4.4 Spanning Trees and Rooted Trees

Spanning Trees

We introduced trees with the example of choosing a minimum-sized set of edges that would connect all the vertices in the graph of Figure 4.1. That led us to discuss trees. In fact the kinds of trees that solve our original problem have a special name. A tree whose edge set is a subset of the edge set of the graph G is called a *spanning tree* of G if the tree has exactly the same vertex set as G. Thus the graphs (a) and (b) of Figure 4.3 are spanning trees of the graph of Figure 4.1.

Exercise 4.4-1 Does every connected graph have a spanning tree? Either give a proof or a counter-example.

Exercise 4.4-2 Give an algorithm that determines whether a graph has a spanning tree, and finds such a tree if it exists, that takes time bounded above by a polynomial in v and e, where v is the number of vertices, and e is the number of edges.

For Exercise 4.4-1, if the graph has no cycles but is connected, it is a tree, and thus is its own spanning tree. This makes a good base step for a proof by induction on the number of cycles of the graph that every connected graph has a spanning tree. Suppose inductively that when $c > 0$ and a connected graph has fewer than c cycles, then the graph has a spanning tree. Suppose that G is a graph with c cycles. Choose a cycle of G and choose an edge of that cycle. Deleting that edge (but not its endpoints) reduces the number of cycles by at least one, and so our inductive hypothesis implies that the resulting graph has a spanning tree. But then that spanning tree is also a spanning tree of G. Therefore by the principle of mathematical induction, every finite connected graph has a spanning tree. We have proved the following theorem.

Theorem 4.11 *Each finite connected graph has a spanning tree.*

Proof: The proof is given before the statement of the theorem.■

In Exercise 4.4-2, we want an algorithm for determining whether a graph has a spanning tree. One natural approach would be to convert the inductive proof of Theorem 4.11 into a recursive algorithm. Doing it in the obvious way, however, would mean that we would have to search for cycles in our graph. A natural way to look for a cycle is to look at each subset of the vertex set and see if that subset is a cycle of the graph. Since there are 2^v subsets of the vertex set, we could not guarantee that an algorithm that works in this way would find a spanning tree in time which is big Oh of a polynomial in v and e. In an algorithms course you will learn a much faster (and much more sophisticated) way to implement this approach. We will use another approach, describing a quite general algorithm which we can then specialize in several different ways for different purposes.

The idea of the algorithm is to build up, one vertex at a time, a tree that is a subgraph (not necessarily an induced subgraph) of the graph $G = (V, E)$. (A subgraph of G that is a tree is called a *subtree* of G.) We start with some vertex, say x_0. If there are no edges leaving the vertex and the graph has more than one vertex, we know the graph is not connected and we therefore don't have a spanning tree. Otherwise, we can choose an edge ϵ_1 that connects x_0 to

another vertex x_1. Thus $\{x_0, x_1\}$ is the vertex set of a subtree of G. Now if there are no edges that connect some vertex in the set $\{x_1, x_2\}$ to a vertex not in that set, $\{x_1, x_2\}$ is a connected component of G. In this case, either G is not connected and has no spanning tree, or it just has two vertices and we have a spanning tree. However if there is an edge that connects some vertex in the set $\{x_1, x_2\}$ to a vertex not in that set, we can use this edge to continue building a tree. This suggests an inductive approach to building up the vertex set S of a subtree of our graph one vertex at a time. For the base case of the algorithm, we let $S = \{x_0\}$. For the inductive step, given S, we choose an edge ϵ that leads from a vertex in S to a vertex in $V - S$ and add it to the edge set E' of the subtree if such an edge exists. If no such edge exists, we stop. If $V = S$ when we stop then E' is the edge set of a spanning tree. (We can prove inductively that E' is the edge set of a tree on S, because adding a vertex of degree one to a tree gives a tree.) If $V \neq S$ when we stop, G is not connected and does not have a spanning tree.

To describe the algorithm a bit more precisely, we give pseudocode.

```
Spantree(V,E)
// Assume that V is the vertex set of the graph.
// Assume that E is an array with |V| entries, and entry i of E is the set of
//edges incident with the vertex in position i of V.
(1)  i = 0;
(2)  Choose a vertex x_0 in V.
(3)  S = {x_0}
(4)  While there is an edge from a vertex in S to a vertex not in S
(5)       i=i+1
(6)       Choose an edge ε_i from a vertex y in S to a vertex x_i not in S
(7)       S = S ∪ {x_i}
(8)       E' = E' ∪ ε_i
(9)  If i = |V| - 1
(10)      return E'
(11) Else
(12)      Print "The graph is not connected."
```

The way in which the vertex x_i and the edge ϵ_i are chosen was deliberately left vague because there are several different ways to specify x_i and ϵ_i that accomplish several different purposes. However, with some natural assumptions, we can still give a big Oh bound on how long the algorithm takes. Presumably we will need to consider at most all v vertices of the graph in order to choose x_i, and so assuming we decide whether or not to use a vertex in constant time, this step of the algorithm will take $O(v)$ time. Presumably we will need to consider at most all e edges of our graph in order to choose ϵ_i, and so assuming we decide whether or not to use an edge in constant time, this step of the algorithm takes at most $O(e)$ time. Given the generality of the condition of the while loop that begins in line 4, determining whether that condition is true might also take $O(e)$ time. Since we repeat the While loop at most v times, all executions of the While loop should take at most $O(ve)$ time. Since line 9 requires us to compute $|V|$, it takes $O(v)$ steps, and all the other lines take constant time. Thus, with the assumptions we have made, the algorithm takes $O(ve + v + e) = O(ve)$ time.

Breadth First Search

Notice that algorithm Spantree will continue as long as a vertex in S is connected to a vertex not in S. Thus when it stops, S will be the vertex set of a connected component of the graph and E' will be the edge set of a spanning tree of this connected component. This suggests that one use that we might make of algorithm Spantree is to find connected components of graphs. If we want the connected component containing a specific vertex x, then we make this choice of x_0 in Line 2. Suppose this is our goal for the algorithm, and suppose that we also want to make the algorithm run as quickly as possible. We could guarantee a faster running time if we could arrange our choice of ϵ_i so that we examined each edge no more than some constant number of times between the start and the end of the algorithm. One way to achieve this is to first use all edges incident with x_0 as ϵ_is, then consider all edges incident with x_1, using them as ϵ_i if we can, and so on.

We can describe this process inductively. We begin by putting vertex x_0 in S and (except for loops or multiple edges) all edges incident with x_0 in E'. Then given vertices 0 through i, all of whose edges we have examined and either accepted or (permanently) rejected as an ϵ_j, we examine the edges leaving vertex $i+1$. For each of these edges that is incident with a vertex not already in S, we add the edge and that vertex to the tree. Otherwise we reject that edge. Eventually we reach a point where we have examined all the edges leaving all the vertices in S, and we stop.

To give a pseudocode description of the algorithm, we assume that we are given an array V that contains the names of the vertices. There are a number of ways to keep track of the edge set of a graph in a computer. One way is to give a list, called an *adjacency list*, for each vertex listing all vertices adjacent to it. In the case of a multigraph, we list each adjacency as many times as there are edges that give the adjacency. In our pseudocode we implement this idea with the array E that gives in position i a list of all locations in the array V of vertices adjacent in G to vertex $V[i]$.

In our pseudocode we also use an array "Edge" to list the edges of the set we called E' in algorithm Spantree, an array "Vertex" to list the positions in V of the vertices in the set S in the algorithm Spantree, an array "Vertexname" to keep track of the names of the vertices we add to the set S, and an array "Intree" to keep track of whether the vertex in position i of V is in S. Because we want our pseudocode to be easily translatable into a computer language, we avoid subscripts, and use x to stand for the place in the array V that holds the name of the vertex where we are to start the search, i.e. the vertex x_0.

```
BFSpantree(x,V,E)
// Assume that V is an array with v entries, the names of the vertices,
// and that x is the location in V of the name of the vertex with which we want
// to start the tree.
//Assume that E is an array with v entries, each a list of the positions
// in V of the names of vertices adjacent to the corresponding entry of V.
(1)  i = 0 ; k = 0 ; Intree[x] = 1; Vertex[0] = x; Vertexname[0] = V[x]
(2)  While i <= k
(3)       i=i+1
(4)       For each j in the list E[Vertex[i]]
(5)            If Intree[j] ≠ 1
(6)                 k = k + 1
```

```
(7)                    Edge[k] = {V[Vertex[i]], V[j]}
(8)                    Intree[j] = 1
(9)                    Vertex[k] = j
(10)                   Vertexname[k] = V[j].
(11) Print "Connected component"
(12) return Vertexname[0 : k]
(13) print "Spanning tree edges of connected component"
(14) return Edge[1 : k]
```

Notice that the pseudocode allows us to deal with loops and multiple edges through the test whether vertex j is in the tree in Line 5. However the primary purpose of this line is to make sure that we do not examine edges that point from vertex i back to a vertex that is already in the tree.

This algorithm requires that we execute the "For" loop that starts in Line 4 once for each edge incident with vertex i. The "While" loop that starts in Line 2 is executed at most once for each vertex. Thus we execute the "For" loop at most twice for each edge, and carry out the other steps of the "While" loop at most once for each vertex, so that the time to carry out this algorithm is $O(V + E)$.

The algorithm carries out what is known as a "breadth first search"[5] of the graph centered at $V[x]$. The reason for the phrase "breadth first" is because each time we start to work on a new vertex, we examine all its edges (thus exploring the graph broadly at this point) before going on to another vertex. As a result, we first add all vertices at distance 1 from $V[x]$ to S, then all vertices at distance 2 and so on. When we choose a vertex $V[\text{Vertex}[k]]$ to put into the set S in Line 9, we are effectively labelling it as vertex k. We call k the *breadth first number* of the vertex $V[j]$ and denote it as $BFN(V[j])$[6]. The breadth first number of a vertex arises twice in the breadth first search algorithm. The breadth first search number of a vertex is assigned to that vertex when it is added to the tree, and (see Problem **??**) is the number of vertices that have been previously added. But it then determines when a vertex of the tree is used to add other vertices to the tree: the vertices are taken in order of their breadth first number for the purpose of examining all incident edges to see which ones allow us to add new vertices, and thus new edges, to the tree.

This leads us to one more description of breadth first search. We create a breadth first search tree centered at x_0 in the following way. We put the vertex x_0 in the tree and give it breadth first number zero. Then we process the vertices in the tree in the order of their breadth first number as follows: We consider each edge leaving the vertex. If it is incident with a vertex z not in the tree, we put the edge into the edge set of the tree, we put z into the vertex set of the tree, and we assign az a breadth first number one more than that of the vertex most recently added to the tree. We continue in this way until all vertices in the tree have been processed.

We can use the idea of breadth first number to make our remark about the distances of vertices from x_0 more precise.

Lemma 4.12 *After a breadth first search of a graph G centered at $V[x]$, if $d(V[x], V[z]) > d(V[x], V[y])$, then $BFN(V[z]) > BFN(V[y])$.*

[5]This terminology is due to Robert Tarjan who introduced the idea in his PhD thesis.

[6]In words, we say that the breadth first number of a vertex is k if it is the kth vertex added to a breadth-first search tree, counting the initial vertex x as the zeroth vertex added to the tree

Proof: We will prove this in a way that mirrors our algorithm. We shall show by induction that for each nonnegative k, all vertices of distance k from x_0 are added to the spanning tree (that is, assigned a breadth first number and put into the set S) after all vertices of distance $k-1$ and before any vertices of distance $k+1$. When $k=1$ this follows because S starts as the set $V[x]$ and all vertices adjacent to $V[x]$ are next added to the tree before any other vertices. Now assume that $n>1$ and all vertices of distance n from $V[x]$ are added to the tree after all vertices of distance $n-1$ from $V[x]$ and before any vertices of distance $n+1$. Suppose some vertex of distance n added to the tree has breadth first number m. Then when i reaches m in Line 3 of our pseudocode we examine edges leaving vertex $V[\text{Vertex}[m]]$ in the "For loop." Since, by the inductive hypothesis, all vertices of distance $n-1$ or less from $V[x]$ are added to the tree before vertex $V[\text{Vertex}[m]]$, when we examine vertices $V[j]$ adjacent to vertex $V[\text{Vertex}[m]]$, we will have $\text{Intree}[j]=1$ for these vertices. Since each vertex of distance n from $V[x]$ is adjacent to some vertex $V[z]$ of distance $n-1$ from $V[x]$, and $\text{BFN}[V[z]]<m$ (by the inductive hypothesis), any vertex of distance n from $V[x]$ and adjacent to vertex $V[\text{Vertex}[m]]$ will have $\text{Intree}[j]=1$. Since any vertex adjacent to vertex $V[\text{Vertex}[m]]$ is of distance at most $n+1$ from $V[x]$, every vertex we add to the tree from vertex $V[\text{Vertex}[m]]$ will have distance $n+1$ from the tree. Thus every vertex added to the tree from a vertex of distance n from $V[x]$ will have distance $n+1$ from $V[x]$. Further, all vertices of distance $n+1$ are adjacent to some vertex of distance n from $V[x]$, so each vertex of distance $n+1$ is added to the tree from a vertex of distance n. Note that no vertices of distance $n+2$ from vertex $V[x]$ are added to the tree from vertices of distance n from vertex $V[x]$. Note also that all vertices of distance $n+1$ are added to the tree from vertices of distance $n-1$ from vertex $V[x]$. Therefore all vertices with distance $n+1$ from $V[x]$ are added to the tree after all edges of distance n from $V[x]$ and before any edges of distance $n+2$ from $V[x]$. Therefore by the principle of mathematical induction, for every positive integer k, all vertices of distance k from $V[x]$ are added to the tree before any vertices of distance $k+1$ from vertex $V[x]$ and after all vertices of distance $k-1$ from vertex $V[x]$. Therefore since the breadth first number of a vertex is the number of the stage of the algorithm in which it was added to the tree, if $d(V[x],V[z])>d(V[x],V[y])$, then $\text{BFN}(V[z])>\text{BFN}(V[y])$. ■

Although we introduced breadth first search for the purpose of having an algorithm that quickly determines a spanning tree of a graph or a spanning tree of the connected component of a graph containing a given vertex, the algorithm does more for us.

Exercise 4.4-3 How does the distance from $V[x]$ to $V[y]$ in a breadth first search centered at $V[x]$ in a graph G relate to the distance from $V[x]$ to $V[y]$ in G?

In fact the unique path from $V[x]$ to $V[y]$ in a breadth first search spanning tree of a graph G is a shortest path in G, so the distance from $V[x]$ to another vertex in G is the same as their distance in a breadth first search spanning tree centered at $V[x]$. This makes it easy to compute the distance between a vertex $V[x]$ and all other vertices in a graph.

Theorem 4.13 *The unique path from $V[x]$ in a breadth first search spanning tree centered at the vertex $V[x]$ of a graph G to a vertex $V[y]$ is a shortest path from $V[x]$ to $V[y]$ in G.*

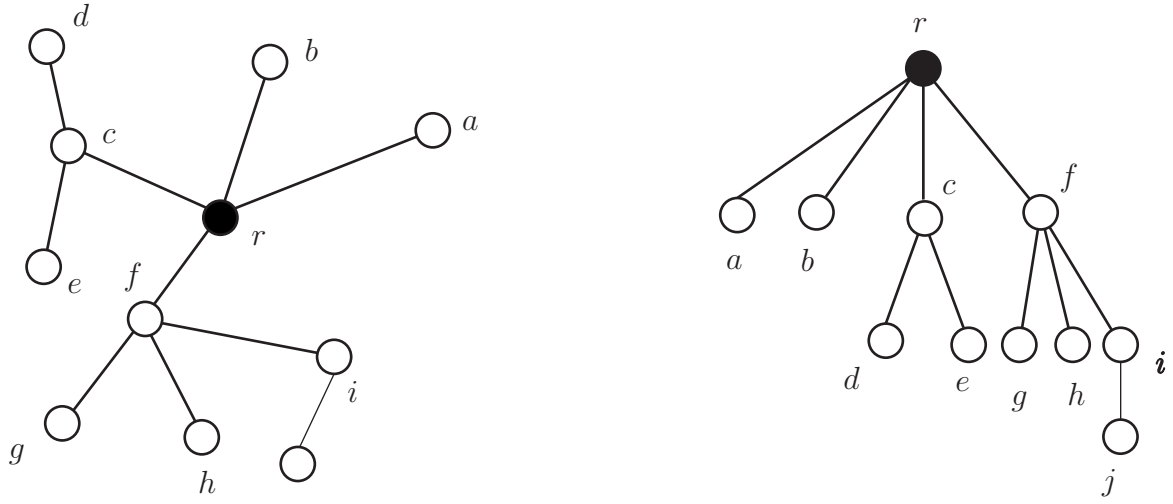
Proof: We prove the theorem by induction on the distance from $V[x]$ to $V[y]$. Fix a breadth first search tree of G centered at $V[x]$. If the distance is 0, then the single vertex $V[x]$ is a shortest path from $V[x]$ to $V[x]$ in G and the unique path in the tree. Assume that $k>0$ and that when distance from $V[x]$ to $V[y]$ is less than k, the path from $V[x]$ to $V[y]$ in the tree is a shortest

path from $V[x]$ to $V[y]$ in G. Now suppose that the distance from $V[x]$ to $V[y]$ is k. Suppose that a shortest path from $V[x]$ to $V[y]$ has $V[z]$ and $V[y]$ as its last two vertices. Suppose that the unique path from $V[x]$ to $V[y]$ in the tree has $V[z']$ and $V[y]$ as its last two vertices. Then $\text{BFN}(V[z']) < \text{BFN}(V[z])$, because otherwise we would have added $V[y]$ to the tree from vertex $V[z]$. Then by the contrapositive of Lemma 4.12, the distance from $V[x]$ to $V[z']$ is less than or equal to that from $V[x]$ to $V[z]$. But then by the inductive hypothesis, the distance from $V[x]$ to $V[z']$ is the length of the unique path in the tree, and by our previous comment is less than or equal to the distance from $V[x]$ to $V[z]$. However then the length of the unique path from $V[x]$ to $V[y]$ in the tree is no more than the distance from $V[x]$ to $V[y]$, so the two are equal. By the principle of mathematical induction, the distance from $V[x]$ to $V[y]$ is the length of the unique path in the tree for every vertex y of the graph. ■

Rooted Trees

A breadth first search spanning tree of a graph is not simply a tree, but a tree with a selected vertex, namely $V[x]$. It is one example of what we call a rooted tree. A *rooted tree* consists of a tree with a selected vertex, called a *root*, in the tree. Another kind of rooted tree you have likely seen is a binary search tree. It is fascinating how much additional structure is provided to a tree when we select a vertex and call it a root. In Figure 4.10 we show a tree with a chosen vertex and the result of redrawing the tree in a more standard way, with the root at the top and all the edges sloping down, as you might expect to see with a family tree.

Figure 4.10: Two different views of the same rooted tree.



We adopt the language of family trees—ancestor, descendant, parent, and child—to describe rooted trees in general. In Figure 4.10, we say that vertex j is a child of vertex i, and a descendant of vertex r as well as a descendant of vertices f and i. We say vertex f is an ancestor of vertex i. Vertex r is the parent of vertices a, b, c, and f. Each of those four vertices is a child of vertex r. Vertex r is an ancestor of all the other vertices in the tree. In general, in a rooted tree with

root r, a vertex x is an *ancestor* of a vertex y, and vertex y is a *descendant* of vertex x if x and y are different and x is on the unique path from the root to y. Vertex x is a *parent* of vertex y and y is a *child* of vertex x in a rooted tree if x is the unique vertex adjacent to y on the unique path from r to y. A vertex can have only one parent, but many ancestors. A vertex with no children is called a *leaf* vertex or an *external vertex*; other vertices are called *internal vertices*.

Exercise 4.4-4 Prove that a vertex in a rooted tree can have at most one parent. Does every vertex in a rooted tree have a parent?

In Exercise 4.4-4, suppose x is not the root. Then, because there is a unique path between a vertex x and the root of a rooted tree and there is a unique vertex on that path adjacent to x, each vertex other than the root has a unique parent. The root, however, has no parent.

Exercise 4.4-5 A binary tree is a special kind of rooted tree that has some additional structure that makes it tremendously useful as a data structure. In order to describe the idea of a binary tree it is useful to think of a tree with no vertices, which we call the null tree or empty tree. Then we can recursively describe a *binary tree* as

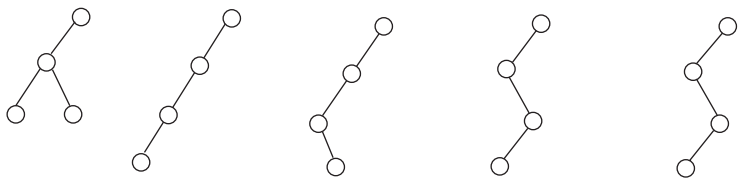
- an empty tree, or
- a structure consisting of a root vertex, a binary tree called the left subtree of the root and a disjoint binary tree called the right subtree of the root, with an edge connecting the root of the left subtree to the root vertex and an edge connecting the root of the right subtree to the root vertex.

Then a single vertex is a binary tree with an empty right subtree and an empty left subtree. A rooted tree with two vertices can occur in two ways as a binary tree, either with a root and a left subtree consisting of one vertex or as a root and a right subtree consisting of one vertex. Draw all binary trees on four vertices in which the root node has an empty right child. Draw all binary trees on four vertices in which the root has a nonempty left child and a nonempty right child.

Exercise 4.4-6 A binary tree is a *full* binary tree if each vertex has either two nonempty children or two empty children (a vertex with two empty children is called a *leaf*.) Are there any full binary trees on an even number of vertices? Prove that what you say is correct.

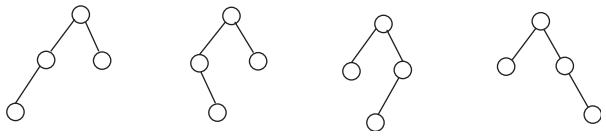
For Exercise 4.4-5 we have five binary trees shown in Figure 4.11 for the first question. Then

Figure 4.11: The four-vertex binary trees whose root has an empty right child.



in Figure 4.12 we have four more trees as the answer to the second question.

Figure 4.12: The four-vertex binary trees whose root has both a left and a right child.



For Exercise 4.4-6, it is possible to have a full binary tree with zero vertices, so there is one such binary tree. But, if a full binary tree is not empty, it must have odd number of vertices. We can prove this inductively. A full binary tree with 1 vertex has an odd number of vertices. Now suppose inductively that $n > 1$ and any full binary tree with fewer than n vertices has an odd number of vertices. For a full binary tree with $n > 1$ vertices, the root must have two nonempty children. Thus removing the root gives us two binary trees, rooted at the children of the original root, each with fewer than n vertices. By the definition of full, each of the subtrees rooted in the two children must be full binary tree. The number of vertices of the original tree is one more than the total number of vertices of these two trees. This is a sum of three odd numbers, so it must be odd. Thus, by the principle of mathematical induction, if a full binary tree is not empty, it must have odd number of vertices.

The definition we gave of a binary tree was a inductive one, because the inductive definition makes it easy for us to prove things about binary trees. We remove the root, apply the inductive hypothesis to the binary tree or trees that result, and then use that information to prove our result for the original tree. We could have defined a binary tree as a special kind of rooted tree, such that

- each vertex has at most two children,
- each child is specified to be a left or right child, and
- a vertex has at most one of each kind of child.

While it works, this definition is less convenient than the inductive definition.

There is a similar inductive definition of a rooted tree. Since we have already defined rooted trees, we will call the object we are defining an r-tree. The inductive definition states that an r-tree is either a single vertex, called a root, or a graph consisting of a vertex called a root and a set of disjoint r-trees, each of which has its root attached by an edge to the original root. We can then prove as a theorem that a graph is an r-tree if and only if it is a rooted tree. Sometimes inductive proofs for rooted trees are easier if we use the method of removing the root and applying the inductive hypothesis to the rooted trees that result, as we did for binary trees in our solution of Exercise 4.4-6.

Important Concepts, Formulas, and Theorems

1. *Spanning Tree.* A tree whose edge set is a subset of the edge set of the graph G is called a *spanning tree* of G if the tree has exactly the same vertex set as G.
2. *Breadth First Search.* We create a *breadth first search* tree centered at x_0 in the following way. We put the vertex x_0 in the tree and give it breadth first number zero. Then we

process the vertices in the tree in the order of their breadth first number as follows: We consider each edge leaving the vertex. If it is incident with a vertex z not in the tree, we put the edge into the edge set of the tree, we put z into the vertex set of the tree, and we assign z a breadth first number one more than that of the vertex most recently added to the tree. We continue in this way until all vertices in the tree have been processed.

3. *Breadth first number.* The *breadth first number* of a vertex in a breadth first search tree is the number of vertices that were already in the tree when the vertex was added to the vertex set of the tree.

4. *Breadth first search and distances.* The distance from a vertex y to a vertex x may be computed by doing a breadth first search centered at x and then computing the distance from y to x in the breadth first search tree. In particular, the path from x to y in a breadth first search tree of G centered at x is a shortest path from x to y in G.

5. *Rooted tree.* A *rooted tree* consists of a tree with a selected vertex, called a *root*, in the tree.

6. *Ancestor, Descendant.* In a rooted tree with root r, a vertex x is an *ancestor* of a vertex y, and vertex is a *descendant* of vertex x if x and y are different and x is on the unique path from the root to y.

7. *Parent, Child.* In a rooted tree with root r, vertex x is a *parent* of vertex y and y is a *child* of vertex x in if x is the unique vertex adjacent to y on the unique path from r to y.

8. *Leaf (External) Vertex.* A vertex with no children in a rooted tree is called a *leaf* vertex or an *external vertex*.

9. *Internal Vertex* A vertex of a rooted tree that is not a leaf vertex is called an *internal* vertex..

10. *Binary Tree* We recursively describe a *binary tree* as

    - an empty tree (a tree with no vertices), or
    - a structure consisting of a root vertex, a binary tree called the left subtree of the root and a binary tree called the right subtree of the root, with an edge connecting the root of the left subtree to the root vertex and an edge connecting the root of the right subtree to the root vertex.

11. *Full Binary Tree* A binary tree is a *full* binary tree if each vertex has either two nonempty children or two empty children.

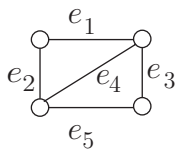
Problems

1. Find all spanning trees (list their edge sets) of the graph in Figure 4.13.

2. Show that a finite graph is connected if and only if it has a spanning tree.

3. Draw all rooted trees on 5 vertices. The order and the place in which you write the vertices down on the page is unimportant. If you would like to label the vertices (as we did in the graph in Figure 4.10), that is fine, but don't give two different ways of labelling or drawing the same tree.

Figure 4.13: A graph.



4. Draw all rooted trees on 6 vertices with four leaf vertices. If you would like to label the vertices (as we did in the graph in Figure 4.10), that is fine, but don't give two different ways of labelling or drawing the same tree.

5. Find a tree with more than one vertex that has the property that all the rooted trees you get by picking different vertices as roots are different as rooted trees. (Two rooted trees are the same (isomorphic), if they each have one vertex or if you can label them so that they have the same (labelled) root and the same (labelled) subtrees.)

6. Create a breadth first search tree centered at vertex 12 for the graph in Figure 4.8 and use it to compute the distance of each vertex from vertex 12. Give the breadth first number for each vertex.

7. It may seem clear to some people that the breadth first number of a vertex is the number of vertices previously added to the tree. However the breadth first number was not actually defined in this way. Give a proof that the breadth first number of a vertex is the number of vertices previously added to the tree.

8. A *(left, right) child* of a vertex in a binary tree is the root of a (left, right) subtree of that vertex. A binary tree is a *full* binary tree if each vertex has either two nonempty children or two empty children (a vertex with two empty children is called a *leaf*.) Draw all full binary trees on seven vertices.

9. The *depth* of a node in a rooted tree is defined to be the number of edges on the (unique) path to the root. A binary tree is *complete* if it is full (see Problem 8) and all its leaves have the same depth. How many vertices does a complete binary tree of depth 1 have? Depth 2? Depth d? (Proof required for depth d.)

10. The *height* of a rooted or binary tree with one vertex is 0; otherwise it is 1 plus the maximum of the heights of its subtrees. Based on Exercise 4.4-9, what is the minimum height of *any* binary tree on n vertices? (Please prove this.)

11. A binary tree is complete if it is full and all its leaves have the same depth (see Exercise 4.4-8 and Exercise 4.4-9). A vertex that is not a leaf vertex is called an *internal* vertex. What is the relationship between the number I of internal vertices and the number L of leaf vertices in a complete binary tree. A full binary tree? (Proof please.)

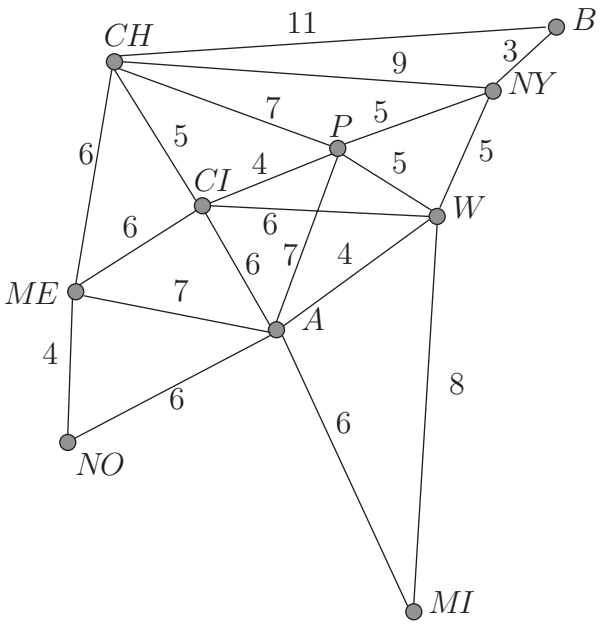
12. The *internal path length* of a binary tree is the sum, taken over all internal (see Exercise 4.4-11) vertices of the tree, of the depth of the vertex. The *external path length* of a binary tree is the sum, taken over all leaf vertices of the tree, of the depth of the vertex. Show that in a full binary tree with n internal vertices, internal path length i and external path length e, we have $e = i + 2n$.

13. Prove that a graph is an r-tree, as defined at the end of the section if and only if it is a rooted tree.

14. Use the inductive definition of a rooted tree (r-tree) given at the end of the section to prove once again that a rooted tree with n vertices has $n - 1$ edges if $n \geq 1$.

15. In Figure 4.14 we have added numbers to the edges of the graph of Figure 4.1 to give what is usually called a *weighted graph*—the name for a graph with numbers, often called *weights* associated with its edges. We use $w(\epsilon)$ to stand for the weight of the edge ϵ. These numbers represent the lease fees in thousands of dollars for the communication lines the edges represent. Since the company is choosing a spanning tree from the graph to save money, it is natural that it would want to choose the spanning tree with minimum total cost. To be precise, a *minimum spanning tree* in a weighted graph is a spanning tree of the graph such that the sum of the weights on the edges of the spanning tree is a minimum among all spanning trees of the graph.

Figure 4.14: A stylized map of some eastern US cities.



Give an algorithm to select a spanning tree of minimum total weight from a weighted graph and apply it to find a minimum spanning tree of the weighted graph in Figure 4.14. Show that your algorithm works and analyze how much time it takes.