Chapter 6

Graphs

6.1 Graphs

In this chapter we introduce a fundamental structural idea of discrete mathematics, that of a graph. Many situations in the applications of discrete mathematics may be modeled by the use of a graph, and many algorithms have their most natural description in terms of graphs. It is for this reason that graphs are important to the computer scientist. Graph theory is an ideal subject for developing a deeper understanding of proof by induction because induction, especially strong induction, seems to enter into the majority of proofs in graph theory.

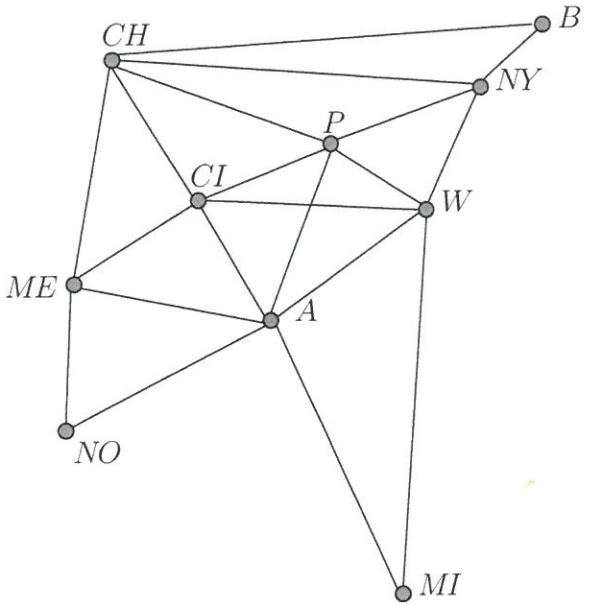
Exercise 6.1-1 In Figure 6.1, you see a stylized map of some cities in the eastern United States (Boston, New York, Pittsburgh, Cincinnati, Chicago, Memphis, New Orleans, Atlanta, Washington DC, and Miami). A company has major offices with data processing centers in each of these cities, and as its operations have grown, it has leased dedicated communication lines between certain pairs of these cities to allow for efficient communication among the computer systems in the various cities. Each grey dot in the figure stands for a data center, and each line in the figure stands for a dedicated communication link. What is the minimum number of links that could be used in sending a message from B (Boston) to NO (New Orleans)? Give a route with this number of links.

Exercise 6.1-2 Which city or cities has or have the most communication links emanating from them?

Exercise 6.1-3 What is the total number of communication links in the figure?

The picture in Figure 6.1 is a drawing of what we call a "graph". A **graph** consists of a set of *vertices* and a set of *edges* with the property that each edge has two (not necessarily different) vertices associated with it and called its *endpoints*. We say the edge *joins* the endpoints, and we say two endpoints are *adjacent* if they are joined by an edge. When a vertex is an endpoint of an edge, we say the edge and the vertex are *incident*. Several more examples of graphs are given in Figure 6.2. To *draw* a graph, we draw a point (in our case a grey circle) in the plane for each vertex, and then for each edge we draw a (possibly curved) line between the points that correspond to the endpoints of the edge. The only vertices that may be touched by the line

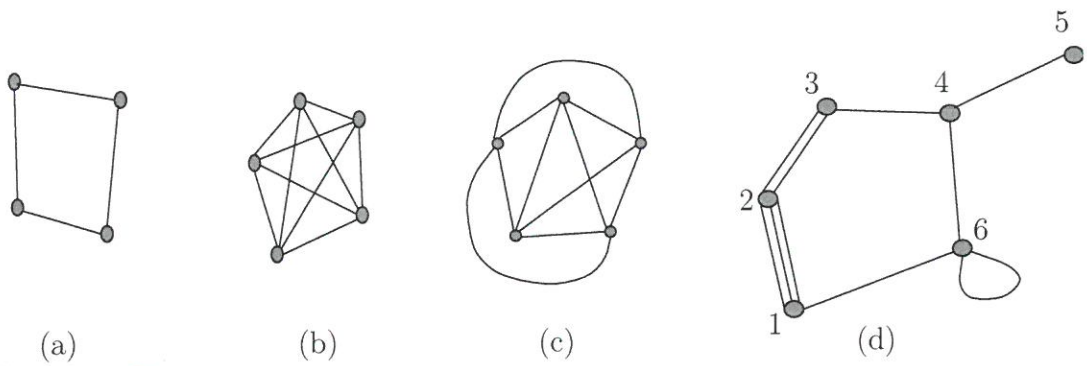
Figure 6.1: A stylized map of some eastern US cities.



representing an edge are the endpoints of the edge. Notice that in graph (d) of Figure 6.2 we have three edges joining the vertices marked 1 and 2 and two edges joining the vertices marked 2 and 3. We also have one edge that joins the vertex marked 6 to itself. This edge has two identical endpoints. The graph in Figure 6.1 and the first three graphs in Figure 6.2 are called simple graphs. A *simple graph* is one that has at most one edge joining each pair of distinct vertices, and no edges joining a vertex to itself.[1] You'll note in Figure 6.2 that we sometimes label the vertices of the graph and we sometimes don't. We label the vertices when we want to give them meaning,

[1]The terminology of graph theory has not yet been standardized, because it is a relatively young subject. The terminology we are using here is the most popular terminology in computer science, but some graph theorists would reserve the word graph for what we have just called a simple graph and would use the word multigraph for what we called a graph.

Figure 6.2: Some examples of graphs



as in Figure 6.1 or when we know we will want to refer to them as in graph (d) of Figure 6.2. We say that graph (d) in Figure 6.2 has a "loop" at vertex 6 and multiple edges joining vertices 1 and 2 and vertices 2 and 3. More precisely, an edge that joins a vertex to itself is called a *loop* and we say we have *multiple edges* between vertices x and y if there is more than one edge joining x and y. If there is an edge from vertex x to vertex y in a simple graph, we denote it by $\{x, y\}$. Thus $\{P, W\}$ denotes the edge between Pittsburgh and Washington in Figure 6.1 Sometimes it will be helpful to have a symbol to stand for a graph. We use the phrase "Let $G = (V, E)$" as a shorthand for "Let G stand for a graph with vertex set V and edge set E."

The drawings in parts (b) and (c) of Figure 6.2 are different drawings of the same graph. The graph consists of five vertices and one edge between each pair of distinct vertices. It is called the complete graph on five vertices and is denoted by K_5. In general, a *complete graph* on n vertices is a graph with n vertices that has an edge between each two of the vertices. We use K_n to stand for a complete graph on n vertices. These two drawings are intended to illustrate that there are many different ways we can draw a given graph. The two drawings illustrate two different ideas. Drawing (b) illustrates the fact that each vertex is adjacent to each other vertex and suggests that there is a high degree of symmetry. Drawing (c) illustrates the fact that it is possible to draw the graph so that only one pair of edges crosses; other than that the only places where edges come together are at their endpoints. In fact, it is impossible to draw K_5 so that no edges cross, a fact that we shall explain later in this chapter.

In Exercise 6.1-1 the links referred to are edges of the graph and the cities are the vertices of the graph. It is possible to get from the vertex for Boston to the vertex for New Orleans by using three communication links, namely the edge from Boston to Chicago, the edge from Chicago to Memphis, and the edge from Memphis to New Orleans. A **path** in a graph is an alternating sequence of vertices and edges such that

- it starts and ends with a vertex, and
- each edge joins the vertex before it in the sequence to the vertex after it in the sequence.[2]

If a is the first vertex in the path and b is the last vertex in the path, then we say the path is a path from a to b. Thus the path we found from Boston to New Orleans is $B\{B, CH\}CH\{CH, ME\}, ME\{ME, NO\}NO$ Because the graph is simple, we can also use the shorter notation B, CH, ME, NO to describe the same path, because there is exactly one edge between successive vertices in this list. The *length* of a path is the number of edges it has, so our path from Boston to New Orleans has length 3. The length of a shortest path between two vertices in a graph is called the *distance* between them. Thus the distance from Boston to New Orleans in the graph of Figure 6.1 is three. By inspecting the map we see that there is no shorter path from Boston to New Orleans. Notice that no vertex or edge is repeated on our path from Boston to New Orleans. A path is called a **simple path** if it has no repeated vertices or edges.[3]

The degree of a vertex

In Exercise 6.1-2, the city with the most communication links is Atlanta (A). We say the vertex A has "degree" 6 because 6 edges emanate from it. More generally the *degree* of a vertex in a

[2]Again, the terminology we are using here is the most popular terminology in computer science, but what we just defined as a path would be called a walk by most graph theorists.

[3]Most graph theorists reserve the word path for what we are calling a simple path, but again we are using the language most popular in computer science.

graph is the number of times it is incident with edges of the graph; that is, the degree of a vertex x is the number of edges from x to other vertices plus twice the number of loops at vertex x. In graph (d) of Figure 6.2 vertex 2 has degree 5, and vertex 6 has degree 4. In a graph like the one in Figure 6.1, it is somewhat difficult to count the edges just because you can forget which ones you've counted and which ones you haven't.

Exercise 6.1-4 Is there a relationship between the number of edges in a graph and the degrees of the vertices? If so, find it. Hint: computing degrees of vertices and number of edges in some relatively small examples of graphs should help you discover a formula. To find one proof, imagine a wild west movie in which the villain is hiding under the front porch of a cabin. A posse rides up and is talking to the owner of the cabin, and the bad guy can just barely look out from underneath the porch and count the horses hoofs. If he counts the hooves accurately, what can he do to figure out the number of horses, and thus presumably the size of the posse?

In Exercise 6.1-4, examples such as those in Figure 6.2 convince us that the sum of the degrees of the vertices is twice the number of edges. How can we prove this? One way is to count the total number of incidences between vertices and edges (similar to counting the horses hooves in the hint). Each edge has exactly two incidences, so the total number of incidences is twice the number of edges. But the degree of a vertex is the number of incidences it has, so the sum of the degrees of the vertices is also the total number of of incidences. Therefore the sum of the degrees of the vertices of a graph is twice the number of edges. Thus to compute the number of edges of a graph, we can sum the degrees of the vertices and divide by two. (In the case of the hint, the horses correspond to edges and the hooves to endpoints.) There is another proof of this result that uses induction.

Theorem 6.1 *Suppose a graph has a finite number of edges. Then the sum of the degrees of the vertices is twice the number of edges.*

Proof: We induct on the number of edges of the graph. If a graph has no edges, then each vertex has degree zero and the sum of the degrees is zero, which is twice the number of edges. Now suppose $e > 0$ and the theorem is true whenever a graph has fewer than e edges. Let G be a graph with e edges and let ϵ be an edge of G.[4] Let G' be the graph (on the same vertex set as G) we get by deleting ϵ from the edge set E of G. Then G has $e-1$ edges, and so by our inductive hypothesis, the sum of the degrees of the vertices of G' is twice $e-1$. Now there are two possible cases. Either e was a loop, in which case one vertex of G' has degree two less in G' than it has in G. Otherwise e has two distinct endpoints, in which case exactly two vertices of G' have degree one less than their degree in G. Thus in both cases the sum of the degrees of the vertices in G' is two less than the sum of the degrees of the vertices in G, so the sum of the degrees of the vertices in G is $(2e-2)+2 = 2e$. Thus the truth of the theorem for graphs with $e-1$ edges implies the truth of the theorem for graphs with e edges. Therefore, by the principle of mathematical induction, the theorem is true for a graph with any finite number of edges. ■

There are a couple instructive points in the proof of the theorem. First, since it wasn't clear from the outset whether we would need to use strong or weak induction, we made the inductive

[4]Since it is very handy to have e stand for the number of edges of a graph, we will use Greek letters such as epsilon (ϵ) to stand for edges of a graph. It is also handy to use v to stand for the number of vertices of a graph, so we use other letters near the end of the alphabet, such as w, x, y,and z to stand for vertices.

hypothesis we would normally make for strong induction. However in the course of the proof, we saw that we only needed to use weak induction, so that is how we wrote our conclusion. This is not a mistake, because we used our inductive hypothesis correctly. We just didn't need to use it for every possible value it covered.

Second, instead of saying that we would take a graph with $e-1$ edges and add an edge to get a graph with e edges, we said that we would take a graph with e edges and remove an edge to get a graph with $e-1$ edges. This is because we need to prove that the result holds for *every* graph with e edges. By using the second approach we avoided the need to say that "every graph with e edges may be built up from a graph with $e-1$ edges by adding an edge," because in the second approach we started with an arbitrary graph on e edges. In the first approach, we would have proved that the theorem was true for all graphs that could be built from an $e-1$ edge graph by adding an edge, and we would have had to explicitly say that every graph with e edges could be built in this way.

In Exercise 3 the sum of the degrees of the vertices is (working from left to right)

$$2+4+5+6+5+2+5+4+2=40,$$

and so the graph has 20 edges.

Connectivity

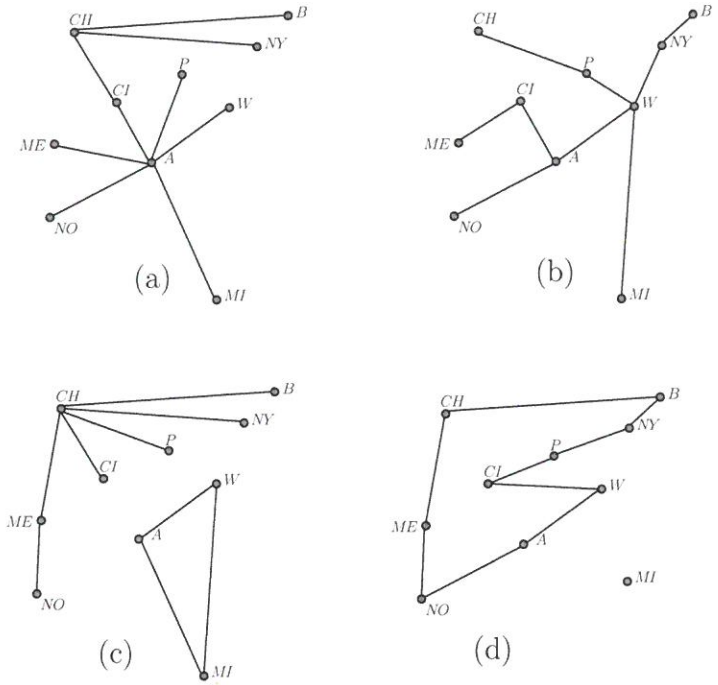
All of the examples we have seen so far have a property that is not common to all graphs, namely that there is a path from every vertex to every other vertex.

Exercise 6.1-5 The company with the computer network in Figure 6.1 needs to reduce its expenses. It is currently leasing each of the communication lines shown in the Figure. Since it can send information from one city to another through one or more intermediate cities, it decides to only lease the minimum number of communication lines it needs to be able to send a message from any city to any other city by using any number of intermediate cities. What is the minimum number of lines it needs to lease? Give two examples of subsets of the edge set with this number of edges that will allow communication between any two cities and two examples of a subset of the edge set with this number of edges that will not allow communication between any two cities.

Some experimentation with the graph convinces us that if we keep eight or fewer edges, there is no way we can communicate among the cities (we will explain this more precisely later on), but that there are quite a few sets of nine edges that suffice for communication among all the cities. In Figure 6.3 we show two sets of nine edges each that allow us to communicate among all the cities and two sets of nine edges that do not allow us to communicate among all the cities.

Notice that in graphs (a) and (b) it is possible to get from any vertex to any other vertex by a path. A graph is called *connected* there is a path between each two vertices of the graph. Notice that in graph (c) it is not possible to find a path from Atlanta to Boston, for example, and in graph (d) it is not possible to find a path from Miami to any of the other vertices. Thus these graphs are not connected; we call them disconnected. In graph (d) we say that Miami is an *isolated vertex.*

Figure 6.3: Selecting nine edges from the stylized map of some eastern US cities.



We say two vertices are *connected* if there is a path between them, so a graph is connected if each two of its vertices are connected. Thus in Graph (c) the vertices for Boston and New Orleans are connected. The relationship of being connected is an equivalence relation (in the sense of Section 1.4). To show this we would have to show that this relationship divides the set of vertices up into mutually exclusive classes; that is, that it partitions the vertices of the graph. The class containing Boston, for example is all vertices connected to Boston. If two vertices are in that set, they both have paths to Boston, so there is a path between them using Boston as an intermediate vertex. If a vertex x is in the set containing Boston and another vertex y is not, then they cannot be connected or else the path from y to x and then on to Boston would connect y to Boston, which would mean y was in the class containing Boston after all. Thus the relation of being connected partitions the vertex set of the graph into disjoint classes, so it is an equivalence relation. Though we made this argument with respect to the vertex Boston in the specific case of graph (c) of Figure 6.3, it is a perfectly general argument that applies to arbitrary vertices in arbitrary graphs. We call the equivalence relation of "being connected to" the *connectivity* relation. There can be no edge of a graph between two vertices in different equivalence classes of the connectivity relation because then everything in one class would be connected to everything in the other class, so the two classes would have to be the same. Thus we also end up with a partition of the edges into disjoint sets. If a graph has edge set E, and C is an equivalence class of the connectivity relation, then we use $E(C)$ to denote the set of edges whose endpoints are both in C. Since no edge connects vertices in different equivalence classes, each edge must be in some set $E(C)$. The graph consisting of an equivalence class C of the connectivity relation together with the edges $E(C)$ is called a *connected component* of our original graph. From now on our emphasis will be on connected components rather than on equivalence classes of the connectivity relation. Notice that graphs (c) and (d) of Figure 6.3 each have two connected components. In

graph (c) the vertex sets of the connected components are $\{NO, ME, CH, CI, P, NY, B\}$ and $\{A, W, MI\}$. In graph (d) the connected components are $\{NO, ME, CH, B, NY, P, CI, W, A\}$ and $\{MI\}$. Two other examples of graphs with multiple connected components are shown in Figure 6.4.

Figure 6.4: A simple graph G with three connected components and a graph H with four connected components.

G H

Cycles

In graphs (c) and (d) of Figure 6.3 we see a feature that we don't see in graphs (a) and (b), namely a path that leads from a vertex back to itself. A path that starts and ends at the same vertex is called a *closed path.* A closed path with at least one edge is called a *cycle* if, except for the last vertex, all of its vertices are different. The closed paths we see in graphs (c) and (d) of Figure 6.3 are cycles. Not only do we say that $\{NO, ME, CH, B, NY, P, CI, W, A, NO\}$ is a cycle in in graph (d) of Figure 6.3, but we also say it is a cycle in the graph of Figure 6.1. The way we distinguish between these situations is to say the cycle $\{NO, ME, CH, B, NY, P, CI, W, A, NO\}$ is an induced cycle in Figure 6.3 but not in Figure 6.1. More generally, a graph H is called a *subgraph* of the graph G if all the vertices and edges of H are vertices and edges of G. We call H an *induced subgraph* of G if every vertex of H is a vertex of G, and every edge of G connecting vertices of H is an edge of H. Thus the first graph of Figure 6.4 has an induced K_4 and an induced cycle on three vertices.

We don't normally distinguish which point on a cycle really is the starting point; for example we consider the cycle $\{A, W, MI, A\}$ to be the same as the cycle $\{W, MI, A, W\}$. Notice that there are cycles with one edge and cycles with two edges in the second graph of Figure 6.4. We call a graph G a *cycle* on n vertices or an n-cycle and denote it by C_n if it has a cycle that contains all the vertices and edges of G and a *path* on n vertices and denote it by P_n if it has a path that contains all the vertices and edges of G. Thus drawing (a) of Figure 6.2 is a drawing of C_4. The second graph of Figure 6.4 has an induced P_3 and an induced C_2 as subgraphs.

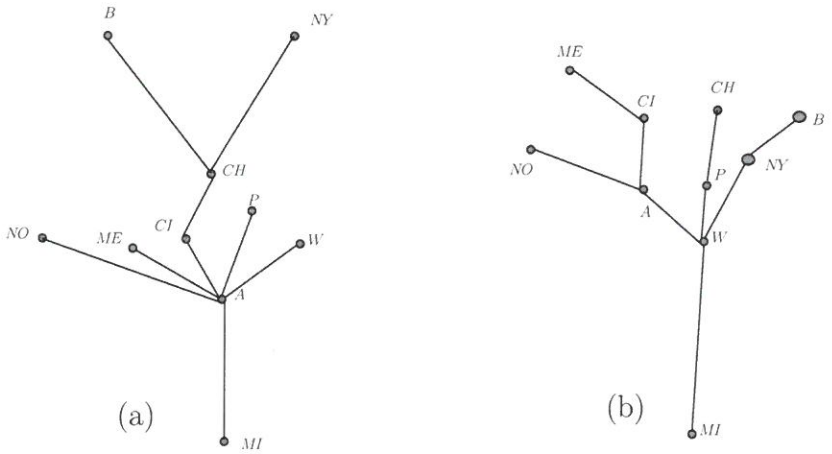
Trees

C_n P_n

The graphs in parts (a) and (b) of Figure 6.3 are called trees. We have redrawn them slightly in Figure 6.5 so that you can see why they are called trees. We've said these two graphs are called trees, but we haven't given a definition of trees. In the examples in Figure 6.3, the graphs we have called trees are connected and have no cycles.

Definition 6.1 *A connected graph with no cycles is called a tree.*

Figure 6.5: A visual explanation of the name tree.



Other Properties of Trees

In coming to our definition of a tree, we left out a lot of other properties of trees we could have discovered by a further analysis of Figure 6.3

Exercise 6.1-6 Given two vertices in a tree, how many distinct simple paths can we find between the two vertices?

Exercise 6.1-7 Is it possible to delete an edge from a tree and have it remain connected?

Exercise 6.1-8 If $G = (V, E)$ is a graph and we add an edge that joins vertices of V, what can happen to the number of connected components?

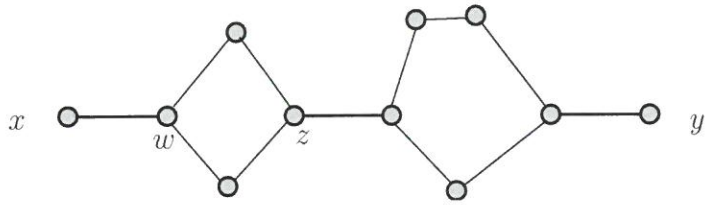
Exercise 6.1-9 How many edges does a tree with v vertices have?

Exercise 6.1-10 Does every tree have a vertex of degree 1? If the answer is yes, explain why. If the answer is no, try to find additional conditions that will guarantee that a tree satisfying these conditions has a vertex of degree 1.

For Exercise 6.1-6, suppose we had two distinct paths from a vertex x to a vertex y. They begin with the same vertex x and might have some more edges in common as in Figure 6.6. Let w be the last vertex after (or including) x the paths share before they become different. The paths must come together again at y, but they might come together earlier. Let z be the first vertex the paths have in common after w. Then there are two paths from w to z that have only w and z in common. Taking one of these paths from w to z and the other from z to w gives us a cycle, and so the graph is not a tree. We have shown that if a graph has two distinct paths from x to y, then it is not a tree. By contrapositive inference, then, if a graph is a tree, it does not have two distinct paths between two vertices x and y. We state this result as a theorem.

Theorem 6.2 *There is exactly one path between each two vertices in a tree.*

Figure 6.6: A graph with multiple paths from x to y.



Proof: By the definition of a tree, there is at least one path between each two vertices. By our argument above, there is at most one path between each two vertices. Thus there is exactly one path. ■

For Exercise 6.1-7, note that if ϵ is an edge from x to y, then x, ϵ, y is the unique path from x to y in the tree. Suppose we delete ϵ from the edge set of the tree. If there were still a path from x to y in the resulting graph, it would also be a path from x to y in the tree, which would contradict Theorem 6.2. Thus the only possibility is that there is no path between x and y in the resulting graph, so it is not connected and is therefore not a tree.

For Exercise 6.1-8, if the endpoints are in the same connected component, then the number of connected components won't change. If the endpoints of the edge are in different connected components, then the number of connected components can go down by one. Since an edge has two endpoints, it is impossible for the number of connected components to go down by more than one when we add an edge. This paragraph and the previous one lead us to the following useful lemma.

Lemma 6.3 *Removing one edge from the edge set of a tree gives a graph with two connected components, each of which is a tree.*

Proof: Suppose as before the lemma that ϵ is an edge from x to y. We have seen that the graph G we get by deleting ϵ from the edge set of the tree is not connected, so it has at least two connected components. But adding the edge back in can only reduce the number of connected compponents by one. Therefore G has exactly two connected components. Since neither has any cycles, both are trees. ■

In Exercise 6.1-9, our trees with ten vertices had nine edges. If we draw a tree on two vertices it will have one edge; if we draw a tree on three vertices it will have two edges. There are two different looking trees on four vertices as shown in Figure 6.7, and each has three edges. On the

Figure 6.7: Two trees on four vertices.

basis of these examples we conjecture that a tree on n vertices has $n - 1$ edges. One approach to proving this is to try to use induction. To do so, we have to see how to build up every tree from smaller trees or how to take a tree and break it into smaller ones. Then in either case we

have to figure out how use the truth of our conjecture for the smaller trees to imply its truth for the larger trees. A mistake that people often make at this stage is to assume that every tree can be built from smaller ones by adding a vertex of degree 1. While that is true for finite trees with more than one vertex (which is the point of Exercise 6.1-10), we haven't proved it yet, so we can't yet use it in proofs of other theorems. Another approach to using induction is to ask whether there is a natural way to break a tree into two smaller trees. There is: we just showed in Lemma 6.3 that if you remove an edge ϵ from the edge set of a tree, you get two connected components that are trees. We may assume inductively that the number of edges of each of these trees is one less than its number of vertices. Thus if the graph with these two connected components has v vertices, then it has $v-2$ edges. Adding ϵ back in gives us a graph with $v-1$ edges, so except for the fact that we have not done a base case, we have proved the following theorem.

Theorem 6.4 *For all integers $v \geq 1$, a tree with v vertices has $v-1$ edges.*

Proof: If a tree has one vertex, it can have no edges, for any edge would have to connect that vertex to itself and would thus give a cycle. A tree with two or more vertices must have an edge in order to be connected. We have shown before the statement of the theorem how to use the deletion of an edge to complete an inductive proof that a tree with v vertices has $v-1$ edges, and so for all $v \geq 1$, a tree with v vertices has $v-1$ edges. ■

Finally, for Exercise 6.1-10 we can now give a contrapositive argument to show that a finite tree with more than one vertex has a vertex of degree one. Suppose instead that G is a graph that is connected and all vertices of G have degree two or more. Then the sum of the degrees of the vertices is at least 2v, and so by Theorem 6.1 the number of edges is at least v. Therefore by Theorem 6.4 G is not a tree. Then by contrapositive inference, if T is a tree, then T must have at least one vertex of degree one. This corollary to Theorem 6.4 is so useful that we state it formally.

Corollary 6.5 *A finite tree with more than one vertex has at least one vertex of degree one.*

Important Concepts, Formulas, and Theorems

1. *Graph.* A *graph* consists of a set of *vertices* and a set of *edges* with the property that each edge has two (not necessarily different) vertices associated with it and called its *endpoints.*

2. *Edge; Adjacent.* We say an edge in a graph *joins* its endpoints, and we say two endpoints are *adjacent* if they are joined by an edge.

3. *Incident.* When a vertex is an endpoint of an edge, we say the edge and the vertex are *incident.*

4. *Drawing of a Graph.* To *draw* a graph, we draw a point in the plane for each vertex, and then for each edge we draw a (possibly curved) line between the points that correspond to the endpoints of the edge. Lines that correspond to edges may only touch the vertices that are their endpoints.

5. *Simple Graph.* A *simple graph* is one that has at most one edge joining each pair of distinct vertices, and no edges joining a vertex to itself.

6. *Length, Distance.* The *length* of a path is the number of edges. The *distance* between two vertices in a graph is the length of a shortest path between them.

7. *Loop; Multiple Edges.* An edge that joins a vertex to itself is called a loop and we say we have multiple edges between vertices x and y if there is more than one edge joining x and y.

8. *Notation for a Graph.* We use the phrase "Let $G = (V, E)$" as a shorthand for "Let G stand for a graph with vertex set V and edge set E."

9. *Notation for Edges.* In a simple graph we use the notation $\{x, y\}$ for an edge from x to y. In any graph, when we want to use a letter to denote an edge we use a Greek letter like ϵ so that we can save e to stand for the number of edges of the graph.

10. *Complete Graph on n vertices.* A *complete graph* on n vertices is a graph with n vertices that has an edge between each two of the vertices. We use K_n to stand for a complete graph on n vertices.

11. *Path.* We call an alternating sequence of vertices and edges in a graph a *path* if it starts and ends with a vertex, and each edge joins the vertex before it in the sequence to the vertex after it in the sequence.

12. *Simple Path.* A path is called a *simple path* if it has no repeated vertices or edges.

13. *Degree of a Vertex.* The *degree* of a vertex in a graph is the number of times it is incident with edges of the graph; that is, the degree of a vertex x is the number of edges from x to other vertices plus twice the number of loops at vertex x.

14. *Sum of Degrees of Vertices.* The sum of the degrees of the vertices in a graph with a finite number of edges is twice the number of edges.

15. *Connected.* A graph is called *connected* if there is a path between each two vertices of the graph. We say two vertices are *connected* if there is a path between them, so a graph is connected if each two of its vertices are connected. The relationship of being connected is an equivalence relation on the vertices of a graph.

16. *Connected Component.* If C is a subset of the vertex set of a graph, we use $E(C)$ to stand for the set of all edges *both* of whose endpoints are in C. The graph consisting of an equivalence class C of the connectivity relation together with the edges $E(C)$ is called a *connected component* of our original graph.

17. *Closed Path.* A path that starts and ends at the same vertex is called a *closed path.*

18. *Cycle.* A closed path with at least one edge is called a *cycle* if, except for the last vertex, all of its vertices are different.

19. *Tree.* A connected graph with no cycles is called a tree.

20. *Important Properties of Trees.*

    (a) There is a unique path between each two vertices in a tree.

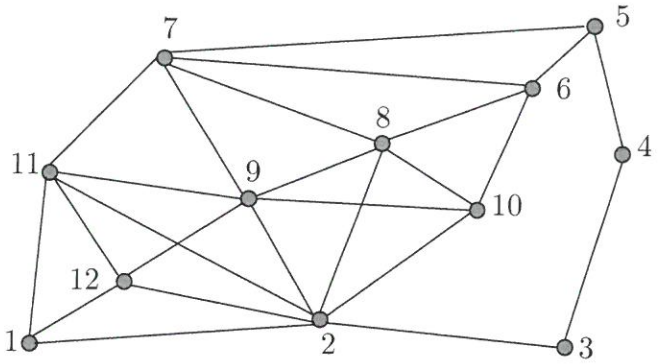
    (b) A tree on v vertices has $v - 1$ edges.

    (c) Every finite tree with at least two vertices has a vertex of degree one.

Problems

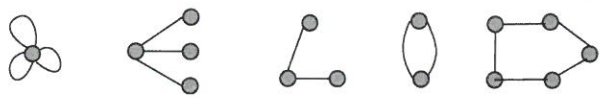
1. Find the shortest path you can from vertex 1 to vertex 5 in Figure 6.8.

Figure 6.8: A graph.



2. Find the longest simple path you can from vertex 1 to vertex 5 in Figure 6.8.
3. Find the vertex of largest degree in Figure 6.8. What is it's degree?

Figure 6.9: A graph with a number of connected components.



4. How many connected components does the graph in Figure 6.9 have?
5. Find all induced cycles in the graph of Figure 6.9.
6. What is the size of the largest induced K_n in Figure 6.9?
7. Find the largest induced K_n (in words, the largest complete subgraph) you can in Figure 6.8.
8. Find the size of the largest induced P_n in the graph in Figure 6.9.
9. A graph with no cycles is called a *forest*. Show that if a forest has v vertices, e edges, and c connected components, then $v = e + c$.
10. What can you say about a five vertex simple graph in which every vertex has degree four?
11. Find a drawing of K_6 in which only three pairs of edges cross.
12. Either prove true or find a counter-example. A graph is a tree if there is one and only one simple path between each pair of vertices.
13. Is there some number m such that if a graph with v vertices is connected and has m edges, then it is a tree? If so, what is m in terms of v?

14. Is there some number m such that a graph on n vertices is a tree if and only if it has m edges and has no cycles.

15. Suppose that a graph G is connected, but for each edge, deleting that edge leaves a disconnected graph. What can you say about G? Prove it.

16. Show that each tree with four vertices can be drawn with one of the two drawings in Figure 6.7.

17. Draw the minimum number of drawings of trees you can so that each tree with five vertices has one of those drawings. Explain why you have drawn all possible trees.

18. Draw the minimum number of drawings of trees you can so that each tree with six vertices is represented by exactly one of those drawings. Explaining why you have drawn all possible drawings is optional.

19. Find the longest induced cycle you can in Figure 6.8.

6.2 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 6.1. 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 6.3 are spanning trees of the graph of Figure 6.1.

Exercise 6.2-1 Does every connected graph have a spanning tree? Either give a proof or a counter-example.

Exercise 6.2-2 Give an algorithm that determines whether a graph has a spanning tree, finds such a tree if it exists, and 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 6.2-1, 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. Let $c > 0$ and suppose inductively that when 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 6.6 *Each finite connected graph has a spanning tree.*

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

In Exercise 6.2-2, 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 6.6 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 $\{x_0, x_1\}$ to a vertex not in that set, then $\{x_0, x_1\}$ 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 $\{x_0, x_1\}$ 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 $V - S$ (provided such an edge exists) and add it to the edge set E' of the subtree. 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 an array that lists 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 choosing a vertex x_0 and putting vertex x_0 in S and (except for loops or multiple edges) all edges incident with x_0 in E'. As we put edges into E', we number them, starting with ϵ_1. This creates a list $\epsilon_1, \epsilon_2, \ldots$ of edges. When we add edge ϵ_i to the tree, one of its two vertices is not yet numbered. We number it as x_i. Then given vertices 0 through i, all of whose incident edges we have examined and either accepted or (permanently) rejected as a member of E' (or more symbolically, 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, numbering the edges and vertices as described above. 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 multiple edges, we list each adjacency as many times as there are edges that give the adjacency. In our pseudocode we implement the idea of an adjacency list 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 7) 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 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.

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

[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

Lemma 6.7 *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($V[y]$).*

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[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[Vertex[m]], when we examine vertices $V[j]$ adjacent to vertex V[Vertex[m]], we will have 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 BFN[$V[z]$] $< m$ (by the inductive hypothesis), any vertex of distance n from $V[x]$ and adjacent to vertex V[Vertex[m]] will have Intree[j] = 1. Since any vertex adjacent to vertex V[Vertex[m]] is of distance at most $n+1$ from $V[x]$, every vertex we add to the tree from vertex V[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 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 BFN($V[z]$) >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 6.2-3 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 6.8 *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.*

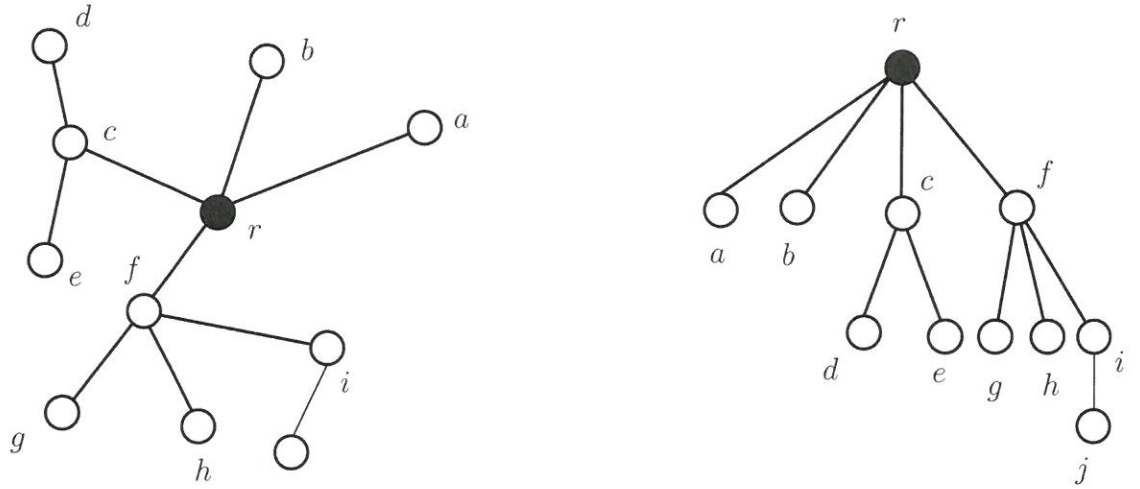
Proof: We prove the theorem by induction on the distance from $V[x]$ to $V[y]$. Fix a breadth first search tree of G centered at $V[x]$. If the distance is 0, then the single vertex $V[x]$ is a shortest

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 6.7, 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 6.10 we show a tree with a chosen vertex and the result of redrawing the tree in a more standard way. The standard way computer scientists draw rooted trees is with the root at the top and all the edges sloping down, as you might expect to see with a family tree.

Figure 6.10: 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 6.10, 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 6.2-4 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 6.2-4, 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 6.2-5 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 (a tree with no vertices), or
- a structure T 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. If the left or right subtree is nonempty, its root node is joined by an edge to the root of T.

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 6.2-6 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 6.2-5 we have five binary trees shown in Figure 6.11 for the first question. Then

Figure 6.11: The four-vertex binary trees whose root has an empty right child.

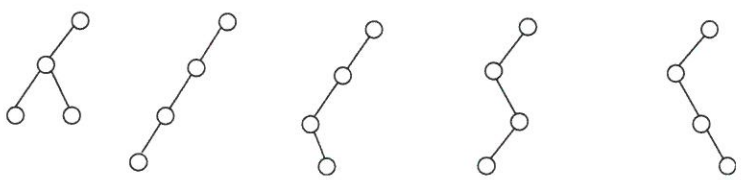
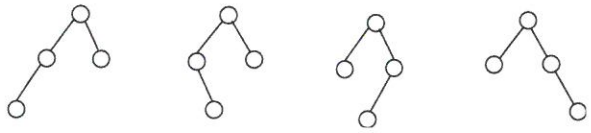


Figure 6.12: The four-vertex binary trees whose root has both a left and a right child.



in Figure 6.12 we have four more trees as the answer to the second question.

For Exercise 6.2-6, 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 an 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 an 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 recursive 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 6.2-6.

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 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.

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.

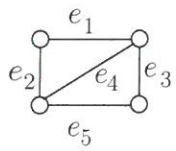
10. *Binary Tree.* We recursively describe a *binary tree* as

    - an empty tree (a tree with no vertices), or
    - a structure T 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. If the left or right subtree is nonempty, its root node is joined by an edge to the root of T.

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

12. *Recursive Definition of a Rooted Tree.* The recursive definition of a rooted tree states that it is either a single vertex, called a root, or a graph consisting of a vertex called a root and a set of disjoint rooted trees, each of which has its root attached by an edge to the original root.

Figure 6.13: A graph.



Problems

1. Find all spanning trees (list their edge sets) of the graph in Figure 6.13.

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 6.10), that is fine, but don't give two different ways of labelling or drawing the same tree.

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 6.10), 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 6.8 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 6.2-9, what is the minimum height of *any* binary tree on n vertices? (Please prove this.)