick 15 large cities in the United States and obtain the distances between them from the World Wide Web. Find a minimum cost spanning tree connecting these cities using both Kruskal's Algorithm and Prim's Algorithm. Did you obtain the same trees? Explain your answer.

31. Draw a connected weighted graph for which Kruskal's Algorithm and Prim's Algorithm find different minimum cost spanning trees. Draw the minimum cost spanning trees the algorithms construct.

32. Let u be a vertex of a connected graph G. Prove that there exists a spanning tree T of G such that the distance from u to every other vertex in G is the same in both G and T.