Chapter 3

Trees

Section 3.1 Fundamental Properties of Trees

Suppose your city is planning to construct a rapid rail system. They want to construct the most economical system possible that will meet the needs of the city. Certainly, a minimum requirement is that passengers must be able to ride from any station in the system to any other station. In addition, several alternate routes are under consideration between some of the stations. Some of these routes are more expensive to construct than others. How can the city select the most inexpensive design that still connects all the proposed stations?

The model for this problem associates vertices with the proposed stations and edges with all the proposed routes that could be constructed. The edges are labeled (weighted) with their proposed costs. To solve the rapid rail problem, we must find a connected graph (so that all stations can be reached from all other stations) with the minimum possible sum of the edge weights.

Note that to construct a graph with minimum edge weight sum, we must avoid cycles, since otherwise we could remove the most expensive edge (largest weight) on the cycle, obtaining a new connected graph with smaller weight sum. What we desire is a connected, acyclic graph (hence, a tree) with minimum possible edge weight sum. Such a tree is called a *minimum weight spanning tree.* Before we determine algorithms for finding minimum weight spanning trees, let's investigate more of the properties of trees.

Trees are perhaps the most useful of all special classes of graphs. They have applications in computer science and mathematics, as well as in chemistry, sociology and many other studies. Perhaps what helps make trees so useful is that they may be viewed in a variety of equivalent forms.

Theorem 3.1.1 A graph T is a tree if, and only if, every two distinct vertices of T are joined by a unique path.

Proof. If T is a tree, by definition it is connected. Hence, any two vertices are joined by at least one path. If the vertices u and v of T are joined by two or more different paths, then a cycle is produced in T, contradicting the definition of a tree.

Conversely, suppose that T is a graph in which any two distinct vertices are joined by a unique path. Clearly, T must be connected. If there is a cycle C containing the vertices u and v, then u and v are joined by at least two paths, contradicting the hypothesis.

Hence, T must be a tree. □

Theorem 3.1.2 A (p, q) graph T is a tree if, and only if, T is connected and $q = p - 1$.

Proof. Given a tree T of order p, we will prove that $q = p - 1$ by induction on p. If $p = 1$, then $T = K_1$ and $q = 0$. Now, suppose the result is true for all trees of order less than p and let T be a tree of order $p \geq 2$. Let $e = uv \in E(T)$. Then, by Theorem 3.1.1, $T - e$ contains no $u - v$ path. Thus, $T - e$ is disconnected and in fact, has exactly two components (see Chapter 3, exercise 1). Let T_1 and T_2 be these components. Then T_1 and T_2 are trees, and each has order less than p; hence, by the inductive hypothesis

$$|E(T_i)| = |V(T_i)| - 1, \quad \text{for } i = 1, 2.$$

Now we see that

$$\begin{aligned} |E(T)| &= |E(T_1)| + |E(T_2)| + 1 \\ &= |V(T_1)| + |V(T_2)| - 1 \\ &= p - 1. \end{aligned}$$

Conversely, suppose T is a connected (p, q) graph and $q = p - 1$. In order to prove that T is a tree, we must show that T is acyclic. Suppose T contains a cycle C and that e is an edge of C. Then $T - e$ is connected and has order p and size $p - 2$. But this contradicts exercise 7 in Chapter 2. Therefore, T is acyclic and hence, T is a tree. □

We summarize various characterizations of a tree in the following theorem.

Theorem 3.1.3 The following are equivalent on a (p, q) graph T:

1. The graph T is a tree.
2. The graph T is connected and $q = p - 1$.
3. Every pair of distinct vertices of T is joined by a unique path.
4. The graph T is acyclic and $q = p - 1$.

In any tree of order $p \geq 3$, any vertex of degree at least 2 is a cut vertex. However, every nontrivial tree contains at least two vertices of degree 1, since it contains at least two vertices that are not cut vertices (see Chapter 2, exercise 28). The vertices of degree 1 in a tree are called *end vertices* or *leaves.* The remaining vertices are called *internal vertices.* It is also easy to see that every edge in a tree is a bridge.

Every connected graph G contains a spanning subgraph that is a tree, called a *spanning tree.* If G is itself a tree, this is clear. If G is not a tree, simply remove edges lying on cycles in G, one at a time, until only bridges remain. Typically, there are many different spanning trees in a connected graph. However, if we are careful, we can construct a subtree in which the distance from a distinguished vertex v to all other vertices in the tree is identical to the distance from v to each of these vertices in the original graph. Such a spanning tree is said to be *distance preserving from v* or *v-distance preserving.* The following result is originally from Ore [7].

Theorem 3.1.4 For every vertex v of a connected graph G, there exists a v-distance preserving spanning tree T.

Proof. The graph constructed in the breadth-first search algorithm starting at v is a tree and is clearly distance preserving from v. □

The following result shows that there are usually many trees embedded as subgraphs in a graph.

Theorem 3.1.5 Let G be a graph with $\delta(G) \geq m$ and let T be any tree of order $m + 1$; then T is a subgraph of G.

Proof. We proceed by induction on m. If $m = 0$, the result is clear since $T = K_1$ is a subgraph of any graph. If $m = 1$, the result is also clear since $T = K_2$ is a subgraph of every nonempty graph. Now, assume the result holds for any tree T_1 of order m and any graph G_1 with $\delta(G_1) \geq m - 1$. Let T be a tree of order $m + 1$ and let G be a graph with $\delta(G) \geq m$.

To see that T is a subgraph of G, consider an end vertex v of T. Also, suppose that v is adjacent to w in T. Since $T - v$ is a tree of order m and the graph $G - v$ satisfies $\delta(G - v) \geq m - 1$, we see from the inductive hypothesis that $T - v \subseteq G - v \subseteq G$. Since $deg_G\ w \geq m$ and $T - v$ has order m, the vertex w has an adjacency in G outside of $V(T - v)$. But this implies that T is a subgraph of G. □

Section 3.2 Minimum Weight Spanning Trees

To solve the rapid rail problem, we now want to determine how to construct a minimum weight spanning tree. The first algorithm is from Kruskal [6]. The strategy of the algorithm is very simple. We begin by choosing an edge of minimum weight in the graph. We then continue by selecting from the remaining edges an edge of minimum weight that does not form a cycle with any of the edges we have already chosen. We continue in this fashion until a spanning tree is formed.

Algorithm 3.2.1 Kruskal's Algorithm.
Input: A connected weighted graph $G = (V, E)$.
Output: A minimum weight spanning tree $T = (V, E(T))$.
Method: Find the next edge e of minimum weight $w(e)$ that does not form a cycle with those already chosen.

1. Let $i \leftarrow 1$ and $T \leftarrow \emptyset$.
2. Choose an edge e of minimum weight such that $e \notin E(T)$ and such that $T \cup \{ e \}$ is acyclic.
   If no such edge exists,
   then stop;
   else set $e_i \leftarrow e$ and $T \leftarrow T \cup \{ e_i \}$.
3. Let $i \leftarrow i + 1$, and go to step 2.

Theorem 3.2.1 When Kruskal's algorithm halts, T induces a minimum weight spanning tree.

Proof. Let G be a nontrivial connected weighted graph of order p. Clearly, the algorithm produces a spanning tree T; hence, T has $p - 1$ edges. Let

$$E(T) = \{e_1, e_2, \ldots, e_{p-1}\} \text{ and let } w(T) = \sum_{i=1}^{p-1} w(e_i).$$

Note that the order of the edges listed in $E(T)$ is also the order in which they were chosen, and so $w(e_i) \leq w(e_j)$ whenever $i \leq j$.

From the collection of all minimum weight spanning trees, let $T_{\min}$ be chosen with the property that it has the maximum number of edges in common with T. If T is not a minimum weight spanning tree, T and $T_{\min}$ are not identical. Let e_i be the first edge of T (following our listing of edges) that is not in $T_{\min}$. If we insert the edge e_i into $T_{\min}$, we get a graph H containing a cycle. Since T is acyclic, there exists an edge e on the cycle in H that is not in T. The graph $H - \{ e \}$ is also a spanning tree of G and

$$w(H - \{ e \}) = w(T_{\min}) + w(e_i) - w(e).$$

Since $w(T_{\min}) \leq w(H - \{ e \})$, it follows that $w(e) \leq w(e_i)$. However, by the algorithm, e_i is an edge of minimum weight such that the graph $< \{ e_1, e_2, \ldots, e_i \} >$ is acyclic. However, since all these edges come from $T_{\min}$,

$< \{ e_1, e_2, \ldots, e_{i-1}, e \} >$ is also acyclic. Thus, we have that $w(e_i) = w(e)$ and $w(H - \{ e \}) = w(T_{\min})$. That is, the spanning tree $H - \{ e \}$ is also of minimum weight, but it has more edges in common with T than $T_{\min}$, contradicting our choice of $T_{\min}$ and completing the proof. □

Example 3.2.1. We demonstrate Kruskal's algorithm on the graph of Figure 3.2.1.

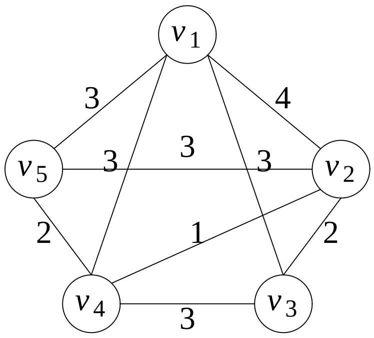


Figure 3.2.1. A weighted graph to test Kruskal's algorithm.

1. $i \leftarrow 1$ and $T \leftarrow \emptyset$. $2-3$. $T \leftarrow e = v_2 v_4$ and $i \leftarrow 2$.

$2-3$. $T \leftarrow T \cup \{ v_2 v_3 \}$ and $i \leftarrow 3$.

$2-3$. $T \leftarrow T \cup \{ v_4 v_5 \}$ and $i \leftarrow 4$.

$2-3$. $T \leftarrow T \cup \{ v_1 v_4 \}$ and $i \leftarrow 5$.

2. Halt (with the minimum spanning tree shown in Figure 3.2.2).

In the performance of Kruskal's algorithm, it is best to sort the edges in order of nondecreasing weight prior to beginning the algorithm. On the average, this can be done in $O(qlog\ q)$ time using either a quicksort or heap sort (see [10]). With this in mind, can you determine the average time complexity of Kruskal's algorithm?

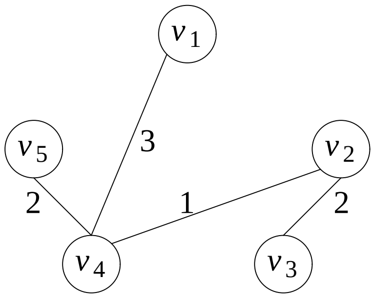


Figure 3.2.2. A minimum spanning tree for the graph of Figure 3.2.1.

Kruskal's algorithm is an example of a type of algorithm known as *greedy*. Simply stated, greedy algorithms are essentially algorithms that proceed by selecting the choice that looks the best at the moment. This local point of view can sometimes work very well, as it does in this case. However, the reader should not think that all processes can be handled so simply. In fact, we shall see examples later in which the greedy approach can be arbitrarily bad.

There are several other algorithms for finding minimum weight spanning trees. The next result is fundamental to these algorithms.

Theorem 3.2.2 Let $G = (V, E)$ be a weighted graph. Let $U \subseteq V$ and let e have minimum weight among all edges from U to $V - U$. Then there exists a minimum weight spanning tree that contains e.

Proof. Let T be a minimum weight spanning tree of G. If e is an edge of T, we are done. Thus, suppose e is not an edge of T and consider the graph $H = T \cup \{ e \}$, which must contain a cycle C. Note that C contains e and at least one other edge $f = uv$, where $u \in U$ and $v \in V - U$. Since e has minimum weight among the edges from U to $V - U$, we see that $w(e) \le w(f)$. Since f is on the cycle C, if we delete f from H, the resulting graph is still connected and, hence, is a tree. Further, $w(H - f) \le w(T)$ and hence $H - f$ is the desired minimum weight spanning tree containing e. □

This result directly inspired the following algorithm from Prim [8]. In this algorithm we continually expand the set of vertices U by finding an edge e from U to $V - U$ of minimum weight. The vertices of U induce a tree throughout this process. The end vertex of e in $V - U$ is then incorporated into U, and the process is repeated until $U = V$. For convenience, if $e = xy$, we denote $w(e)$ by $w(x, y)$. We also simply consider the tree T induced by the vertex set U.

Algorithm 3.2.2 Prim's Algorithm.

Input: A connected weighted graph $G = (V, E)$ with $V = \{ v_1, v_2, \ldots, v_n \}$.

Output: A minimum weight spanning tree T.

Method: Expand the tree T from $\{ v_1 \}$ using the minimum weight edge from T to $V - V(T)$.

1. Let $T \leftarrow v_1$.
2. Let $e = tu$ be an edge of minimum weight joining a vertex t of T and a vertex u of $V - V(T)$ and set $T \leftarrow T \cup \{ e \}$.
3. If $|E(T)| = p - 1$ then halt, else go to step 2.

Example 3.2.2. We now perform Prim's algorithm on the graph of Example 3.2.1.

1. $T \leftarrow \{ v_1 \}$.

2. $w(t, u) = 3$ and $T \leftarrow T \cup \{ v_1 v_4 \}$. (Note that any of $v_1 v_3$, $v_1 v_4$, or $v_1 v_5$ could have been chosen.)

3. Go to step 2.

2. $w(t, v_2) = 1$, $w(t, v_3) = 3$, $w(t, v_5) = 2$ so select $w(t, v_2) = 1$ and set $T \leftarrow T \cup \{ v_2 v_4 \}$.

3. Go to step 2.

2. $w(t, v_3) = 2$, $w(t, v_5) = 2$ so select $w(t, u) = 2$ and set $T \leftarrow T \cup \{ v_4 v_5 \}$.

3. Go to step 2.

2. $w(t, u) = 2$ and so set $T \leftarrow T \cup \{ v_2 v_3 \}$.

3. Halt

We again obtain the minimum spanning tree T of Figure 3.2.2. □

To determine the time complexity of Prim's algorithm, note that step 2 requires at most $|V| - 1$ comparisons and is repeated $|V| - 1$ times (and, hence, requires $O(|V|^2)$

time). Hence, Prim's algorithm requires $O(|V|^2)$ time.

At this stage we must point out that the corresponding problem of finding minimum weight spanning trees in digraphs is much harder. In fact, there is no known polynomial algorithm for solving such a problem.

Section 3.3 Counting Trees

Let's turn our attention now to problems involving counting trees. Although there is no simple formula for determining the number of nonisomorphic spanning trees of a given order, if we place labels on the vertices, we are able to introduce a measure of control on the situation. We say two graphs G_1 and G_2 are *identical* if $V(G_1) = V(G_2)$ and $E(G_1) = E(G_2)$. Now we consider the question of determining the number of nonidentical spanning trees of a given graph (that is, on a given number of vertices). Say $G = (V, E)$ and for convenience we let $V = \{ 1, 2, \ldots, p \}$. For $p = 2$, there is only one tree, namely K_2. For $p = 3$, there are three such trees (see Figure 3.3.1).

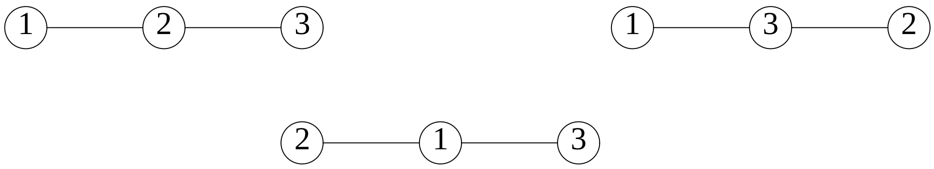


Figure 3.3.1. The spanning trees on $V = \{ 1, 2, 3 \}$.

Cayley [1] determined a simple formula for the number of nonidentical spanning trees on $V = \{ 1, 2, \ldots, p \}$. The proof presented here is from Prüfer [9]. This result is known as Cayley's tree formula.

Theorem 3.3.1 (Cayley's tree formula). The number of nonidentical spanning trees on p distinct vertices is p^{p-2}.

Proof. The result is trivial for $p = 1$ or $p = 2$ so assume $p \geq 3$. The strategy of this proof is to find a one-to-one correspondence between the set of spanning trees of G and the p^{p-2} sequences of length $p - 2$ with entries from the set $\{ 1, 2, \ldots, p \}$. We demonstrate this correspondence with two algorithms, one that finds a sequence corresponding to a tree and one that finds a tree corresponding to a sequence. In what follows, we will identify each vertex with its label. The algorithm for finding the sequence that corresponds to a given tree is:

1. Let $i \leftarrow 1$.
2. Let $j \leftarrow$ the end vertex of the tree with smallest label. Remove j and its incident edge $e = jk$. The ith term of the sequence is k.
3. If $i = p - 2$ then halt; else $i \leftarrow i + 1$ and go to 2.

Since every tree of order at least 3 has two or more end vertices, step 2 can always be performed. Thus, we can produce a sequence of length $p - 2$. Now we must show that no sequence is produced by two or more different trees and that every possible sequence is produced from some tree. To accomplish these goals, we show that the mapping that assigns sequences to trees also has an inverse.

Let $w = n_1, n_2, \ldots, n_{p-2}$ be an arbitrary sequence of length $p - 2$ with entries from the set V. Each time (except the last) that an edge incident to vertex k is removed from the tree, k becomes the next term of the sequence. The last edge incident to vertex k may never actually be removed if k is one of the final two vertices remaining in the tree. Otherwise, the last time that an edge incident to vertex k is removed, it is because vertex k has degree 1, and, hence, the other end vertex of the edge was the one inserted into the sequence. Thus, $deg_T\ k = 1 +$ (the number of times k appears in w). With this observation in mind, the following algorithm produces a tree from the sequence w:

1. Let $i \leftarrow 1$.
2. Let j be the least vertex such that $deg_T\ j = 1$. Construct an edge from vertex j to vertex n_i and set $deg_T\ j \leftarrow 0$ and $deg_T\ n_i \leftarrow deg_T\ n_i - 1$.
3. If $i = p - 2$, then construct an edge between the two vertices of degree 1 and halt; else set $i \leftarrow i + 1$ and go to step 2.

It is easy to show that this algorithm selects the same vertex j as the algorithm for producing the sequence from the tree (Chapter 3, exercise 17). It is also easy to see that a tree is constructed. Note that at each step of the algorithm, the selection of the next vertex is forced and, hence, only one tree can be produced. Thus, the inverse mapping is produced and the result is proved. □

Example 3.3.1. Prüfer mappings. We demonstrate the two mappings determined in the proof of Cayley's theorem. Suppose we are given the tree T of Figure 3.3.2.

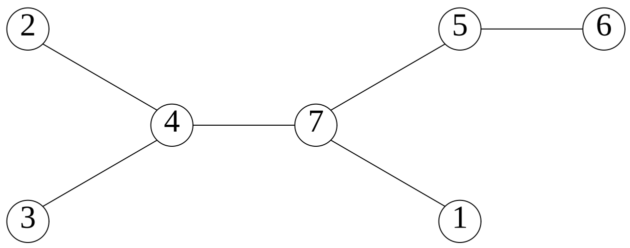


Figure 3.3.2. The tree T.

Among the leaves of T, vertex 1 has the minimum label, and it is adjacent to 7; thus, $n_1 = 7$. Our next selection is vertex 2, adjacent to vertex 4, so $n_2 = 4$. Our tree now appears as in Figure 3.3.3.

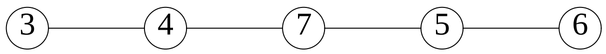


Figure 3.3.3. The tree after the first two deletions.

The third vertex selected is 3, so $n_3 = 4$. We then select vertex 4; thus, $n_4 = 7$. Finally, we select vertex 6, setting $n_5 = 5$. What remains is just the edge from 5 to 7; hence, we halt. The sequence corresponding to the tree T of Figure 3.3.2 is 74475.

To reverse this process, suppose we are given the sequence $s = 74475$. Then we note that:

$$\deg 1 = 1,\ \deg 2 = 1, \deg 3 = 1, \deg 4 = 3,$$
$$\deg 5 = 2,\ \deg 6 = 1, \deg 7 = 3.$$

According to the second algorithm, we select the vertex of minimum label with degree 1; hence, we select vertex 1. We then insert the edge from 1 to $n_1 = 7$. Now set $deg\ 1 = 0$ and $deg\ 7 = 2$ and repeat the process. Next, we select vertex 2 and insert the edge from 2 to $n_2 = 4$:

Figure 3.3.4. The reconstruction after two passes.

Again reducing the degrees, $deg\ 2 = 0$ and $deg\ 4 = 2$. Next, we select vertex 3 and insert the edge from 3 to $n_3 = 4$ (see Figure 3.3.5).

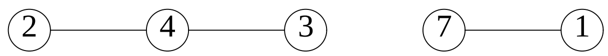


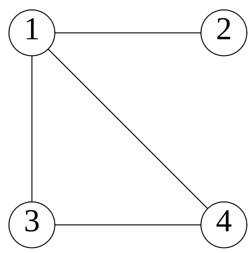
Figure 3.3.5. The reconstruction after three passes.

Now, select vertex 4 and insert the edge from 4 to $n_4 = 7$. This is followed by the selection of vertex 6 and the insertion of the edge from 6 to $n_5 = 5$. Finally, since $i = p - 2$, we end the construction by inserting the edge from 5 to 7, which completes the reconstruction of T. □

An alternate expression for the number of nonidentical spanning trees of a graph is from Kirchhoff [5]. This result uses the $p \times p$ *degree matrix* $C = [\, c_{ij} \,]$ of G, where $c_{ii} = deg\ v_i$ and $c_{ij} = 0$ if $i \neq j$. This result is known as the matrix-tree theorem. For each pair $(i,\ j)$, let the matrix B_{ij} be the $n - 1 \times n - 1$ matrix obtained from the $n \times n$ matrix B by deleting row i and column j. Then det B_{ij} is called the *minor of B* at position $(i,\ j)$ and, $(-1)^{i+j}\det\ B_{ij}$ is called the *cofactor of B* at position $(i,\ j)$.

Theorem 3.3.2 (The matrix-tree theorem) Let G be a nontrivial graph with adjacency matrix A and degree matrix D. Then the number of nonidentical spanning trees of G is the value of any cofactor of $D - A$.

Example 3.3.2. Consider the following graph:



We can use the matrix-tree theorem to calculate the number of nonidentical spanning trees of this graph as follows. The matrices

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

are easily seen to be the adjacency matrix and degree matrix for this graph, while

$$(D - A) = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}.$$

Thus,

$$\det (D - A_{11}) = \det \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} = \det \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & -1 & 2 \end{bmatrix}$$

$$= -1 \det \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = 3.$$

These three spanning trees are easily found since the triangle has three spanning trees. □

Section 3.4 Directed Trees

As with connectivity, directed edges create some additional complications with trees. Initially, we need to decide exactly what we want a directed tree to be. For our purposes, the following definition is most useful: A *directed tree* $T = (V, E)$ has a distinguished vertex r, called the *root,* with the property that for every vertex $v \in V$, there is a directed $r - v$ path in T and the underlying undirected graph induced by V is also a tree. As with trees, directed trees have many possible characterizations. We consider some of them in the next theorem.

If there is an edge e in a digraph D with the property that for some pair of vertices u, v in D, e lies on every $u - v$ path, then we say that e is a *bridge* in D.

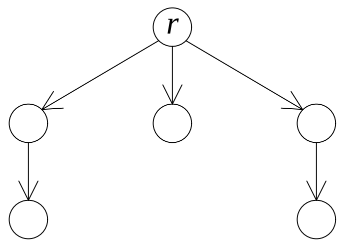


Figure 3.4.1. A directed tree with root r.

Theorem 3.4.1 The following conditions are equivalent for the digraph $T = (V, E)$:

1. The digraph T is a directed tree.
2. The digraph T has a root r, and for every vertex $v \in V$, there is a unique $r - v$ path in T.
3. The digraph T has a root r with $id\ r = 0$, and for every $v \neq r$, $id\ v = 1$, and there is a unique directed $(r - v)$–path in T.
4. The digraph T has a root r, and for every $v \in V$, there is an $r - v$ path and every arc of T is a bridge.
5. The graph underlying T is connected, and in T, there is a vertex r with $id\ r = 0$ and for every other vertex $v \in V$, $id\ v = 1$.

Proof. Our strategy is to show the following string of implications: 1 => 2 => 3 => 4 => 5 => 1.

To see that 1 => 2, assume that T has a root r, there are paths from r to v for every $v \in V$ and the underlying graph of T is a tree. Since there is an $r - v$ path in T and since the underlying graph is a tree, this $r - v$ path must be unique.

To see that 2 => 3, assume that T has a root r and a unique directed path from r to every vertex $v \in V$. Suppose that $e = u \to r$ is an arc of T. Since there is a directed path from r to u, the arc e completes a directed cycle containing r. But then there are at least two paths from r to r, namely the trivial path and the path obtained by following the arcs of this cycle. This contradicts the hypothesis; hence, $id\ r = 0$. Now, consider an arbitrary vertex $v \neq r$. Clearly, $id\ v > 0$ since there is a directed $r - v$ path in T. Suppose that $id\ v > 1$; in fact, suppose that $e_1 = v_1 \to v$ and $e_2 = v_2 \to v$ are two arcs into v. Note that T contains a directed $r - v_1$ path P_1 and a directed $r - v_2$ path P_2. By adding the arc e_1 to P_1 and adding e_2 to the path P_2, we obtain two different

$r - v$ paths, producing a contradiction. If $v \in P_1$ (or P_2) then the segment of P_1 from r to v and the segment of P_2 followed by the arc e_2 are two different $r - v$ paths in T, again producing a contradiction.

To see that 3 => 4, note that the deletion of any arc $e = u \to v$ means that v is unreachable from r; hence, each arc must be a bridge.

To see that 4 => 5, suppose that T has root r and that every arc is a bridge. Since any arc into r can be deleted without changing the fact that there are $r - v$ paths to all other vertices v, no such arc can exist. Hence, $id\ r = 0$. If $v \neq r$, $id\ v > 0$ since there is a directed $r - v$ path in T. Suppose e_1 and e_2 are two arcs into v. Then the path P from r to v cannot use both of these arcs. Thus, the unused arc can be deleted without destroying any $r - v$ path. But this contradicts the fact that every arc is a bridge. Hence, $id\ v = 1$ for every vertex $v \neq r$.

To see that 5 => 1, assume that the graph G underlying T is connected, $id\ r = 0$ and $id\ v = 1$ for all $v \neq r$. To see that there is an $r - v$ path for any vertex v, let P_G be an $r - v$ path in G. Then P_G corresponds to a directed path in T for otherwise, some arc along P_G is oriented incorrectly and, hence, either $id\ r > 0$ or $id\ w > 1$ for some $w \neq r$. Similarly, G must be acyclic or else a cycle in G would correspond to a directed cycle in T. □

A subgraph T of a digraph D is called a *directed spanning tree* if T is a directed tree and $V(T) = V(D)$. In order to be able to count the number of nonidentical directed spanning trees of a digraph D, we need a useful variation of the adjacency matrix. For a digraph D with m arcs from vertex j to vertex k we define the *indegree matrix*, as $A_i(D) = A_i = [\ d_{jk}\ ]$, where

$$d_{jk} = \begin{cases} id\ j & \text{if } j = k \\ -m & \text{if } j \neq k. \end{cases}$$

Using the indegree matrix, we can obtain another characterization of directed trees (Tutte [11]).

Theorem 3.4.2 A digraph $T = (V, E)$ is a directed tree with root r if, and only if, $A_i(T) = [\ d_{jk}\ ]$ has the following properties:

1. The entry $d_{jj} = 0$ if $j = r$ and the entry $d_{jj} = 1$ otherwise.
2. The minor at position (r, r) of $A_i(T)$ has value 1.

Proof. Let $T = (V, E)$ be a directed tree with root r. By Theorem 3.4.1, condition (1) must be satisfied. We now assign an ordering to the vertices of T as follows:

1. The root r is numbered 1.
2. If the arc $u \to v$ is in T, then the number i assigned to u is less than the number j assigned to v.

This numbering is done by assigning the neighbors of r the numbers $2, 3, \ldots, (1 + od\ r)$, and we continue the numbering with the vertices a distance 2 from r, then number those a distance 3 from r, etc.

The indegree matrix $A_i^* = [\, d_{jk}^* \,]$ (with row and column ordering according to our new vertex ordering) has the following properties:

1. $d_{11}^* = 0$.
2. $d_{jj}^* = 1$ for $j = 2, 3, \ldots, |V|$.
3. $d_{jk}^* = 0$ if $j > k$.

Note that A_i^* can be obtained from the original indegree matrix A_i by permuting rows and performing the same permutations on the columns. Since such permutations do not change the determinant except for sign, and since each row permutation is matched by the corresponding column permutation, the two minors are equal. The value of the minor obtained from A_i^* by deleting the first row and column and computing the determinant is easily seen to be 1.

Conversely, suppose that A_i satisfies conditions (1) and (2). By (1) and Theorem 3.4.1, either T is a tree or its underlying graph contains a cycle C. The root r is not a member of C since $id\ r = 0$ and $id\ v = 1$ for all other vertices. Thus, C must be of the form:

$$C\colon x_1, x_2, \ldots, x_a, x_1, \text{ where } x_i \neq r \text{ for each } i = 1, 2, \ldots, a.$$

Any of the vertices may have other arcs going out, but no other arcs may come into these vertices. Thus, each column of A_i corresponding to one of these vertices must contain exactly one +1 (on the main diagonal) and exactly one -1, and all other entries are 0. Each row of this submatrix either has all zeros as entries or contains exactly one +1 and one -1. But then the sum of the entries on these columns is zero, and, hence, the minor at position (r, r) is zero. This contradicts condition (2), and the result is proved. □

Corollary 3.4.1 If D is a digraph with indegree matrix A_i and the minor of A_i is zero, then D is not a directed tree.

Corollary 3.4.2 The number of directed spanning trees with root r of a digraph D equals the minor of $A_i(D)$ resulting from the deletion of the rth row and column.

Proof. Define the digraph D_G obtained from G in the natural manner; that is, for every edge $e = uv$ in G, replace e by the symmetric pair of arcs $u \to v$ and $v \to u$ to form D_G. Now, let $v \in V(D_G)$ (hence, of $V(G)$). There is a one-to-one correspondence between the set of spanning trees of G and the set of directed spanning trees of D_G rooted at r. To see that this is the case, suppose that T is a spanning tree of G and suppose that $e = uv$ is an edge of T. In the directed tree T^* rooted at r, we insert the arc $u \to v$ if $d_T(u, r) < d_T(v, r)$, and we insert the arc $v \to u$ otherwise. Hence, for every spanning tree T of G, we obtain a distinct directed spanning tree T^* of D_G.

Conversely, given a directed tree T^* rooted at r, we can simply ignore the directions on the arcs to obtain a spanning tree T of G. □

Example 3.4.1. Determine the number of spanning trees rooted at vertex 1 of the digraph D of Figure 3.4.2.

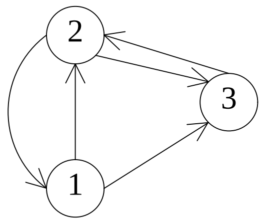


Figure 3.4.2. A digraph D with three spanning trees rooted at 1.

We begin by constructing the indegree matrix A_i of the digraph D.

$$A_i = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

Then, the determinant resulting from the deletion of row 1 and column 1 can be found as:

$$\det \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = 4 - 1 = 3.$$

Hence, D has three spanning trees rooted at vertex 1. Consulting Figure 3.4.3(a), we see these spanning trees are exactly those shown. For spanning trees rooted at vertex 2 we find that there are two such; while the number rooted at vertex 3 is one. These are shown in Figure 3.4.3(b)

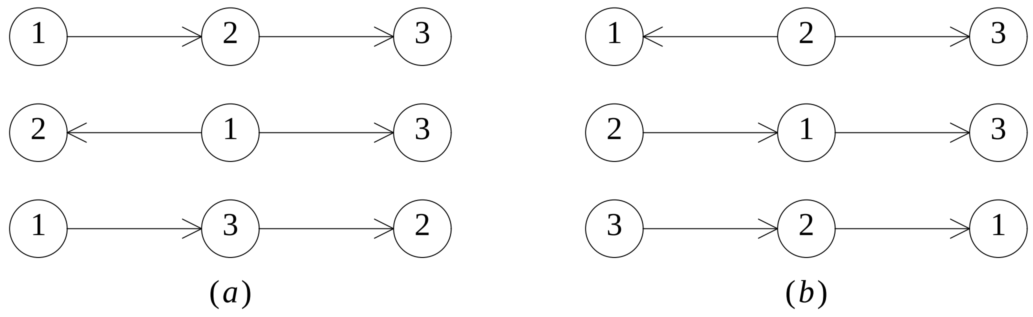


Figure 3.4.3. (a) Directed spanning trees of D rooted at 1 and (b) others.

Suppose we now consider the undirected case. Let $G = (V, E)$ be an undirected graph. We form a digraph D_G from G as follows: Let $V(D_G) = V(G)$, and for every edge $e = uv$ in G, we form two arcs, $e_1 = u \to v$ and $e_2 = v \to u$. If $r \in V(G)$, then there is a 1-1 correspondence between the set spanning trees of G and the set of directed spanning trees of D_G rooted at r. To see that this is the case, let T be a spanning tree of G. If the edge $e = uv$ is in T and if $d_T(u, r) < d_T(v, r)$, then select e_1 for the directed spanning tree T_D; otherwise, select e_2. Conversely, given a directed spanning tree T_D of D_G, it is easy to see that simply ignoring the directions of the arcs of T_D produces a spanning tree of G.

Thus, to compute the number of spanning trees of G, begin by forming D_G. If there are m arcs from vertex i to vertex j, then let

$$A_i(D_G) = \begin{cases} deg_G\ v_i & \text{if } i = j, \\ -m & \text{if } i \neq j. \end{cases}$$

Thus, applying Corollary 3.4.2 produces the desired result; the choice of r makes no difference.

As an example of this, consider the graph of Figure 3.3.2. We earlier determined it had three spanning trees. We now verify this again, using our result on digraphs. First we form D_G.

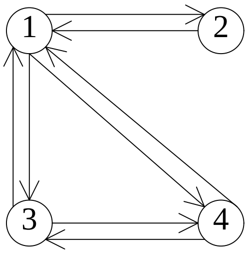


Figure 3.4.4. D_G for the graph of Example 3.3.2.

Now form

$$A_i(D_G) = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}.$$

But note $A_i(D_G)$ equals the matrix $C - A$ of Example 3.3.2 (this is no coincidence!). Hence, we must get the number of spanning trees with the choice of r making no difference, by applying the matrix-tree theorem.

Section 3.5 Optimal Directed Subgraphs

We now wish to consider a problem for digraphs similar to the minimal spanning tree problem for graphs; that is, given a weighted digraph D, we want to find a minimal weight acyclic subgraph of D. Notice that we are not restricting the subgraphs under consideration to directed trees or even to spanning subgraphs, but rather to a somewhat larger class of digraphs often called *branchings.* A subgraph $B = (V, E^*)$ of a digraph $D = (V, E)$ is a branching if B is acyclic and $id\ v \le 1$ for every $v \in V$. If for exactly one vertex r, $id\ r = 0$ and for all other vertices v, $id\ v = 1$, then B is a directed tree with root r.

For finding optimum branchings, the following idea is useful. An arc $e = u \to v$ is called *critical* if it satisifies the following two conditions:

1. $w(e) < 0$.
2. $w(e) \le w(e_1)$ for all other arcs $e_1 = z \to v$.

Form the arc set $E_c \subseteq E$ by selecting one critical arc entering each vertex of V. Using this arc set, we obtain the *critical subgraph* $C = (V, E_c)$. Karp [4] showed the relationship between critical subgraphs and minimum (as well as maximum) branchings. (For maximum branchings, merely reverse the inequalities in the above definition of critical arcs.)

Proposition 3.5.1 Let $C = (V, E_c)$ be a critical subgraph of a weighted digraph $D = (V, E)$. Then

1. Each vertex of C is on at most one cycle.
2. If C is acyclic, then C is a minimum weight branching.

Proof. (1) Suppose that v is on two directed cycles. Then there must be a vertex with indegree at least 2, which is a contradiction to the way we selected arcs for C.
(2) It is clear that if C is acyclic, then it is a branching. Suppose the vertex v has no negatively weighted arcs entering it in D. Then in a branching B of D, either B has no arc entering v or we can remove the arc entering v without increasing the weight of B. It is clear that C has no arc entering v. If the vertex v has negatively weighted arcs entering it in D, then the arc entering v contained in E_c is of minimum weight. Thus, no branching can have a smaller weighted arc at v and, hence, C is a minimum weight branching. □

It is possible to have many different branchings in a given digraph. In fact, some of these branchings may actually be very similar, that is, they may have a large number of arcs in common with one another. In fact, it is possible that simply deleting one arc and inserting another creates a new branching. If $B = (V, E_B)$ is a branching and if $e = u \to v$ is an arc of D that is not in B, then we say that e is *eligible relative to B* if there is an arc $e_1 \in E_B$ such that

$$B_1 = (V, E_B \cup \{ e \} - e_1)$$

is also a branching. We can characterize eligible arcs using directed paths.

Theorem 3.5.1 Let B be a branching of the digraph D and let $e = u \to v$ be an arc of D that is not in B. Then e is eligible relative to B if, and only if, there is no directed $v - u$ path in B.

Proof. Suppose there is a directed $v - u$ path in B. Then when e is inserted into B, a directed cycle is formed. The removal of the arc in B entering v (if any did exist) does not destroy this cycle. Thus, e is not eligible.

Conversely, if there is no directed $v - u$ path in B, then inserting e cannot form a directed cycle. The arc set that results from the insertion of e is not a branching if there already is an arc entering v. Removing any such arc ensures that B is a branching and, hence, e is eligible. □

There is a strong tie between the set of eligible arcs and the arc set of a branching. In fact, we can show that there is a time when they are nearly the same.

Theorem 3.5.2 Let $B = (V, E_B)$ be a branching and C a directed circuit of the digraph D. If no arc of $E(C) - E_B$ is eligible relative to B, then $|E(C) - E_B| = 1$.

Proof. Since B is acyclic, it contains no circuits. Thus, $|E(C) - E_B| \geq 1$. Let $e_1, e_2, \ldots, e_k$ be the arcs of $E(C) - E_B$ in the order in which they appear in C. Say

$$C\colon u_1, e_1, v_1, P_1, u_2, e_2, v_2, P_2, \ldots, v_k, P_k, u_k, e_k, v_k, u_1$$

is the circuit, where the P_i's are the directed paths in both B and C. Since e_1 is not eligible relative to B, by Theorem 3.5.1 there must be a directed path P^* in B from v_1 to u_1. This path leaves P_1 at some point and enters v_k and continues on to u_1, so P^* cannot enter the path p_k after v_k; if it could, B would have two arcs entering the same vertex. Similarly, e_j is not eligible relative to B and, thus, there must be a directed path from v_j to u_j in B. This path must leave P_j at some point and enter P_{j-1} at v_{j-1}. But we now see that B contains a directed circuit from v_1, along a section of P_1, to a path leading to v_k, via part of P_k, to a path leading to v_{k-1}, etc., until it finally returns to v_1. Since B is circuit-free, $k \leq 1$. □

Theorem 3.5.3 Let $C = (V, E_C)$ be a critical subgraph of the digraph $D = (V, E)$. For every directed circuit C^* in C, there exists a branching $B = (V, E_B)$ such that $|E(C^*) - E_B| = 1$.

Proof. Among all maximum branchings of D, let B be one that contains the maximum number of arcs of C. Let $e = u \rightarrow v \in E_C - E_B$. If e is eligible, then

$$E_B \cup \{ e \} - \{ e' \mid e' \text{ enters } v \text{ in } B \}$$

determines another maximum branching which contains more arcs of C than does B; thus, we have a contradiction to our choice of B. Thus, no arc of $E_C - E_B$ is eligible relative to B, and, by the last theorem, $|E(C^*) - E_B| = 1$, for every directed circuit C^* of C. □

Thus, we see that in trying to construct maximum branchings, we can restrict our attention to those branchings which have all arcs (except one per circuit) in common with a critical subgraph $C = (V, E_1)$. Edmonds [2] realized this and developed an algorithm to construct maximum branchings. His approach is as follows: Traverse D, examining vertices and arcs. The vertices are placed in the set B_v as they are examined, and the arcs are placed in the set B_e if they have been previously selected for a branching. The set B_e is always the arc set of a branching. Examining a vertex simply means selecting the critical arc e into that vertex, if one exists. We then check to see if e forms a circuit with the arcs already in B_e. If e does not form a circuit, then it is inserted in B_e and we begin examining another vertex. If e does form a circuit, then we "restructure" D by shrinking all the vertices of the circuit to a single new vertex and assigning new weights to the arcs that are incident to this new vertex. We then continue the vertex examination until all vertices of the final restructured graph have been examined. The final restructured graph contains these new vertices, while the vertices and arcs of the circuit corresponding to a new vertex have been removed. The set B_e contains the arcs of a maximum branching for this restructured digraph.

The reverse process of replacing the new vertices by the circuits that they represent then begins. The arcs placed in B_e during this process are chosen so that B_e remains a maximum branching for the digraph at hand. The critical phase of the algorithm is the rule for assigning weights to arcs when circuits are collapsed to single vertices. This process really forces our choice of arcs for B_e when the reconstruction is performed.

Let $C_1, C_2, \ldots, C_k$ be the circuits of the critical graph (V, H). Let e_i^m be an edge of minimum weight on C_i. Let $\overline{e}$ be the edge of C_i which enters v. For arcs $e = u \rightarrow w$ on C_i, define the new weight $\overline{w}$ as: $\overline{w}(e) = w(e) - w(\overline{e}) + w(e_i^m)$. Edmonds's algorithm is now presented.

Algorithm 3.5.1 Edmonds's Maximum Branching Algorithm.
Input: A directed graph $D = (V, E)$.
Output: The set B_e of arcs in a maximum branching.
Method: The digraph is shrunk around circuits.

1. $B_v \leftarrow B_e \leftarrow \emptyset$ and $i \leftarrow 0$.
2. If $B_v = V_i$, then go to step 13.
3. For some $v \notin B_v$ and $v \in V_i$, do steps $4-6$:
4. $\quad B_v \leftarrow B_v \cup \{ v \}$,

5. find an arc $e = x \rightarrow v$ in E_i with maximum weight.

6. If $w(e) \geq 0$, then go to step 2.

7. If $B_e \cup \{ e \}$ contains a circuit C_i, then do steps $8-10$:

8. $i \leftarrow i + 1$.

9. Construct G_i by shrinking C_i to u_i.

10. Update B_e and B_v and the necessary arc weights.

11. $B_e \leftarrow B_e \cup \{ e \}$.

12. Go to step 2.

13. While $i \neq 0$, do steps $14-17$:

14. Reconstruct G_{i-1} and rename some arcs in B_e.

15. If u_i was a root of an out-tree in B_e,
then $B_e \leftarrow B_e \cup \{ e \mid e \in C_i \text{ and } e \notin e_i^m \}$;

16. else $B_e \leftarrow b_e \cup \{ e \mid e \in C_i \text{ and } e \notin \overline{e}_i \}$.

17. $i \leftarrow i - 1$.

18. $w(B) \leftarrow \sum_{e \in B_e} w(e)$.

Section 3.6 Binary Trees

One of the principle uses of trees in computer science is in data representation. We use trees to depict the relationship between pieces of information, especially when the usual linear (list-oriented) representation seems inadequate. Tree representations are more useful when slightly more structure is given to the tree. In this section, all trees will be rooted. We will consider the trees to be *leveled*, that is, the root r will constitute level 0, the neighbors of r will constitute level 1, the neighbors of the vertices on level 1 that have not yet been placed in a level will constitute level 2, etc. With this structure, if v is on level k, the neighbors of v on level $k + 1$ are called the *children* (or *descendents*) of v, while the neighbor of v on level $k - 1$ (if it exists) is called the *parent* (or *father*, or *predecessor*) of v.

A *binary tree* is a rooted, leveled tree in which any vertex has at most two children. We refer to the descendents of v as the *left child* and *right child* of v. In any drawing of this tree, we always place the root at the top, vertices in level 1 below the root, etc. The left child of v is always placed to the left of v in the drawing and the right child of v is always placed to the right of v. With this orientation of the children, these trees are said to be *ordered.* If every vertex of a binary tree has either two children or no children, then we say the tree is a *full* binary tree.

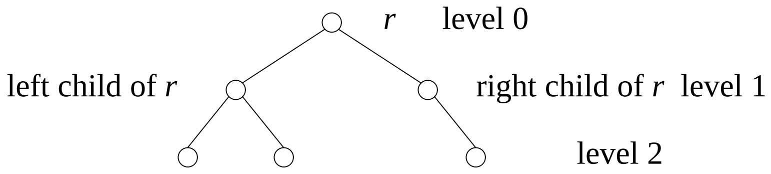


Figure 3.6.1. A binary tree.

The additional structure that we have imposed in designating a distinction between the left and right child means that we obtain distinct binary trees at times when ordinary graph isomorphism actually holds. The trees of Figure 3.6.2 are examples of this situation.

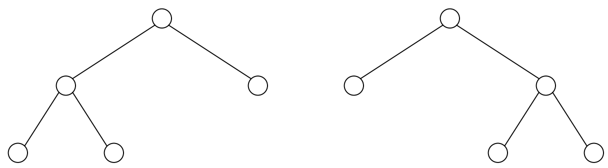


Figure 3.6.2. Two different binary trees of order 5.

The *height* of a leveled tree is the length of a longest path from the root to a leaf, or alternately, the largest level number of any vertex.

As an example of the use of binary trees in data representation, suppose that we wish to represent the arithmetic expression $a + 3b = a + 3 \times b$. Since this expression involves binary relations, it is natural to try to use binary trees to depict these relations. For example, the quantity $3b$ represents the number 3 multiplied by the value of b. We can represent this relationship in the binary tree as shown in Figure 3.6.3.

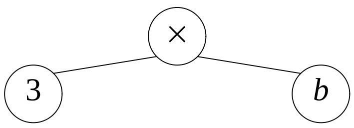


Figure 3.6.3. Representing the arithmetic expression $3b$.

Now the quantity represented by $3b$ is to be added to the quantity represented by a. We repeat the tree depiction of this relationship to obtain another binary tree (see Figure 3.6.4).

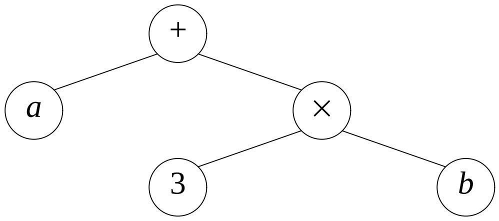


Figure 3.6.4. The expression $a + 3b$ represented using a binary tree.

Once we have a representation for data using a tree, it is also necessary to recover this information and its inherent relationships. This usually involves examining the data contained in (or represented within) the vertices of the tree. This means we must somehow visit the vertices of the tree in an order that allows us not only to retrieve the necessary information but also to understand how these data are related. This visiting process is called a *tree traversal*, and it is done with the aid of the tree structure and a particular set of rules for deciding which neighbor we visit next. One such traversal, the *inorder traversal* (or *symmetric order*) is now presented.

Algorithm 3.6.1 Inorder Traversal of a Binary Tree.
Input: A binary tree $T = (V, E)$ with root r.
Output: An ordering of the vertices of T (that is, the data contained within these vertices, received in the order of the vertices).
Method: Here "visit" the vertex simply means perform the operation of your choice on the data contained in the vertex.

procedure inorder(r)

1. If $T \neq \phi$, then
2. inorder (left child of v)
3. visit the present vertex
4. inorder (right child of v)
end

This recursive algorithm amounts to performing the following steps at each vertex:

1. Go to the left child if possible.
2. Visit the vertex.
3. Go to the right child if possible.

Thus, on first visiting a vertex v, we immediately proceed to its left child if one exists. We only visit v after we have completed all three operations on all vertices of the subtree rooted at the left child of v.

Applying this algorithm to the tree of Figure 3.6.4 and assuming that visit the vertex simply means write down the data contained in the vertex, we obtain the following traversal.

First, we begin at the root vertex and immediately go to its left child. On reaching this vertex, we immediately attempt to go to its left child. However, since it has no left child, we then "visit" this vertex; hence, we write the data a. We now attempt to visit the right child of this vertex, again the attempt fails. We have now completed all operations on this vertex, so we backtrack (or recurse) to the parent of vertex a. Thus, we are back at the root vertex. Having already performed step 2 at this vertex, we now visit this vertex, writing the data +. Next, we visit the right child of the root. Following the three steps, we immediately go to the left child, namely vertex 3. Since it has no left child, we visit it, writing its data 3. Then, we attempt to go to the right child (which fails), and so we recurse to its parent. We now write the data of this vertex, namely $\times$. We proceed to the right child, attempt to go to its left child, write out its data b, attempt to go to the right child, recurse to $\times$ and recurse to the root. Having completed all three instructions at every vertex, the algorithm halts. Notice that the data were written as $a + 3 \times b$. We have recovered the expression.

Two other useful and closely related traversal algorithms are the *preorder* and *postorder* traversals. The only difference between these traversals is the order in which we apply the three steps. In the preorder traversal, we visit the vertex, go to the left child and then go to the right child. In the postorder traversal, we go to the left child, go to the right child and, finally, visit the vertex. Can you write the preorder and postorder algorithms in recursive form?

Another interesting application of binary trees concerns the transmission of coded data. If we are sending a message across some medium, such as an electronic cable, the characters in the message are sent one at a time, in some coded form. Usually, that form is a binary number (that is, a sequence of 1s and 0s). Since speed is sometimes important, it will be helpful if we can shorten the encoding scheme as much as possible, while still maintaining the ability to distinguish between the characters. An algorithm for determining binary codes for characters, based on the frequency of use of these characters, was developed by Huffman [3]. Our aim is to assign very short code numbers to frequently used characters, thereby attempting to reduce the overall length of the binary string that represents the message.

As an example of Huffman's construction, suppose that our message is composed of characters from the set $\{ a, b, c, d, e, f \}$ and that the corresponding frequencies of these characters is $(13, 6, 7, 12, 18, 10)$. Huffman's technique is to build a binary tree based on the frequency of use of the characters. More frequently used characters appear closer to the root of this tree, and less frequently used characters appear in lower levels. All characters (and their corresponding frequencies) are initially represented as the root vertex of separate trivial trees. Huffman's algorithm attempts to use these trees to build other binary trees, gradually merging the intermediate trees into one binary tree. The vertices representing the characters from our set will be leaves of the Huffman tree. The internal vertices of the tree will represent the sum of the frequencies of the leaves in the subtree rooted at that vertex.

For example, suppose we assign each of the characters of our message and its corresponding frequency to the root vertex of a trivial tree. From this collection of trees, select the two trees with roots having the smallest frequencies. In case of ties, randomly select the trees from those having the smallest frequencies. In this example the frequencies selected are 6 and 7. Make these two root vertices the left and right children of a new vertex, with this new vertex having frequency 13 (Figure 3.6.5). Return this new tree to our collection of trees.

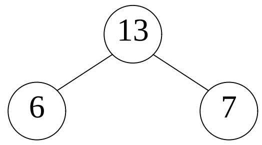


Figure 3.6.5. The first stage of Huffman's algorithm.

Again, we choose from the collection of trees the two trees with roots having the lowest frequencies, 10 and 12, and again form a larger tree by inserting a new vertex whose frequency is 22 and that has vertex 10 and vertex 12 as its left and right children, respectively. Again, return this tree to the collection. Repeating this process a third time, we select vertices 13 and 13. Following the construction, we obtain the tree of Figure 3.6.6.

We return this tree to the collection and again repeat this process. The next two roots selected have frequencies 18 and 22. We build the new tree and return it to the collection. Finally, only the roots 26 and 40 remain. We select them and build the tree shown in Figure 3.6.7. In addition to the tree we constructed, we also place a value of 0 on the edge from any vertex to its left child and a value of 1 on any edge from a vertex to its right child. Note that this choice is arbitrary and could easily be reversed.

We can read the code for each of the characters by following the unique path from the root 66 to the leaf representing the character of interest. The entire code is given in Table 3.6.1.

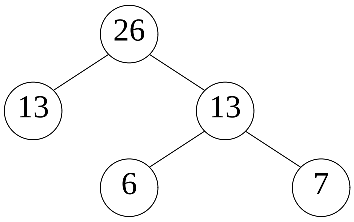


Figure 3.6.6. The new tree formed.

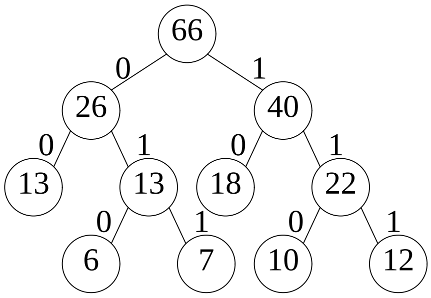


Figure 3.6.7. The final Huffman tree.

Next, suppose we are presented with an encoded message string, say, for example, a string like:

01011101100000101010110.

Assuming this encoded string was created from the Huffman tree of our example, we can decode this string by again using the Huffman tree. Beginning at the root, we use the next digit of the message to decide which edge of the tree we will follow. Initially, we follow the 0 edge from vertex 66 to vertex 26, then the 1 edge to vertex 13 and then the 0 edge to vertex 6. Since we are now at a leaf of the Huffman tree, the first character of the message is the letter *b*, as it corresponds to this leaf.

character	code
a	00
b	010
c	011
d	111
e	10
f	110

Table 3.6.1 The Huffman codes for the example character set.

Return to the root and repeat this process on the remaining digits of the message. The next three digits are 111, the code that corresponds to d, followed in turn by 011 (c), 00 (a), 00 (a), 010 (b), 010 (b), 10 (e) and 110 (f). Thus, the encoded message was bdcaabef.

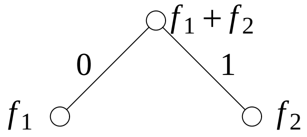
Algorithm 3.6.2 Construction of a Huffman Tree.

Input: Ordered frequencies $(f_1, f_2, \ldots, f_n)$ corresponding to the characters $(a_1, a_2, \ldots, a_n)$.

Output: A Huffman tree with leaves corresponding to the frequencies above.

Method: From a collection of trees, select the two root vertices corresponding to the smallest frequencies. Then insert a new vertex and make the two selected vertices the children of this new vertex. Return this tree to the collection of trees and repeat this process until only one tree remains.

1. If $n = 2$, then halt, thereby forming the tree:



2. If $n > 2$, then reorder the frequencies so that f_1 and f_2 are the two smallest frequencies. Let T_1 be the Huffman tree resulting from the algorithm being recursively applied to the frequencies $(f_1 + f_2, f_3, \ldots, f_n)$ and let T_2 be the Huffman tree that results from calling the algorithm recursively on the frequencies (f_1, f_2). Halt the algorithm with the tree that results from substituting T_2 for

some leaf of T_1 (which has value $f_1 + f_2$).

Note that the Algorithm does not produce a unique tree. If several frequencies are equal, their positions in the frequency list and, hence, their positions in the tree can vary. Can you find a different Huffman tree for the example data? What conditions would produce a unique Huffman tree?

Huffman trees are in a sense the optimal structure for binary encoding. That is, we would like to show that the Huffman code minimizes the length of encoded messages, with characters and frequencies matching those used in the construction of the Huffman tree. Our measure of the efficiency of the code is called the *weighted path length* of the coding tree and is defined to be: $\sum_{1 \le i \le n} f_i l_i$, where f_i is the frequency of the ith letter and l_i is the length of the path from the root in the Huffman tree to the vertex corresponding to the ith letter. The weighted path length is a reasonable measure to minimize since, when this value is divided by $\sum_{i=1}^{n} f_i$ (that is, the number of characters being encoded), we obtain the average length of the encoding per character.

Theorem 3.6.1 A Huffman tree for the frequencies $(f_1, f_2, \ldots, f_n)$ has minimum weighted path length among all full binary trees with leaves

$f_1, f_2, \ldots, f_n$.

Proof. We proceed by induction on *n*, the number of frequencies. If $n = 2$, the weighted path length of any full binary tree is $f_1 + f_2$, as is the value we obtain from the algorithm. Now, suppose that $n \ge 3$ and assume that the result follows for all Huffman trees with fewer than *n* leaves.

Reorder the frequencies so that $f_1 \le f_2 \le, \ldots, \le f_n$ (if necessary). Since there are only a finite number of full binary trees with *n* leaves, there must be one, call it *T*, with minimum weighted path length. Let *x* be an internal vertex of *T* whose distance from the root *r* is a maximum (for the internal vertices).

If f_1 and f_2 are not the frequencies of the leaves of *T* that are children of *x*, then we can exchange the frequencies of the children of *x*, say f_i and f_j, with f_1 and f_2 without increasing the weighted path length of *T*. This follows since $f_i \ge f_1$ and $f_j \ge f_2$ and this interchange moves f_i closer to the root and f_1 farther away from the root. But *T* has minimum weighted path length, and thus, its value cannot decrease. Hence, there must be a tree with minimum weighted path length that does have f_1 and f_2 as the frequencies of the children of an internal vertex that is a maximum distance from the root (again, this maximum is taken over internal vertices only).

Finally, it remains for us to show that this tree is minimal if, and only if, the tree that results from deleting the leaves f_1 and f_2 is also minimal for the frequencies that remain, namely $f_1 + f_2, f_3, \ldots, f_n$.

Note that the value at any internal vertex equals the sum of the frequencies of the leaves of the subtree rooted at that internal vertex. Thus, the weighted path length of a Huffman tree is the sum of the values of the internal vertices of the tree. If W_T is the weighted path length of T and W^* is the weighted path length of the graph $T^* = T - \{ f_1, f_2 \}$, then $W_T = W^* + f_1 + f_2$. But now we see that T has minimum weighted path length if, and only if, T^* does, since their weights differ by the constant $f_1 + f_2$. Thus, if either tree failed to be minimal, both trees would fail to be minimal. □

We can also verify that Huffman's algorithm assigns the shortest codes to the most frequently used characters.

Theorem 3.6.2 If $c_1, c_2, \ldots, c_n$ are the binary codes assigned by Huffman's algorithm to the characters with frequencies $f_1, f_2, \ldots, f_n$, respectively, and if $f_i < f_j$, then $length(c_i) \geq length(c_j)$.

Proof. Suppose instead that $length(c_j) > length(c_i)$. If we assign the code word c_i to the character with frequency f_j and c_j to the character with frequency f_i and leave all other code assignments the same, then we obtain a new code with minimum weighted path length less than the Huffman code. To see that this is the case, note that if W_H is the minimum weighted path length for the original Huffman tree and W^* is the new minimum weighted path length for the modified code, then

$$\begin{aligned} W_H - W^* &= (f_i \; length(c_i) + f_j \; length(c_j)) \\ &\quad - (f_i \; length(c_j) - f_j \; length(c_i)) \\ &= (f_j - f_i)(length(c_j) - length(c_i)) > 0, \end{aligned}$$

contradicting the fact that W_H is minimum. □

Can you verify that Huffman's algorithm has time complexity $0(n^2)$?

Section 3.7 More About Counting Trees – Using Generating Functions

In this section, our fundamental goal is to introduce one of the principal tools used to count combinatorial objects, namely generating functions.

To begin with, what is a generating function? Suppose we have a sequence of numbers (c_i); then the power series $\sum_{i=0}^{\infty} c_i x^i$ is called the *generating function* for the sequence. Examples from calculus and algebra demonstrate that a power series can be used to approximate a function of x. Our goal is to use the tools already developed for series to help us count objects. Our purpose is not to present a complete development of generating functions, but rather to show the reader already familiar with series how they can be used for counting.

Suppose we consider the question of the number of binary trees T possible on a set of n vertices. Let's call this number b_n. Clearly, if $n = 1$, $b_1 = 1$. For $n > 1$, we can select one vertex to be the root, and the remaining $n - 1$ vertices can be partitioned into those in the left and right subtrees of T. If there are j vertices in the left subtree and $n - 1 - j$ vertices in the right subtree, then the number of binary trees we can form on n vertices depends on the number of ways we can build the left and right subtrees; that is b_n depends on b_j and b_{n-1-j}. To determine exactly how many binary trees b_n there are, we must sum the products $b_j b_{n-1-j}$ for $j = 0, 1, \ldots, n - 1$. This gives us the following *recurrence relation:*

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0 .$$

We now illustrate how to solve this recurrence relation using generating functions.

Suppose our generating function $g(x) = \sum_{i=0}^{\infty} b_i x^i$. When we square $g(x)$, we obtain the following:

$$(g(x))^2 = g(x) \times g(x) = \sum_{n \geq 0} \Big(\sum_{0 \leq j \leq n} b_j \; b_{n-j} \Big) \, x^n$$

$$= \sum_{n \geq 0} (b_0 b_n + b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_n b_0) \, x^{n.}$$

But on careful examination we see that the coefficient of x^n in $g^2(x)$ is nothing but b_{n+1}. Hence, we obtain another relationship, namely

$$1 + x g^2(x) = g(x) .$$

But this equation is quadratic in $g(x)$, and so it yields the solution

$$g(x) = \frac{1 - \sqrt{1 - 4x}}{2x}.$$

Now, to obtain a series expansion for $g(x)$, we use the binomial generating function

$$(1 + z)^r = 1 + rz + r\left(\frac{r-1}{2}\right)z^2 + r\left(\frac{(r-1)(r-2)}{6}\right)z^3 + \cdots$$

We write the coefficients of this power series using the definition of generalized binomial coefficients, where for any real number r and integer k,

$$\binom{r}{k} = \frac{r(r-1)(r-2)\cdots(r-k+1)}{k!} \text{ ,for } k > 0,$$

while it has value zero if $k < 0$ and 1 when $k = 0$. Thus, we can rewrite the binomial generating function as

$$(1 + z)^r = \sum_{k \ge 0} \binom{r}{k} z^k.$$

Substituting this function in our expression for $g(x)$, we obtain

$$g(x) = \frac{1}{2x}\left(1 - \sum_{k \ge 0} \binom{1/2}{k} (-4x)^k\right).$$

Changing the dummy variable k to $n + 1$ and simplifying yields

$$\begin{aligned} g(x) &= \frac{1}{2x}\left(1 - \sum_{n+1 \ge 0} \binom{1/2}{n+1} (-4x)^{n+1}\right) \\ &= \frac{1}{2x} + \sum_{n+1 \ge 0} \binom{1/2}{n+1} (-1)^n 2^{2n+1} x^n \\ &= \sum_{n \ge 0} \binom{1/2}{n+1} (-1)^n 2^{2n+1} x^n. \end{aligned}$$

But this process yields coefficients of x^n which match b_n in our original definition of $g(x)$. Thus, we have that,

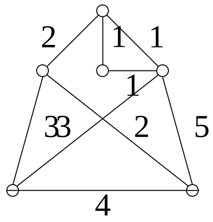
$$b_n = \binom{1/2}{n+1} (-1)^n 2^{2n+1}.$$

Using exercise 26, we can simplify this expression for b_n to obtain:

$$b_n = \frac{1}{n+1}\binom{2n}{n}.$$

Exercises

1. Show that if $T = (V, E)$ is a tree, then for any $e \in E$, $T - e$ has exactly two components.
2. Show that any connected graph on p vertices contains at least $p - 1$ edges.
3. Show that if T is a tree with $\Delta(T) \geq k$, then T has at least k leaves.
4. In a connected graph G, a vertex v is called *central* if $\max\limits_{u \in V(G)} d(u, v) = rad(G)$. Show that for a tree T, the set of central vertices consists of either one vertex or two adjacent vertices.
5. Show that the sequence $d_1, d_2, \ldots, d_p$ of positive integers is the degree sequence of a tree if, and only if, the graph is connected and $\sum\limits_{i=1}^{p} d_i = 2(p - 1)$.
6. Show that the number of end vertices in a nontrivial tree of order n equals $2 + \sum\limits_{deg\ v_i \geq 3} (deg\ v_i - 2)$.
7. Determine the time complexity of Kruskal's algorithm.
8. What happens to the time complexity of Kruskal's Algorithm if we do not presort the edges in nondecreasing order of weight?
9. Apply Kruskal's algorithm to the graph:



10. Apply Prim's algorithm to the graph of the previous problem.

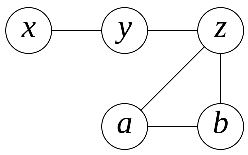
11. Prove that a graph G is acyclic if, and only if, every induced subgraph of G contains a vertex of degree one at most.

12. Characterize those graphs with the property that every connected subgraph is also an induced subgraph.

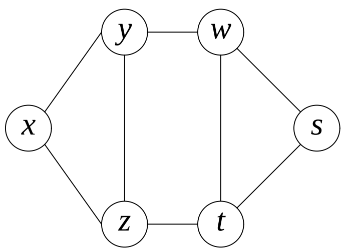
13. Find the binary tree representations for the expressions $4x - 2y$, $(3x + z)(xy - 7z)$, and $\sqrt{b^2 - 4ac}$.

14. Perform a preorder, postorder and inorder traversal on the trees constructed in the previous problem.

15. Determine the number of nonidentical spanning trees of the graph below. Before you begin your computation, make an observation about this graph that will simplify the calculations.



16. (a) Find the number of nonidentical spanning trees of the graph



(b) Determine the number of spanning trees in the above graph using D_G, the directed graph obtained from G.

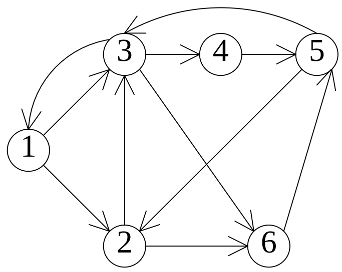
17. Find the spanning trees on the set { 1, 2, 3, 4 }.

18. Using the proof of Cayley's theorem, determine the sequences of length $p - 2$ on $\{ 1, 2, 3, 4 \}$ that correspond to any two of the trees found in the previous problem.

19. Show that the Prüfer algorithm for creating a tree from a sequence selects the same vertex as the algorithm for producing the sequence.

20. Use the matrix-tree theorem to prove Cayley's tree formula.

21. Find the number of directed spanning trees with root 3 in the following digraph:



How many are rooted at vertex 5? How many are rooted at vertex 2?

22. Given a graph G with adjacency matrix A and degree matrix C, show that the matrices $C - A$ and $A_i(D_G)$ are equal.

23. Show that the number of trees with m labeled edges and no labels on the vertices is $(m + 1)^{m-2}$.

24. Determine the number of trees that can be built on p labeled vertices such that one specified vertex is of degree k.

25. By *contracting an edge* $e = uv$, we mean removing e and identifying the vertices u and v as a single new vertex. Let $num_T(\ G\ )$ denote the number of spanning trees of the graph G. Show that the following recursive formula holds:

$$num_T(G) = num_T(G - e) + num_T(G \ o\ e)$$

where $G\ o\ e$ means the graph obtained from G by contracting the edge e. Hint: Interpret what $num_T(G - e)$ and $num_T(G\ o\ e)$ really count.

26. Show that the algorithm for determining the number of spanning trees of G implied by the previous problem takes exponential time.

27. Determine the Huffman tree and code for the alphabet $\{ x, y, z, a, b, c \}$ with corresponding frequencies $(3, 8, 1, 5, 4, 4)$.

28. What is the minimum weighted path length for the Huffman tree you constructed in the previous problem?

29. Using the definition of $\binom{1/2}{n+1}$, show that

$$\binom{1/2}{n+1} = \frac{(-1)^n}{2^{n+1}} \times \frac{(1)(3)(5)\cdots(2n-1)}{(1)(2)(3)\cdots(n+1)}.$$

Also show that $(1)(3)(5)\cdots(2n-1) = \frac{(2n)!}{2^n n!}$ and use these two facts to obtain the formula for b_n.

References

1. Cayley, A., A Theorem on Trees. *Quart. J. Math.,* 23(1889), 376 – 378.
2. Edmonds, J., Optimum Branchings. *J. of Res. of the Nat. Bureau of Standards,* 71B(1967), 233 – 240.
3. Huffman, D. A., A Method for the Construction of Minimum Redundancy Codes. *Proc. IRE,* No. 10, 40(1952), 1098 – 1101.
4. Karp, R. M., A Simple Derivation of Edmonds Algorithm for Optimum Branchings. *Networks,* 1(1972), 265 – 272.
5. Kirchhoff, G., Über die Aufflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Ströme geführt wird. *Ann. Phy. Chem.,* 72(1847), 497 – 508.
6. Kruskal, J. B. Jr., On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. Amer. Math. Soc.,* 7(1956), 48 – 50.
7. Ore, O., *Theory of Graphs.* Amer. Math. Soc. Publ., Providence (1962).
8. Prim, R. C., Shortest Connection Networks and Some Generalizations. *Bell System Tech. J.,* 36(1957), 1389 – 1401.
9. Prüfer, H., Neuer Beweis eines satzes über Permutationen. *Arch. Math. Phys.,* 27(1918), 742 – 744.
10. Standish, T. A., *Data Structure Techniques.* Addison-Wesley Pub. Co., Reading, Mass. (1980).
11. Tutte, W. T., The Dissection of Equilateral Triangles into Equilateral Triangles. *Proc. Cambridge Phil. Soc.,* 44(1948), 463 – 482.