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.

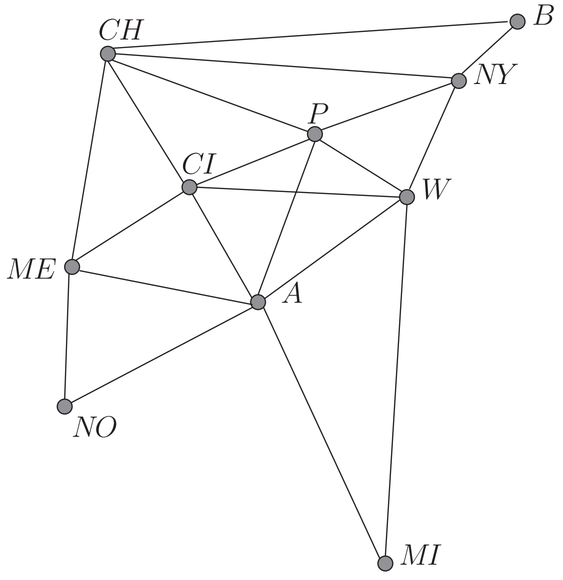
Exercise 6.1-1 In Figure 6.1, you see a stylized map of some cities in the eastern United States (Boston, New York, Pittsburgh, Cincinnati, Chicago, Memphis, New Orleans, Atlanta, Washington DC, and Miami). A company has major offices with data processing centers in each of these cities, and as its operations 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

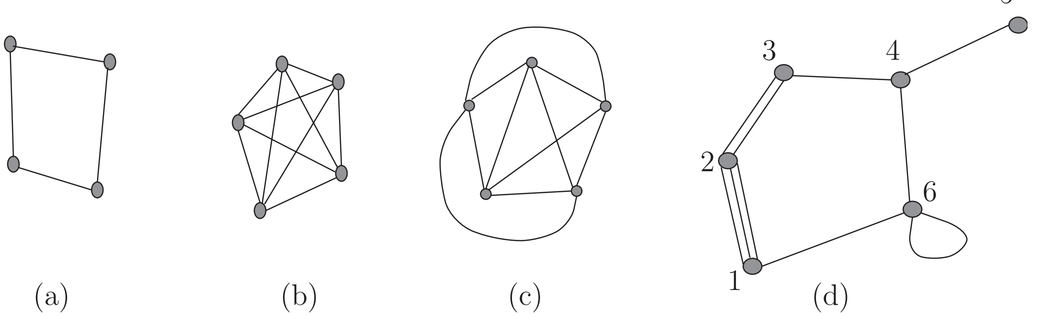
Figure 6.1: A stylized map of some eastern US cities.



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

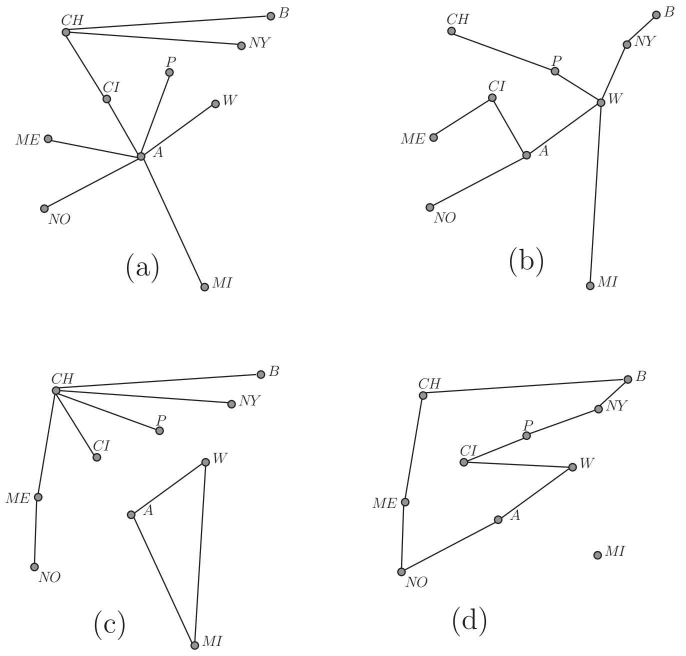
All of the examples we have seen so far have a property that is not common to all graphs, namely that there is a path from every vertex to every other vertex.

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.

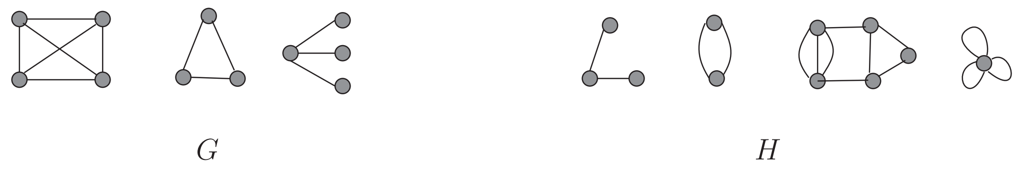
Notice that in graphs (a) and (b) it is possible to get from any vertex to any other vertex by a path. A graph is called *connected* there is a path between each two vertices of the graph. Notice that in graph (c) it is not possible to find a path from Atlanta to Boston, for example, and in graph (d) 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.



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.

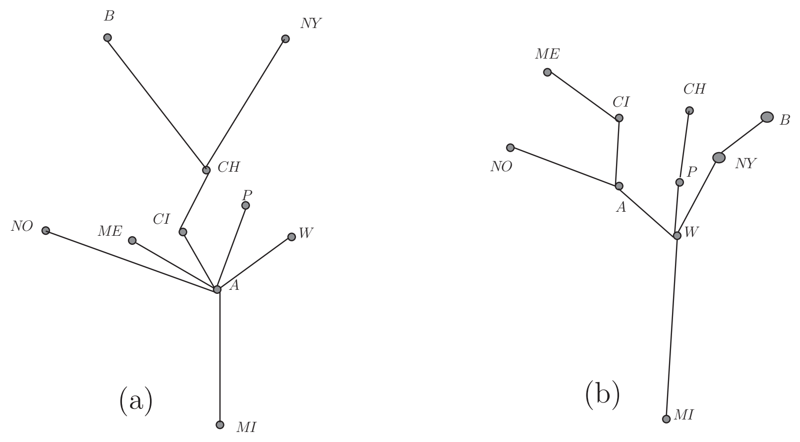
We don't normally distinguish which point on a cycle really is the starting point; for example we consider the cycle $\{A, W, MI, A\}$ to be the same as the cycle $\{W, MI, A, W\}$. Notice that there are cycles with one edge and cycles with two edges in the second graph of Figure 6.4. We call a graph 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

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?

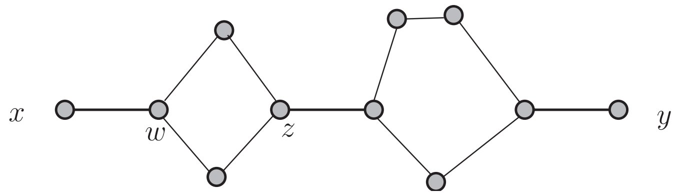
Exercise 6.1-9 How many edges does a tree with v vertices have?

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.

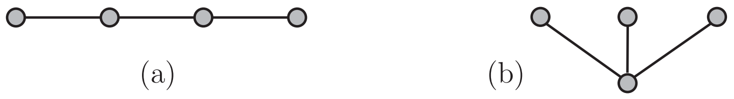
For Exercise 6.1-8, if the endpoints are in the same connected component, then the number of connected components won't change. If the endpoints of the edge are in different connected components, then the number of connected components can go down by one. Since an edge has two endpoints, it is impossible 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.

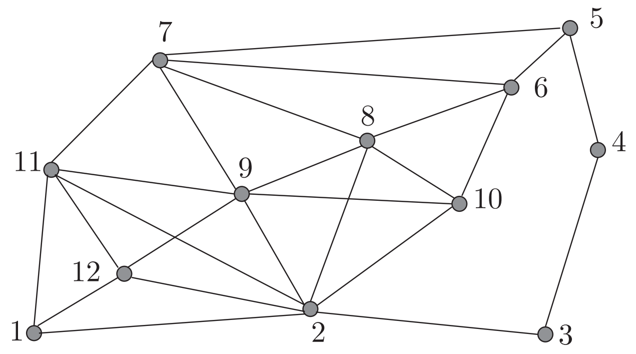
    (b) A tree on v vertices has $v - 1$ edges.

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

Problems

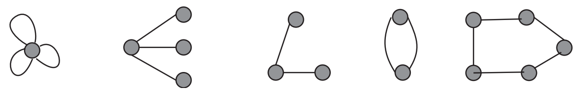
1. Find the shortest path you can from vertex 1 to vertex 5 in Figure 6.8.

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 $\epsilon_i$ from a vertex y in $S$ to a vertex $x_i$ not in $S$
(7)       $S = S \cup \{x_i\}$
(8)       $E' = E' \cup \epsilon_i$
(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 $E[\text{Vertex}[i]]$
(5)             If $\text{Intree}[j] \neq 1$
(6)                  $k = k + 1$
(7)                  $\text{Edge}[k] = \{V[\text{Vertex}[i]], V[j]\}$
(8)                  $\text{Intree}[j] = 1$
(9)                  $\text{Vertex}[k] = j$
(10)                 $\text{Vertexname}[k] = V[j]$.
(11) Print "Connected component"
(12) return $\text{Vertexname}[0 : k]$
(13) print "Spanning tree edges of connected component"
(14) return $\text{Edge}[1 : k]$
```

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 $V[\text{Vertex}[m]]$ 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 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.

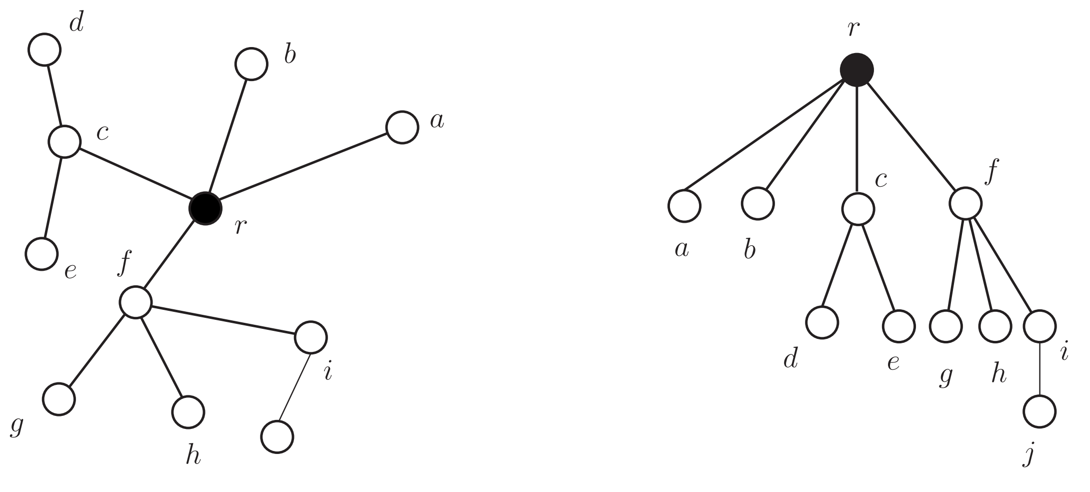
Theorem 6.8 *The unique path from $V[x]$ in a breadth first search spanning tree centered at the vertex $V[x]$ of a graph G to a vertex $V[y]$ is a shortest path from $V[x]$ to $V[y]$ in G.*

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 $\mathrm{BFN}(V[z']) < \mathrm{BFN}(V[z])$, 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.

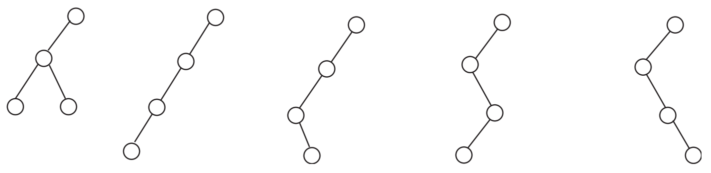
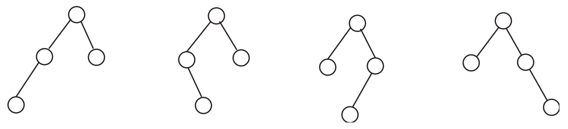


Figure 6.12: The four-vertex binary trees whose root has both a left and a right child.



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.

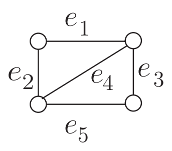
10. *Binary Tree.* We recursively describe a *binary tree* as

    - 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.)

11. A binary tree is complete if it is full and all its leaves have the same depth (see Exercise 6.2-8 and Exercise 6.2-9). 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.)

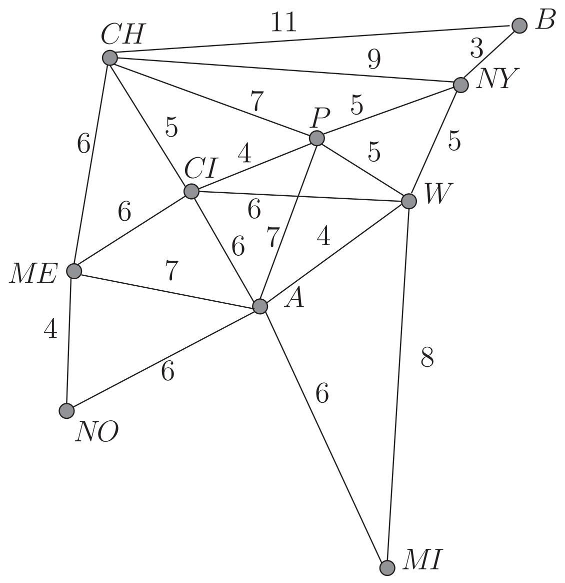
12. The *internal path length* of a binary tree is the sum, taken over all internal (see Exercise 6.2-11) vertices of the tree, of the depth of the vertex. The *external path length* of a binary tree is the sum, taken over all leaf vertices of the tree, of the depth of the vertex. Show that in a full 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 6.14 we have added numbers to the edges of the graph of Figure 6.1 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 6.14: A stylized map of some eastern US cities.



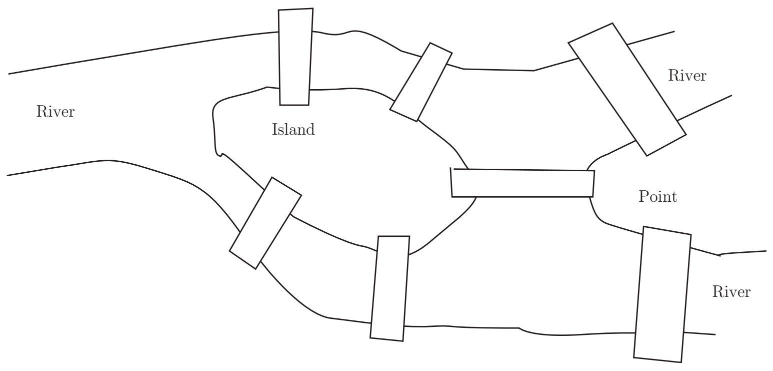
Give an algorithm to select a spanning tree of minimum total weight from a weighted graph and apply it to find a minimum spanning tree of the weighted graph in Figure 6.14. Show that your algorithm works and analyze how much time it takes.

6.3 Eulerian and Hamiltonian Paths and Tours

Eulerian Tours and Trails

Exercise 6.3-1 In an article generally acknowledged to be one of the origins of the graph theory [7] Leonhard Euler (pronounced Oiler) described a geographic problem that he offered as an elementary example of what he called "the geometry of position." The problem, known as the "Königsberg Bridge Problem," concerns the town of Königsberg in Prussia (now Kaliningrad in Russia), which is shown in a schematic map (circa 1700) in Figure 6.15. Euler tells us that the citizens amused themselves

Figure 6.15: A schematic map of Königsberg

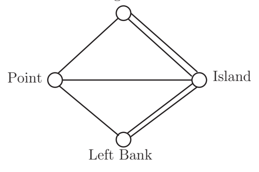


by trying to find a walk through town that crossed each of the seven bridges once and only once (and, hopefully, ended where it started). Is such a walk possible?

In Exercise 6.3-1, such a walk will enter a land mass on a bridge and leave it on a different bridge, so except for the starting and ending point, the walk requires two new bridges each time it enters and leaves a land mass. Thus each of these land masses must be at the end of an even number of bridges. However, as we see from Figure 6.15 each land mass is at the end of an odd number of bridges. Therefore no such walk is possible.

We can represent the map in Exercise 6.3-1 more compactly with the graph in Figure 6.16. In graph theoretic terminology Euler's question asks whether there is a path, starting and ending

Figure 6.16: A graph to replace the schematic map of Königsberg



[7]Reprinted in *Graph Theory 1736-1936* by Biggs, Lloyd and Wilson (Clarendon, 1976)

at the same vertex, that uses each edge exactly once.

Exercise 6.3-2 Determine whether the graph in Figure 6.1 has a closed path that includes each edge of the graph exactly once, and find one if it does.

Exercise 6.3-3 Find the strongest condition you can that has to be satisfied by a graph that has a path, starting and ending at the same place, that includes each vertex at least once and each edge once and only once. Such a path is known as an *Eulerian Tour* or *Eulerian Circuit.*

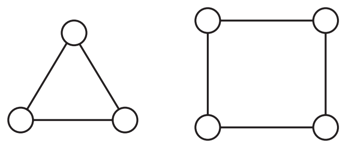
Exercise 6.3-4 Find the strongest condition you can that has to be satisfied by a graph that has a path, starting and ending at different places, that includes each vertex at least once and each edge once and only once. Such a path is known as an *Eulerian Trail*

Exercise 6.3-5 Determine whether the graph in Figure 6.1 has an Eulerain Trail and find one if it does.

The graph in Figure 6.1 cannot have a closed path that includes each edge exactly once because if the initial vertex of the path were P, then the number of edges incident with P would have to be one at the beginning of the path, plus two for each time P appears before the end of the path, plus one more for the time P would appear at the end of the path, so the degree of P would have to be even. But if P were not the initial vertex of a closed path including all the edges, each time we entered P on one edge, we would have to leave it on a second edge, so the number of edges incident with P would have to be even. Thus in Exercise 6.3-2 there is no closed path that includes each edge exactly once.

Notice that, just as we argued for a walk through Königsberg, in any graph with an Eulerian Circuit, each vertex except for the starting-finishing one will be paired with two new edges (those preceding and following it on the path) each time it appears on the path. Therefore each of these vertices is incident with an even number of edges. Further, the starting vertex is incident with one edge at the beginning of the path and is incident with a different edge at the end of the path. Each other time it occurs, it will be paired with two edges. Thus this vertex is incident with an even number of edges as well. Therefore a natural condition a graph must satisfy if it has an Eulerian Tour is that each vertex has even degree. But Exercise 6.3-3 asked us for the strongest condition we could find that a graph with an Eulerian Tour would satisfy. How do we know whether this is as strong a condition as we could devise? In fact it isn't, the graph in Figure 6.17 clearly has no Eulerian Tour because it is disconnected, but every vertex has even degree.

Figure 6.17: This graph has no Eulerian Tour, even though each vertex has even degree.



The point that Figure 6.17 makes is that in order to have an Eulerian Tour, a graph must be connected as well as having only vertices of even degree. Thus perhaps the strongest condition we can find for having an Eulerian Tour is that the graph is connected and every vertex has even degree. Again, the question comes up "How do we show this condition is as strong as possible, if indeed it is?" We showed a condition was not as strong as possible by giving an example of a graph that satisfied the condition but did not have an Eulerian Tour. What if we could show that no such example is possible, i.e. we could prove that a graph which is connected and in which every vertex has even degree does have an Eulerian Tour? Then we would have shown our condition is as strong as possible.

Theorem 6.9 *A graph has an Eulerian Tour if and only if it is connected and each vertex has even degree.*

Proof: A graph must be connected to have an Eulerian tour, because there must be a path that includes each vertex, so each two vertices are joined by a path. Similarly, as explained earlier, each vertex must have even degree in order for a graph to have an Eulerian Tour. Therefore we need only show that if a graph is connected and each vertex has even degree, then it has an Eulerain Tour. We do so with a recursive construction. If G has one vertex and no edges, we have an Eulerian tour consisting of one vertex and no edges. So suppose G is connected, has at least one edge, and each vertex of G has even degree. Now, given distinct vertices x_0, x_1, …, x_i and edges ϵ_1, ϵ_2, …, ϵ_i such that $x_0\epsilon_1x_1\dots\epsilon_ix_i$ is a path, choose an edge ϵ_{i+1} from x_i to a vertex x_{j+1}. If x_{j+1} is x_0, stop. Eventually this process must stop because G is finite, and (since each vertex in G has even degree) when we enter a vertex other than x_0, there will be an edge on which we can leave it. This gives us a closed path C. Delete the edges of this closed path from the edge set of G. This gives us a graph G' in which each vertex has even degree, because we have removed two edges incident with each vertex of the closed path (or else we have removed a loop). However, G' need not be connected. Each connected component of G' is a connected graph in which each vertex has even degree. Further, each connected component of G' contains at least one element x_i. (Suppose a connected component C contained no x_i. Since G is connected, for each i, there is a path in G from each vertex in C to each vertex x_i. Choose the shortest such path, and suppose it connects a vertex y in C to x_j. Then no edge in the path can be in the closed path, or else we would have a shorter path from y to a different vertex x_i. Therefore removing the edges of the closed path leaves y connected to x_j in C, so that C contains an x_i after all, a contradiction.) We may assume inductively that each connected component has fewer edges than G, so each connected component has an Eulerian Tour. Now we may begin to recursively construct an Eulerian Tour of G by starting at x_j, and taking an Eulerian Tour of the connected component containing x_j. Then given a sequence x_j, x_1, …, x_k such that the Eulerian tour we have constructed so far includes the vertices x_j through x_k, the vertices and edges of the connected components of G' containing the vertices x_k through x_k, the edges ϵ_{j+1} through ϵ_k, we add the edge e_{k+1} and the vertex x_{k+1} to our tour, and if the vertices and edges of the connected component of G' containing x_{k+1} are not already in our tour, we add an Eulerian Tour of the connected component of G' containing x_{k+1} to our tour. When we add the last edge and vertex of our closed path to the path we have been constructing, every vertex and edge of the graph will have to be in the path we have constructed, because every vertex is in some connected component of G', and every edge is either an edge of the first closed path or an edge of some connected component of G'. Therefore if G is connected and each vertex of G has even degree, then G has an Eulerian Tour. ■

A graph with an Eulerian Tour is called an *Eulerian Graph.*

In Exercise 6.3-4, each vertex other than the initial and final vertices of the walk must have even degree by the same reasoning we used for Eulerian tours. But the initial vertex must have odd degree, because the first time we encounter it in our Eulerian Trail it is incident with one edge in the path, but each succeeding time it is incident with two edges in the path. Similarly the final vertex must have odd degree. This makes it natural to guess the following theorem.

Theorem 6.10 *A graph G has an Eulerian Trail if and only if G is connected and all but two of the vertices of G have even degree.*

Proof: We have already shown that if G has an Eulerian Trail, then all but two vertices of G have even degree and these two vertices have odd degree.

Now suppose that G is a connected graph in which all but two vertices have even degree. Suppose the two vertices of odd degree are x and y. Add an edge ϵ joining x and y to the edge set of G to get G'. Then G' has an Eulerian Tour by Theorem 6.9. One of the edges of the tour is the added edge. We may traverse the tour starting with any vertex and any edge following that vertex in the tour, so we may begin the tour with either $x\epsilon y$ or $y\epsilon x$. By removing the first vertex and ϵ from the tour, we get an Eulerian Trail. ■

By Theorem 6.10, there is no Eulerian Trail in Exercise 6.3-5.

Notice that our proof of Theorem 6.9 gives us a recursive algorithm for constructing a Tour. Namely, we find a closed walk W starting and ending at a vertex we choose, identify the connected components of the graph $G - W$ that results from removing the closed walk, and then follow our closed walk, pausing each time we enter a new connected component of $G - W$ to recursively construct an Eulerian Tour of the component and traverse it before returning to following our closed walk. It is possible that the closed walk we remove has only one edge (or in the case of a simple graph, some very small number of edges), and the number of steps needed for a breadth first search is $\Theta(e')$, where e' the number of edges in the graph we are searching. Thus our construction could take $\Theta(e)$ steps, each of which involves examining $\Theta(e)$ edges, and therefore our algorithm takes $O(e^2)$ time. (We get a big Oh bound and not a big Theta bound because it is also possible that the closed walk we find the first time is an Eulerian tour.)

It is an interesting observation on the progress of mathematical reasoning that Euler made a big deal in his paper of explaining why it is necessary for each land mass to have an even number of bridges, but seemed to consider the process of constructing the path rather self-evident, as if it was hardly worth comment. For us, on the other hand, proving that the construction is possible if each land mass has an even number of bridges (that is, showing that the condition that each land mass has an even number of bridges is a sufficient condition for the existence of an Eulerian tour) was a much more significant effort than proving that having an Eulerian tour requires that each land mass has an even number of bridges. The standards of what is required in order to back up a mathematical claim have changed over the years.

Hamiltonian Paths and Cycles

A natural question to ask in light of our work on Eulerian tours is whether we can state necessary and sufficient conditions for a graph to have a closed path that includes each vertex exactly once (except for the beginning and end). An answer to this question would have the potential to be quite useful. For example, a salesperson might have to plan a trip through a number of cities which are connected by a network of airline routes. Planning the trip so the salesperson would travel through a city only when stopping there for a sales call would minimize the number of flights the needed. This question came up in a game, called "around the world," designed by William Rowan Hamilton. In this game the vertices of the graph were the vertices of a dodecahedron (a twelve sided solid in which each side is a pentagon), and the edges were the edges of the solid. The object was to design a trip that started at one vertex and visited each vertex once and then returned to the starting vertex along an edge. Hamilton suggested that players could take turns, one choosing the first five cities on a tour, and the other trying to complete the tour. It is because of this game that a cycle that includes each vertex of the graph exactly once (thinking of the first and last vertex of the cycle as the same) is called a *Hamiltonian Cycle*. A graph is called Hamiltonian if it has a Hamiltonian cycle.. A *Hamiltonian Path* is a simple path that includes each vertex of the graph exactly once. It turns out that nobody yet knows (and as we shall explain briefly at the end of the section, it may be reasonable to expect that nobody will find) uesful necessary and sufficient conditions for a graph to have a Hamiltonian Cycle or a Hamiltonian Path that are significantly easier to verify than trying all permutations of the vertices to see if taking the vertices in the order of that permutation to see if that order defines a Hamiltonian Cycle or Path. For this reason this branch of graph theory has evolved into theorems that give sufficient conditions for a graph to have a Hamiltonian Cycle or Path; that is theorems that say all graphs of a certain type have Hamiltonian Cycles or Paths, but do not characterize all graphs that have Hamiltonian Cycles of Paths.

Exercise 6.3-6 Describe all values of n such that a complete graph on n vertices has a Hamiltonian Path. Describe all values of n such that a complete graph on n vertices has a Hamiltonian Cycle.

Exercise 6.3-7 Determine whether the graph of Figure 6.1 has a Hamiltonian Cycle or Path, and determine one if it does.

Exercise 6.3-8 Try to find an interesting condition involving the degrees of the vertices of a simple graph that guarantees that the graph will have a Hamiltonian cycle. Does your condition apply to graphs that are not simple? (There is more than one reasonable answer to this exercise.)

In Exercise 6.3-6, the path consisting of one vertex and no edges is a Hamiltonian path but not a Hamiltonian cycle in the complete graph on one vertex. (Recall that a path consisting of one vertex and no edges is not a cycle.) Similarly, the path with one edge in the complete graph K_2 is a Hamiltonian path but not a Hamiltonian cycle, and since K_2 has only one edge, there is no Hamiltonian cycle in the K_2. In the complete graph K_n, any permutation of the vertices is a list of the vertices of a Hamiltonian path, and if $n > 3$, such a Hamiltonian Path from x_1 to x_n, followed by the edge from x_n to x_1 and the vertex x_1 is a Hamiltonian Cycle. Thus each complete graph has a Hamiltonian Path, and each complete graph with more than three vertices has a Hamiltonian Cycle.

In Exercise 6.3-7, the path with vertices NO, A, MI, W, P, NY, B, CH, CL, and ME is a Hamiltonian Path, and adding the edge from ME to NO gives a Hamiltonian Cycle.

Based on our observation that the complete graph on n vertices has a Hamiltonian Cycle if $n > 2$, we might let our condition be that the degree of each vertex is one less than the number of vertices, but this would be uninteresting since it would simply restate what we already know for complete graphs. The reason why we could say that K_n has a Hamiltonian Cycle when $n > 3$ was that when we entered a vertex, there was always an edge left on which we could leave the vertex. However the condition that each vertex has degree $n-1$ is stronger than we needed, because until we were at the second-last vertex of the cycle, we had more choices than we needed for edges on which to leave the vertex. On the other hand, it might seem that even if n were rather large, the condition that each vertex should have degree $n-2$ would not be sufficient to guarantee a Hamiltonian cycle, because when we got to the second last vertex on the cycle, all of the $n-2$ vertices it is adjacent to might already be on the cycle and different from the first vertex, so we would not have an edge on which we could leave that vertex. However there is the possibility that when we had some choices earlier, we might have made a different choice and thus included this vertex earlier on the cycle, giving us a different set of choices at the second last vertex. In fact, if $n > 3$ and each vertex has degree at least $n-2$, then we could choose vertices for a path more or less as we did for the complete graph until we arrived at vertex $n-1$ on the path. Then we could complete a Hamiltonian path unless x_{n-1} was adjacent only to the first $n-2$ vertices on the path. In this last case, the first $n-1$ vertices would form a cycle, because x_{n-1} would be adjacent to x_1. Suppose y was the vertex not yet on the path. Since y has degree $n-2$ and y is not adjacent to x_{n-1}, y would have to be adjacent to the first $n-2$ vertices on the path. Then since $n > 3$, we could take the path $x_1yx_2\ldots x_{n-1}x_1$ and we would have a Hamiltonian cycle. Of course unless n were four, we could also insert y between x_2 and x_3 (or any x_{i-1} and x_i such that $i < n-1$, so we would still have a great deal of flexibility. To push this kind of reasoning further, we will introduce a new technique that often appears in graph theory. We will point out our use of the technique after the proof.

Theorem 6.11 (Dirac) *If every vertex of a v-vertex simple graph G with at least three vertices has degree at least $v/2$, then G has a Hamiltonian cycle.*

Proof: Suppose, for the sake of contradiction that there is a graph G_1 with no Hamiltonian Cycle in which each vertex has degree at least $v/2$. If we add edges joining existing vertices to G_1, each vertex will still have degree at least $v/2$. If add all possible edges to G_1 we will get a complete graph, and it will have a Hamiltonian cycle. Thus if we continue adding edges to G_1, we will at some point reach a graph that does have a Hamiltonian cycle. Instead, we add edges to G_1 until we reach a graph G_2 that has no Hamiltonian cycle but has the property that if we add any edge to G_2, we get a Hamiltonian cycle. We say G_2 is *maximal* with respect to not having a Hamiltonian cycle. Suppose x and y are not adjacent in G_2. Then adding an edge between x and y to G_2 gives a graph with a Hamiltonian cycle, and x and y must be connected by the added edge in this Hamiltonian cycle. (Otherwise G_2 would have a Hamiltonian cycle.) Thus G_2 has a Hamiltonian path $x_1x_2\ldots x_v$ that starts at $x = x_1$ and ends at $y = x_v$. Further x and y are not adjacent.

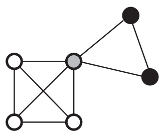
Before we stated our theorem we considered a case where we had a cycle on $f-1$ vertices and were going to put an extra vertex into it between two adjacent vertices. Now we have a path on f vertices from $x = x_1$ to $y = x_f$, and we want to convert it to a cycle. If we had that y is adjacent to some vertex x_i on the path while x is adjacent to x_{i+1}, then we could construct the Hamiltonian cycle $x_1x_{i+1}x_{i+2}\ldots x_fx_ix_{i-1}\ldots x_2x_1$. But we are assuming our graph does not have a Hamiltonian cycle. Thus for each x_i that x is adjacent to on the path $x_1x_2\ldots x_v$, y is not adjacent to x_{i-1}. Since all vertices are on the path, x is adjacent to at least $v/2$ vertices among x_2 through x_v. Thus y is not adjacent to at least $v/2$ vertices among x_1 through x_{v-1}. But there are only $v-1$ vertices, namely x_1 through x_{v-1}, vertices y could be adjacent to, since it is not adjacent to itself. Thus y is adjacent at most $v - 1 - v/2 = v/2 - 1$ vertices, a contradiction. Therefore if each vertex of a simple graph has degree at least $v/2$, the graph has a Hamiltonian Cycle. ■ The new tachnique was that of assuming we had a maximal graph (G_2) that did not have our desired property and then using this maximal graph in a proof by contradiction.

Exercise 6.3-9 Suppose $v = 2k$ and consider a graph G consisting of two complete graphs, one with k vertices, $x_1, \ldots x_k$ and one with $k + 1$ vertices, $x_k, \ldots x_{2k}$. Notice that we get a graph with exactly $2k$ vertices, because the two complete graphs have one vertex in common. How do the degrees of the vertices relate to v? Does the graph you get have a Hamiltonian cycle? What does this say about whether we can reduce the lower bound on the degree in Theorem 6.11?

Exercise 6.3-10 In the previous exercise, is there a similar example in the case $v = 2k+1$?

In Exercise 6.3-9, the vertices that lie in the complete graph with k vertices, with the exception of x_k, have degree $k - 1$. Since $v/2 = k$, this graph does not satisfy the hypothesis of Dirac's theorem which assumes that every vertex of the graph has degree at least $v/2$. We show the case in which $k = 3$ in Figure 6.18.

Figure 6.18: The vertices of K_4 are white or grey; those of K_3 are black or grey.



You can see that the graph in Figure 6.18 has no Hamiltonian cycle as follows. If an attempt at a Hamiltonian cycle begins at a white vertex, after crossing the grey vertex to include the black ones, it can never return to a white vertex without using the grey one a second time. The situation is similar if we tried to begin a Hamiltonian cycle at a black vertex. If we try to begin a Hamiltonian cycle at the grey vertex, we would next have to include all white vertices or all black vertices in our cycle and would then be stymied because we would have to take our path through the grey vertex a second time to change colors between white and black. As long as $k \geq 2$, the same argument shows that our graph has no Hamiltonian cycle. Thus the lower bound of $v/2$ in Dirac's theorem is tight; that is, we have a way to construct a graph with minimum degree $v/2 - 1$ (when v is even) for which there is no Hamiltonian cycle. If $v = 2k + 1$ we might consider two complete graphs of size $k + 1$, joined at a single vertex. Each vertex other than the one at which they are joined would have degree k, and we would have $k < k + 1/2 = v/2$, so again the minimum degree would be less than $v/2$. The same kind of argument that we used with the graph in Figure 6.18 would show that as long as $k \geq 1$, we have no Hamiltonian cycle.

If you analyze our proof of Dirac's theorem, you will see that we really used only a consequence of the condition that all vertices have degree at least $v/2$, namely that for any two vertices, the sum of their degrees is at least n.

Theorem 6.12 (Ore) *If G is a v-vertex simple graph with $n \geq 3$ such that for each two nonadjacent vertices x and y the sum of the degrees of x and y is at least v, then G has a Hamiltonian cycle.*

Proof: See Problem 13. ■

NP-Complete Problems

As we began the study of Hamiltonian Cycles, we mentioned that the problem of determining whether a graph has a Hamiltonian Cycle seems significantly more difficult than the problem of determining whether a graph has a Eulerian Tour. On the surface these two problems have significant similarities.

- Both problems whether a graph has a particular property. (Does this graph have a Hamiltonian/Eulerian closed path?) The answer is simply yes or no.
- For both problems, there is additional information we can provide that makes it relatively easy to check a yes answer if there is one. (The additional information is a closed path. We simply check whether the closed path includes each edge or each vertex exactly once.)

But there is a striking difference between the two problems as well. It is reasonably easy to find an Eulerian path in a graph that has one (we saw that the time to use the algorithm implicit in the proof of Theorem 6.9 is $O(e^2)$ where e is the number of edges of the graph. However, nobody knows how to actually find a permutation of the vertices that is a Hamiltonian path without checking essentially all permutations of the vertices.[8] This puts us in an interesting position. Although if someone gets lucky and guesses a permutation that is a Hamiltonian path, we can quickly verify the person's claim to have a Hamiltonian path, but in a graph of reasonably large size we have no practical method for finding a Hamiltonian path.

This difference is the essential difference between the class **P** of problems said to be solvable in polynomial time and the class **NP** of problems said to be solvable in nondeterministic polynomial time. We are not going to describe these problem classes in their full generality. A course in formal languages or perhaps algorithms is a more appropriate place for such a discussion. However in order to give a sense of the difference between these kinds of problems, we will talk about them in the context of graph theory. A question about whether a graph has a certain property is called a *graph decision problem.* Two examples are the question of whether a graph has an Eulerian tour and the question of whether a graph has a Hamiltonian cycle.

A graph decision problem has a yes/no answer. A **P**-*algorithm* or polynomial time algorithm for a property takes a graph as input and in time $O(n^k)$, where k is a positive integer independent of the input graph and n is a measure of the amount of information needed to specify the input graph, it outputs the answer "yes" if and only if the graph does have the property. We say the algorithm *accepts* the graph if it answers yes. (Notice we don't specify what the algorithm does if the graph does not have the property, except that it doesn't output yes.) We say a property of graphs is *in the class* **P** if there is a **P**-algorithm that accepts exactly the graphs with the property.

An **NP**-algorithm (non-deterministic polynomial time) for a property takes a graph and $O(n^j)$ additional information, and in time $O(n^k)$, where k and j are positive integers independent of

[8]We say essentially because one can eliminate some permutations immediately; for example if there is no edge between the first two elements of the permutation, then not only can we ignore the rest of this permutation, but we may ignore any other permutation that starts in this way. However nobody has managed to find a sub-exponential time algorithm for solving the Hamiltonian cycle problem.

the the input graph and n is a measure of the amount of information needed to specify the input graph, outputs the answer yes if and only if it can use the additional information to determine that the graph has the property. For example for the property of being Hamiltonian, the algorithm might input a graph and a permutation of the vertex set of the graph. The algorithm would then check the permutation to see if it lists the vertices in the order of a Hamiltonian cycle and output "yes" if it does. We say the algorithm *accepts* a graph if there is additional information it can use with the graph as an input to output "yes." We call such an algorithm nondeterministic, because whether or not it accepts a graph is determined not merely by the graph but by the additional information as well. In particular, the algorithm might or might not accept every graph with the given property. We say a property is *in the class* **NP** if there is an **NP**-algorithm that accepts exactly the graphs with the property. Since graph decision problems ask us to decide whether or not a graph has a given problem, we adopt the notation **P** and **NP** to describe problems as well. We say a decision problem is in **P** or **NP** if the graph property it asks us to decide is in **P** or **NP** respectively.

When we say that a nondeterministic algorithm uses the additional information, we are thinking of "use" in a very loose way. In particular, for a graph decision problem in **P**, the algorithm could simply ignore the additional information and use the polynomial time algorithm to determine whether the answer should be yes. Thus every graph property in **P** is also in **NP** as well. The question as to whether **P** and **NP** are the same class of problems has vexed computer scientists since it was introduced in 1968.

Some problems in **NP**, like the Hamiltonian path problem have an exciting feature: any instance[9] of any problem in **NP** can be translated in $O(n^k)$ steps, where n and k are as before, into $O(n^j)$ instances of the Hamilton path problem, where j is independent of n and k. In particular, the answer to the original problem is yes if and only if the answer to one of the Hamiltonian path problems is yes. The translation preserves whether or not the graph in the original instance of the problem is accepted. We say that the Hamiltonian Path problem is **NP**-complete. More generally, a problem in **NP** is called **NP**-complete if, for each other problem in **NP**, we can devise an algorithm for the second problem that has $O(n^k)$ steps (where n is a measure of the size of the input graph, and k is independent of n), including counting as one step solving an instance of the first problem, and accepts exactly the instances of the second problem that have a yes answer. The question of whether a graph has a clique (a subgraph that is a complete graph) of size j is another problem in **NP** that is **NP**-complete. In particular, if one **NP** complete problem has a polynomial time algorithm, every problem in NP is in **P**. Thus we can determine in polynomial time whether an arbitrary graph has a Hamiltonian path if and only if we can determine in polynomial time whether an arbitrary graph has a clique of (an arbitrary) size j. Literally hundreds of interesting problems are NP-complete. Thus a polynomial time solution to any one of them would provide a polynomial time solution to all of them. For this reason, many computer scientists consider a demonstration that a problem is NP-complete to be a demonstration that it is unlikely to be solved by a polynomial time algorithm.

This brief discussion of NP-completeness is intended to give the reader a sense of the nature and importance of the subject. We restricted ourselves to graph problems for two reasons. First, we expect the reader to have a sense of what a graph problem is. Second, no treatment of graph theory is complete without at least some explanation of how some problems seem to be much more intractable than others. However, there are **NP**-complete problems throughout mathematics and

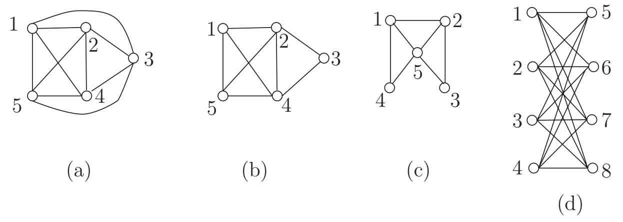
[9]An instance of a problem is a case of the problem in which all parameters are specified; for example a particular instance of the Hamiltonian Cycle problem is a case of the problem for a particular graph.

computer science. Providing a real understanding of the subject would require much more time than is available in an introductory course in discrete mathematics.

Important Concepts, Formulas, and Theorems

1. A graph that has a path, starting and ending at the same place, that includes each vertex at least once and each edge once and only once is called an *Eulerian Graph.* Such a path is known as an *Eulerian Tour* or *Eulerian Circuit.*
2. A graph has an Eulerian Tour if and only if it is connected and each vertex has even degree.
3. A path that includes each vertex of the graph at least once and each edge of the graph exactly once, but has different first and last endpoints, is known as an *Eulerian Trail*
4. A graph G has an Eulerian Trail if and only if G is connected and all but two of the vertices of G have even degree.
5. A cycle that includes each vertex of a graph exactly once (thinking of the first and last vertex of the cycle as the same) is called a *Hamiltonian Cycle.* A graph is called Hamiltonian if it has a Hamiltonian cycle.
6. A *Hamiltonian Path* is a simple path that includes each vertex of the graph exactly once.
7. (Dirac's Theorem) If every vertex of a v-vertex simple graph G with at least three vertices has degree at least $v/2$, then G has a Hamiltonian cycle.
8. (Ore's Theorem) If G is a v-vertex simple graph with $v \geq 3$ such that for each two non-adjacent vertices x and y the sum of the degrees of x and y is at least v, then G has a Hamiltonian cycle.
9. A question about whether a graph has a certain property is called a *graph decision problem.*
10. A **P**-*algorithm* or polynomial time algorithm for a property takes a graph as input and in time $O(n^k)$, where k is a positive integer independent of the input graph and n is a measure of the amount of information needed to specify the input graph, it outputs the answer "yes" if and only if the graph does have the property. We say the algorithm *accepts* the graph if it answers yes.
11. We say a property of graphs is *in the class* **P** if there is a **P**-algorithm that accepts exactly the graphs with the property.
12. An **NP**-algorithm (non-deterministic polynomial time) for a property takes a graph and $O(n^j)$ additional information, and in time $O(n^k)$, where k and j are positive integers independent of the the input graph and n is a measure of the amount of information needed to specify the input graph, outputs the answer yes if and only if it can use the additional information to determine that the graph has the property.
13. A graph decision problem in **NP** if called **NP**-complete if, for each other problem in **NP**, we can devise an algorithm for the second problem that has $O(n^k)$ steps (where n is a measure of the size of the input graph, and k is independent of n), including counting as one step solving an instance of the first problem, and accepts exactly the instances of the second problem that have a yes answer.

Figure 6.19: Some graphs

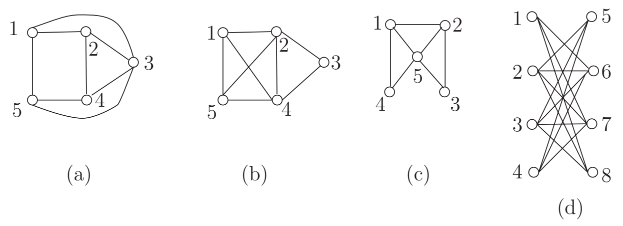


Problems

1. For each graph in Figure 6.19, either explain why the graph does not have an Eulerian circuit or find an Eulerian Circuit.

2. For each graph in Figure 6.20, either explain why the graph does not have an Eulerian Trail or find an Eulerian Trail.

Figure 6.20: Some more graphs



3. What is the minimum number of new bridges that would have to be built in Königsberg and where could they be built in order to give a graph with an Eulerian circuit?

4. If we built a new bridge in Königsberg between the Island and the top and bottom banks of the river, could we take a walk that crosses all the bridges and uses none twice? Either explain where could we start and end in that case or why we couldn't do it.

5. For which values of n does the complete graph on n vertices have an Eulerian Circuit?

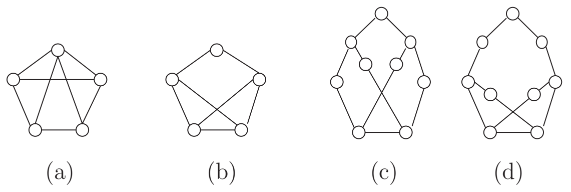
6. The hypercube graph Q_n has as its vertex set the n-tuples of zeros and ones. Two of these vertices are adjacent if and only if they are different in one position. The name comes from the fact that Q_3 can be drawn in three dimensional space as a cube. For what values of n is Q_n Eulerian?

7. For what values of n is the hypercube graph Q_n (see Problem 6) Hamiltonian?

8. Give an example of a graph which has a Hamiltonian cycle but no Eulerian Circuit and a graph which has an Eulerian Circuit but no Hamiltonian Cycle.

9. The complete bipartite graph $K_{m,n}$ is a graph with $m+n$ vertices. These vertices are divided into a set of size m and a set of size n. We call these sets the *parts* of the graph. Within each of these sets there are *no* edges. But between each pair of vertices in different sets, there is an edge. The graph $K_{4,4}$ is pictured in part (d) of Figure 6.19.
   (a) For what values of m and n is $K_{m,n}$ Eulerian?
   (b) For which values of m and n is $K_{m,n}$ Hamiltonian?
10. Show that the edge set of a graph in which each vertex has even degree may be partitioned into edge sets of cycles of the graph.
11. A cut-vertex of a graph is a vertex whose removal (along with all edges incident with it) increases the number of connected components of the graph. Describe any circumstances under which a graph with a cut vertex can be Hamiltonian.
12. Which of the graphs in Figure 6.21 satisfy the hypotheses of Dirac's Theorem? of Ore's Theorem? Which have Hamiltonian cycles?

Figure 6.21: Which of these graphs have Hamiltonian Cycles?



13. Prove Theorem 6.12.
14. The Hamiltonian Path problem is the problem of determining whether a graph has a Hamiltonian Path. Explain why this problem is in **NP**. Explain why the problem of determining whether a graph has a Hamiltonian Path is **NP**-complete.
15. The k-Path problem is the problem of determining whether a graph on n vertices has a path of length k, where k *is* allowed to depend on n. Show that the k-Path problem is **NP**-complete.
16. We form the Hamiltonian closure of a graph by constructing a sequence of graphs G_i with $G_0 = G$, and G_i formed from G_{i-1} by adding an edge between two nonadjacent vertices whose degree sum is at least nv. When we reach a G_i to which we cannot add such an edge, we call it a Hamiltonian Closure of G. Prove that a Hamiltonian Closure of a simple graph G is Hamiltonian if and only if G is.
17. Show that a simple connected graph has one and only one Hamiltonian closure.

6.4 Matching Theory

The idea of a matching

Suppose a school board is deciding among applicants for faculty positions. The school board has positions for teachers in a number of different grades, a position for an assistant librarian, two coaching positions, and for high school math and English teachers. They have many applicants, each of whom can fill more than one of the positions. They would like to know whether they can fill all the positions with people who have applied for jobs and have been judged as qualified.

Exercise 6.4-1 Table 6.1 shows a sample of the kinds of applications a school district might get for its positions. An x below an applicant's number means that that applicant

Table 6.1: Some sample job application data

job\applicant	1	2	3	4	5	6	7	8	9
assistant librarian	x		x	x					
second grade	x	x	x	x					
third grade	x	x		x					
high school math				x	x	x			
high school English				x		x	x		
asst baseball coach						x	x	x	x
asst football coach					x	x		x	

qualifies for the position to the left of the x. Thus candidate 1 is qualified to teach second grade, third grade, and be an assistant librarian. The coaches teach physical education when they are not coaching, so a coach can't also hold one of the listed teaching positions. Draw a graph in which the vertices are labelled 1 through 9 for the applicants, and s, t, l, m, e, b, and f for the positions. Draw an edge from an applicant to a position if that applicant can fill that position. Use the graph to help you decide if it is possible to fill all the positions from among the applicants deemed suitable. If you can do so, give an assignment of people to jobs. If you cannot, try to explain why not.

Exercise 6.4-2 Table 6.2 shows a second sample of the kinds of applications a school district might get for its positions. Draw a graph as before and use it to help you decide if it is possible to fill all the positions from among the applicants deemed suitable. If you can do so, give an assignment of people to jobs. If you cannot, try to explain why not.

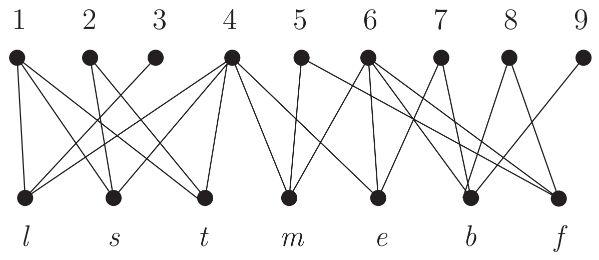
The graph of the data in Table 6.1 is shown in Figure 6.22.

From the figure it is clear that l:1, s:2, t:4, m:5, e:6, b:7, f:8 is one assignment of jobs to people that works. This assignment picks out a set of edges that share no endpoints. For example, the edge from l to 1 has no endpoint among s, t, m, e, b, f, 2, 3, 4, 5, 6, 7, or 8. A set of edges in a graph that share no endpoints is called a *matching* of the graph. Thus we have a matching between jobs and people that can fill the jobs. Since we don't want to assign two jobs to one

Table 6.2: Some other sample job application data

job\applicant	1	2	3	4	5	6	7	8	9
library assistant				x	x				
second grade	x	x	x					8	
third grade		x	x	x			x		x
high school math				x	x	x			
high school English					x	x			
asst baseball coach						x	x	x	x
asst football coach				x	x	x			

Figure 6.22: A graph of the data from Table 6.1.

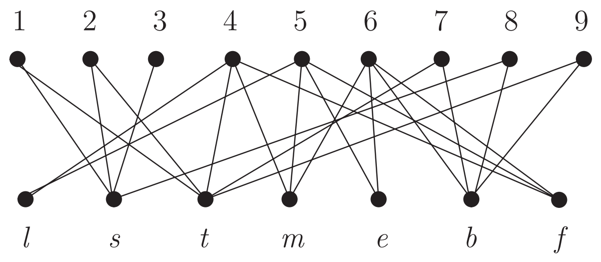


person or two people to one job, this is exactly the sort of solution we were looking for. Notice that the edge from l to 1 is a matching all by itself, so we weren't simply looking for a matching; we were looking for a matching that fills all the jobs. A matching is said to *saturate* a set X of vertices if every vertex in X is matched. We wanted a matching that saturates the jobs. In this case a matching that saturates all the jobs is a matching that is as big as possible, so it is also a *maximum* matching, that is, a matching that is at least as big as any other matching.

The graph in Figure 6.22 is an example of a "bipartite graph." A graph is called *bipartite* whenever its vertex set can be partitioned into two sets X and Y so that each edge connects a vertex in X with a vertex in Y. We can think of the jobs as the set X and the applicants as the set Y. Each of the two sets is called a *part* of the graph. A part of a bipartite graph is an example of an "independent set." A subset of the vertex set of a graph is called *independent* if no two of its vertices are joined by an edge. Thus a graph is bipartite if and only if its vertex set is a union of two independent sets. Notice that a bipartite graph cannot have any loop edges, because a loop would connect a vertex to a vertex in the same set. More generally, a vertex joined to itself by a loop cannot be in an independent set.

In a bipartite graph, it is sometimes easy to pick a maximum matching out just by staring at a drawing of the graph. However that is not always the case. Figure 6.23 is a graph of the data in Table 6.2. Staring at this Figure gives us many matchings, but no matching that saturates the set of jobs. But staring is not a proof, unless we can describe what we are staring at very well. Perhaps you tried to construct a matching by matching l to 4, s to 2, t to 7, m to 5, e to 6, b to 7, and then were frustrated when you got to f and 4, 5 and 6 were already used. You may then have gone back and tried to redo your earlier choices so as to keep one of 4, 5, or 6 free, and found you couldn't do so. This is because jobs l, m, e, and f are adjacent only to people 4, 5,

Figure 6.23: A graph of the data of Table 6.2.



and 6. Thus there are only three people qualified for these four jobs, and so there is no way we can fill them all.

We call the set $N(S)$ of all vertices adjacent to at least one vertex of S the *neighborhood* of S or the *neighbors* of S. In these terms, there is no matching that saturates a part X of a bipartite graph if there is some subset S of X such that the set $N(S)$ of neighbors of S is smaller than S. We call the set $N(S)$ of all vertices adjacent to at least one vertex of S the *neighborhood* of S or the *neighbors* of S. In symbols, we can summarize as follows.

Lemma 6.13 *If we can find a subset S of a part X of a bipartite graph G such that $|N(S)| < |S|$, then there is no matching of G that saturates X.*

Proof: A matching that saturates X must saturate S. But if there is such a matching, each element of S must be matched to a different vertex, and this vertex cannot be in S since $S \subseteq X$. Therefore there are edges from vertices in S to at least $|S|$ different vertices not in S, so $|N(S)| > |S|$, a contradiction. Thus there is no such matching.■

This gives a proof that there is no matching that saturates all the jobs, so the matching that matches l to 4, s to 2, t to 7, m to 5, e to 6, b to 7 is a maximum matching for the graph in Figure 6.23.

Another method you may have used to prove that there is no larger matching than the one we found is the following. When we matched l to 4, we may have noted that 4 is an endpoint of quite a few edges. Then when we matched s to 2, we may have noted that s is an endpoint of quite a few edges, and so is t. In fact, 4, s, and t touch 12 edges of the graph, and there are only 23 edges in the graph. If we could find three more vertices that touch the remaining edges of the graph, we would have six vertices that among them are incident with every edge. A set of vertices such that at least one of them is incident with each edge of a graph G is called a *vertex cover of the edges* of G, or a *vertex cover* of G for short. What does this have to do with a matching? Each matching edge would have to touch one, or perhaps two of the vertices in a vertex cover of the edges. Thus the number of edges in a matching is always less than the number of vertices in a vertex cover of the edges of a graph. Thus if we can find a vertex cover of size six in our graph in Figure 6.23, we will know that there is no matching that saturates the set of jobs since there are seven jobs. For future reference, we state our result about the size of a matching and the size of a vertex cover as a lemma.

Lemma 6.14 *The size of a matching in a graph G is no more than the size of a vertex cover of G.*

Proof: Given in the preceding discussion.■

We have seen that 4, s, and t are good candidates for being members of a relatively small vertex cover of the graph in Figure 6.23, since they cover more than half the edges of the graph. Continuing through the edges we first examined, we see that m, 6, and b are good candidates for a small vertex cover as well. In fact, $\{4, s, t, m, 6, b\}$ do form a vertex cover. Since we have a vertex cover of size six, we know a maximum matching has size no more than six. Since we have already found a six-edge matching, that is a maximum matching. Therefore with the data in Table 6.2, it is not possible to fill all the jobs.

Making matchings bigger

Practical problems involving matchings will usually lead us to search for the largest possible matching in a graph. To see how to use a matching to create a larger one, we will assume we have two matchings of the same graph and see how they differ, especially how a larger one differs from a smaller one.

Exercise 6.4-3 In the graph G of Figure 6.22, let M_1 be the matching $\{l,1\}$, $\{s,2\}$, $\{t,4\}$, $\{m,5\}$, $\{e,6\}$, $\{b,9\}$, $\{f,8\}$, and let M_2 be the matching $\{l,4\}$, $\{s,2\}$ $\{t,1\}$, $\{m,6\}$, $\{e,7\}$ $\{b,8\}$. Recall that for sets S_1 and S_2 the symmetric difference of S_1 and S_2, denoted by $S_1 \Delta S_2$ is $(S_1 \cup S_2) - (S_1 \cap S_2)$. Compute the set $M_1 \Delta M_2$ and draw the graph with the same vertex set as G and edge set $M_1 \Delta M_2$. Use different colors or textures for the edges from M_1 and M_2 so you can see their interaction. Describe the kinds of graphs you see as connected components as succinctly as possible.

Exercise 6.4-4 In Exercise 6.4-3, one of the connected components suggests a way to modify M_2 by removing one or more edges and substituting one or more edges from M_1 that will give you a larger matching M_2' related to M_2. In particular, this larger matching should saturate everything M_2 saturates and more. What is M_2' and what else does it saturate?

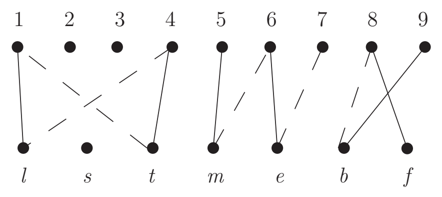
Exercise 6.4-5 Consider the matching $M = \{s,1\}, \{t,4\}, \{m,6\}, \{b,8\}$ in the graph of Figure 6.23. How does it relate to the simple path whose vertices are $3, s, 1, t, 4, m, 6, f$? Say as much as you can about the set M' that you obtain from M by deleting the edges of M that are in the path and adding to the result the edges of the path that are not in M.

In Exercise 6.4-3

$$M_1 \Delta M_2 = \{l,1\}, \{l,4\}, \{t,4\}, \{t,1\}, \{m,5\}, \{m,6\}, \{e,6\}, \{e,7\}, \{b,8\}, \{f,8\}, \{b,9\}.$$

We have drawn the graph in Figure 6.24. We show the edges of M_2 as dashed. As you see, it consists of a cycle with four edges, alternating between edges of M_1 and M_2, a path with four edges, alternating between edges of M_1 and M_2, and a path with three edges, alternating between edges of M_1 and M_2. We call a simple path or cycle an *alternating path* or *alternating cycle* for a matching M of a graph G if its edges alternate between edges in M and edges not in M. Thus our connected components were alternating paths and cycles for both M_1 and M_2. The example we just discussed shows all the ways in which two matchings can differ in the following sense.

Figure 6.24: The graph for Exercise 6.4-3.



Lemma 6.15 (Berge) *If M_1 and M_2 are matchings of a graph $G = (V, E)$ then the connected components of $M_1 \Delta M_2$ are cycles with an even number of vertices and simple paths. Further, the cycles and paths are alternating cycles and paths for both M_1 and M_2.*

Proof: Each vertex of the graph $(V, M_1\Delta M_2)$ has degree 0, 1, or two. If a component has no cycles it is a tree, and the only kind of tree that has vertices of degree 1 and two is a simple path. If a component has a cycle, then it cannot have any edges other than the edges of the cycle incident with its vertices because the graph would then have a vertex of degree 3 or more. Thus the component must be a cycle. If two edges of a path or cycle in $(V, M_1\Delta M_2)$ share a vertex, they cannot come from the same matching, since two edges in the same matching do not share a vertex. Thus alternating edges of a path or cycle of $(V, M_1\Delta M_2)$ must come from different matchings. ■

Corollary 6.16 *If M_1 and M_2 are matchings of a graph $G = (V, E)$ and $|M_2| < |M_1|$, then there is an alternating path for M_1 and M_2 that starts and ends with vertices saturated by M_2 but not by M_1.*

Proof: Since an even alternating cycle and an even alternating path in $(V, M_1\Delta M_2)$ have equal numbers of edges from M_1 and M_2, at least one component must be an alternating path with more edges from M_1 than M_2, because otherwise $|M_2| \leq |M_1|$. Since this is a component of $(V, M_1\Delta M_2)$, its endpoints lie only in edges of M_2, so they are saturated by M_2 but not M_1. ■

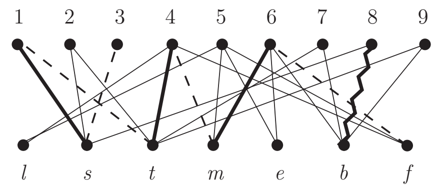
The path with three edges in Exercise 6.4-3 has two edges of M_1 and one edge of M_2. We see that if we remove $\{b, 8\}$ from M_2 and add $\{b, 9\}$ and $\{f, 8\}$, we get the matching

$$M_2' = \{\{l, 4\}, \{s, 2\}, \{t, 1\}, \{m, 6\}, \{e, 7\}, \{b, 9\}, \{f, 8\}\}.$$

This answers the question of Exercise 6.4-4. Notice that this matching saturates everything M_2 does, and also saturates vertices f and 9.

In Figure 6.25 we have shown the matching edges of the path in Exercise 6.4-5 in bold and the non-matching edges of the path as dashed. The edge of the matching not in the path is shown in zig-zag. Notice that the dashed edges and the zig-zag edge form a matching which is larger than M and saturates all the vertices that M does in addition to 3 and f. The path begins and ends with unmatched vertices, namely 3 and f, and and alternates between matching edges and non-matching edges. All but the first and last vertices of such a path lie on matching edges of the path and the endpoints of the path do not lie on matching edges. Thus no edges of the matching that are not path-edges will be incident with vertices on the path. Thus if we delete all the matching edges of the path from M and add all the other edges of the path to M, we will get

Figure 6.25: The path and matching of Exercise 6.4-5.



a new matching, because by taking every second edge of a simple path, we get edges that do not have endpoints in common. An alternating path is called an *augmenting path* for a matching M if it begins and ends with M-unsaturated vertices. That is, it is an alternating path that begins and ends with unmatched vertices. Our preceding discussion suggests the proof of the following theorem.

Theorem 6.17 (Berge) *A matching M in a graph is of maximum size if and only if M has no augmenting path. Further, if a matching M has an augmenting path P with edge set $E(P)$, then we can create a larger matching by deleting the edges in $M \cap E(P)$ from M and adding in the edges of $E(P) - M$.*

Proof: First if there is a matching M_1 larger than M, then by Corollary 6.16 there is an augmenting path for M. Thus if a matching has maximum size, it has no augmenting path. Further, as in our discussion of Exercise 6.4-5, if there is an augmenting path for M, then there is a larger matching for M. Finally, this discussion showed that if P is an augmenting path, we can get such a larger matching by deleting the edges in $M \cap E(P)$ and adding in the edges of $E(P) - M$. ■

Corollary 6.18 *While the larger matching of Theorem 6.17 may not contain M as a subset, it does saturate all the vertices that M saturates and two additional vertices.*

Proof: Every vertex incident with an edge in M is incident with some edge of the larger matching, and each of the two endpoints of the augmenting path is also incident with a matching edge. Because we may have removed edges of M to get the larger matching, it may not contain M.■

Matching in Bipartite Graphs

While our examples have all been bipartite, all our lemmas, corollaries and theorems about matchings have been about general graphs. In fact, it some of the results can be strengthened in bipartite graphs. For example, Lemma 6.14 tells us that the size of a matching is no more than the size of a vertex cover. We shall soon see that in a bipartite graph, the size of a maximum matching actually equals the size of a minimum vertex cover.

Searching for Augmenting Paths in Bipartite Graphs

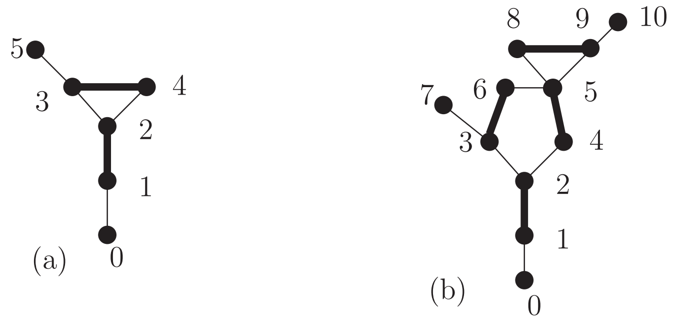
We have seen that if we can find an augmenting path for a matching M in a graph G, then we can create a bigger matching. Since our goal from the outset has been to create the largest matching possible, this helps us achieve that goal. However, you may ask, how do we find an augmenting path? Recall that a breadth-first search tree centered at a vertex x in a graph contains a path, in fact a shortest path, from x to every vertex y to which it is connected. Thus it seems that we ought to be able to alternate between matching edges and non-matching edges when doing a breadth-first search and find alternating paths. In particular, if we add vertex i to our tree by using a matching edge, then any edge we use to add a vertex *from* vertex i should be a non-matching edge. And if we add vertex i to our tree by using a non-matching edge, then any edge we use to add a vertex *from* vertex i should be a matching edge. (Thus there is just one such edge.) Because not all edges are available to us to use in adding vertices to the tree, the tree we get will not necessarily be a spanning tree of our original graph. However we can hope that if there is an augmenting path starting at vertex x and ending at vertex y, then we will find it by using breadth first search starting from x in this alternating manner.

Exercise 6.4-6 Given the matching $\{s, 2\}, \{t, 4\}, \{b, 7\}\{f, 8\}$ of the graph in Figure 6.22 use breadth-first search starting at vertex 1 in an alternating way to search for an augmenting path starting at vertex 1. Use the augmenting path you get to create a larger matching.

Exercise 6.4-7 Continue using the method of Exercise 6.4-6 until you find a matching of maximum size.

Exercise 6.4-8 Apply breadth-first search from vertex 0 in an alternating way to graph (a) in Figure 6.26. Does this method find an augmenting path? Is there an augmenting path?

Figure 6.26: Matching edges are shown in bold in these graphs.



For Exercise 6.4-6, if we begin at vertex 1, we add vertices l,s and t to our tree, giving them breadth-first numbers 1,2, and 3. Since l is not incident with a matching edge, we cannot continue the search from there. Since vertex s is incident with matching edge $\{s, 2\}$, we can use this edge to add vertex 2 to the tree and give it breadth-first number 4. This is the only vertex we can add from l since we can only use matching edges to add vertices from l. Similarly, from t we can add vertex 4 by using the matching edge $\{t, 4\}$ and giving it breadth-first number 5. All vertices adjacent to vertex 2 have already been added to the tree, but from vertex 4 we can use non-matching edges to add vertices m and e to our tree, giving them breadth first numbers 6 and 7. Now we can only use matching edges to add vertices to the tree from m or e, but there are no matching edges incident with them, so our alternating search tree stops here. Since m and e are unmatched, we know we have a path in our tree from vertex 1 to vertex m and a path from vertex 1 to vertex e. The vertex sequence of the path from 1 to m is $1s2t4m$ Thus our matching becomes $\{1,s\},\{2,t\},\{4,m\},\{b,7\},\{f,8\}$.

For Exercise 6.4-7 find another unmatched vertex and repeat the search. Working from vertex l, say, we start a tree by using the edges $\{l,1\},\{l,3\},\{l,4\}$ to add vertices 1, 3, and 4. We could continue working on the tree, but since we see that $l\{l,3\}3$ is an augmenting path, we use it to add the edge $\{l,3\}$ to the matching, short-circuiting the tree-construction process. Thus our matching becomes $\{1,s\},\{2,t\},\{l,3\},\{4,m\},\{b,7\}\{f,8\}$. The next unmatched vertex we see might be vertex 5. Starting from it, we add m and f to our tree, giving them breadth first numbers 1 and 2. From m we have the matching edge $\{m,4\}$, and from f we have the matching edge $\{f,8\}$, so we use them to add the vertices 4 and 8 to the tree. From vertex 4 we add l, s, t, and e to the tree, and from vertex 8 we add vertex b to the tree. All these vertices except e are in matching edges. Since e is not in a matching edge, we have discovered a vertex connected by an augmenting path to vertex 5. The path in the tree from vertex 5 to vertex e has vertex sequence $5m4e$, and using this augmenting path gives us the matching $\{1,s\},\{2,t\},\{l,3\},\{5,m\},\{4,e\},\{b,7\},\{f,8\}$. Since we now have a matching whose size is the same as the size of a vertex cover, namely the bottom part of the graph in Figure 6.22, we have a matching of maximum size.

For Exercise 6.4-8 we start at vertex 0 and add vertex 1. From vertex 1 we use our matching edge to add vertex 2. From vertex 2 we use our two non-matching edges to add vertices 3 and 4. However, vertices 3 and 4 are incident with the same matching edge, so we cannot use that matching edge to add any vertices to the tree, and we must stop without finding an augmenting path. From staring at the picture, we see there is an augmenting path, namely 012435, and it gives us the matching $\{\{0,1\},\{2,4\},\{3,5\}\}$. We would have similar difficulties in discovering either of the augmenting paths in part (b) of Figure 6.26.

It turns out to be the odd cycles in Figure 6.26 that prevent us from finding augmenting paths by our modification of breadth-first search. We shall demonstrate this by describing an algorithm which is a variation on the alternating breadth-first search we were using in solving our exercises. This algorithm takes a bipartite graph and a matching and either gives us an augmenting path or constructs a vertex cover whose size is the same as the size of the matching.

The Augmentation-Cover algorithm

We begin with a bipartite graph with parts X and Y and a matching M. We label the unmatched vertices in X with the label a (which stands for alternating). We number them in sequence as we label them. Starting with $i = 1$ and taking labeled vertices in order of the numbers we have assigned them, we use vertex number i to do additional labelling as follows.

1. If vertex i is in X, we label all unlabeled vertices adjacent to it with the label a and the name of vertex i. Then we number these newly labeled vertices, continuing our sequence of numbers without interruption.

2. If vertex i is in Y, and it is incident with an edge of M, its neighbor in the matching edge cannot yet be labeled. We label this neighbor with the label a and the name of vertex i.

3. If vertex i is in Y, and it is not incident with an edge of M, then we have discovered an augmenting path: the path from vertex i to the vertex we used to add it (and recorded at vertex i) and so on back to one of the unlabeled vertices in X. It is alternating by our labeling method, and it starts and ends with unsaturated vertices, so it is augmenting.

If we can continue the labelling process until no more labeling is possible and we do not find an augmenting path, then we let A be the set of labeled vertices. The set $C = (X - A) \cup (Y \cap A)$ then turns out to be a vertex cover whose size is the size of M. We call this algorithm the *augmentation-cover algorithm.*

Theorem 6.19 (König and Egerváry) *In a bipartite graph with parts X and Y, the size of a maximum sized matching equals the size of a minimum-sized vertex cover.*

Proof: In light of Berge's Theorem, if the augmentation-cover algorithm gives us an augmenting path, then the matching is not maximum sized, and in light of Lemma 6.14, if we can prove that the set C the algorithm gives us when there is no augmenting path is a vertex cover whose size is the size of the matching, we will have proved the theorem. To see that C is a vertex cover, note that every edge incident with a vertex in $X \cap A$ is covered, because its endpoint in Y has been marked with an a and so is in $Y \cap A$. But every other edge must be covered by $X - A$ because in a bipartite graph, each edge must be incident with a vertex in each part. Therefore C is a vertex cover. If an element of $Y \cap A$, were not matched, it would be an endpoint of an augmenting path, and so all elements of $Y \cap A$ are incident with matching edges. But every vertex of $X - A$ is matched because A includes all unmatched vertices of X. By step 2 of the augmentation-cover algorithm, if ϵ is a matching edge with an endpoint in $Y \cap A$, then the other endpoint must be in A. Therefore each matching edge contains only one member of C. Therefore the size of a maximum matching is the size of C. ■

Corollary 6.20 *The augmentation-cover algorithm applied to a bipartite graph and a matching of that graph produces either an augmenting path for the matching or a minimum vertex cover whose size equals the size of the matching.*

Before we proved the König-Egerváry Theorem, we knew that if we could find a matching and a vertex cover of the same size, then we had a maximum sized matching and a minimum sized vertex cover. However it is possible that in some graphs we can't test for whether a matching is as large as possible by comparing its size to that of a vertex cover because a maximum sized matching might be smaller than a minimum sized vertex cover. The König-Egárvary Theorem tells us that in bipartite graphs this problem never arises, so the test always works.

We had a second technique we used to show that a matching could not saturate the set X of all jobs in Exercise 6.4-2. In Lemma 6.13 we showed that if we can find a subset S of a part X of a bipartite graph G such that $|N(S)| < |S|$, then there is no matching of G that saturates X. In other words, to have a matching that saturates X in a bipartite graph on parts X and Y, it is necessary that $|N(S)| \geq |S|$ for *every* subset S of X. (When $S = \emptyset$, then so does $N(S)$.) This necessary condition is called *Hall's condition*, and Hall's theorem says that this necessary condition is sufficient.

Theorem 6.21 (Hall) *If G is a bipartite graph with parts X and Y, then there is a matching of G that saturates X if and only if $|N(S)| \geq |S|$ for every subset $\subseteq X$.*

Proof: In Lemma 6.13 we showed (the contrapositive of the statement) that if there is a matching of G, then $|N(S)| \geq |S|$ for every subset of X. (There is no reason to use a contrapositive argument though; if there is a matching that saturates X, then because matching edges have no endpoints in common, the elements of each subset S of X will be matched to at least $|S|$ different elements, and these will all be in $N(S)$.)

Thus we need only show that if the graph satisfies Hall's condition then there is a matching that saturates S. We will do this by showing that X is a minimum-sized vertex cover. Let C be some vertex cover of G. Let $S = X - C$. If ϵ is an edge from a vertex in S to a vertex $y \in Y$, ϵ cannot be covered by a vertex in $C \cap X$. Therefore ϵ must be covered by a vertex in $C \cap Y$. This means that $N(S) \subseteq C \cap Y$, so $|C \cap Y| \geq |N(S)|$. By Hall's condition, $N(S)| > |S|$. Therefore $|C \cap Y| \geq |S|$. Since $C \cap X$ and $C \cap Y$ are disjoint sets whose union is C, we can summarize our remarks with the equation

$$|C| = |C \cap X| + |C \cap Y| \geq |C \cap X| + |N(S)| \geq |C \cap X| + |S| = |C \cap X| + |C - X| = |X|.$$

X is a vertex cover, and we have just shown that it is a vertex cover of minimum size . Therefore a matching of maximum size has size $|X|$. Thus there is a matching that saturates X. ■

Good Algorithms

While Hall's theorem is quite elegant, applying it requires that we look at every subset of X, which would take us $\Omega\left(2^{|X|}\right)$ time. Similarly, actually finding a minimum vertex cover could involve looking at all (or nearly all) subsets of $X \cup Y$, which would also take us exponential time. However, the augmentation-cover algorithm requires that we examine each edge at most some fixed number of times and then do a little extra work; certainly no more than $O(e)$ work. We need to repeat the algorithm at most X times to find a maximum matching and minimum vertex cover. Thus in time $O(ev)$, we can not only find out whether we have a matching that saturates X; we can find such a matching if it exists and a vertex cover that proves it doesn't exist if it doesn't. However this only applies to bipartite graphs. The situation is much more complicated in non-bipartite graphs. In a paper which introduced the idea that a good algorithm is one that runs in time $O(n^c)$, where n is the amount of information needed to specify the input and c is a constant, Edmunds[10] developed a more complicated algorithm that extended the idea of a search tree to a more complicated structure he called a flower. He showed that this algorithm was good in his sense, introduced the problem class **NP**, and conjectured that $\mathbf{P} \neq \mathbf{NP}$. In a wry twist of fate, the problem of finding a minimum vertex cover problem (actually the problem of determining whether there is a vertex cover of size k, where k can be a function of v) is, in fact, **NP**-complete in arbitrary graphs. It is fascinating that the matching problem for general graphs turned out to be solvable in polynomial time, while determining the "natural" upper bound on the size of a matching, an upper bound that originally seemed quite useful, remains out of our reach.

[10]Jack Edmonds. Paths, Trees and Flowers. *Canadian Journal of Mathematics*, **17**, 1965 pp449-467

Important Concepts, Formulas, and Theorems

1. *Matching.* A set of edges in a graph that share no endpoints is called a *matching* of the graph.

2. *Saturate.* A matching is said to *saturate* a set X of vertices if every vertex in X is matched.

3. *Maximum Matching.* A matching in a graph is a *maximum matching* if it is at least as big as any other matching.

4. *Bipartite Graph.* A graph is called *bipartite* whenever its vertex set can be partitioned into two sets X and Y so that each edge connects a vertex in X with a vertex in Y. Each of the two sets is called a *part* of the graph.

5. *Independent Set.* A subset of the vertex set of a graph is called *independent* if no two of its vertices are connected by an edge. (In particular, a vertex connected to itself by a loop is in no independent set.) A part of a bipartite graph is an example of an 'independent set.

6. *Neighborhood.* We call the set $N(S)$ of all vertices adjacent to at least one vertex of S the *neighborhood* of S or the *neighbors* of S.

7. *Hall's theorem for a Matching in a Bipartite Graph.* If we can find a subset S of a part X of a bipartite graph G such that $|N(S)| < |S|$, then there is no matching of G that saturates X. If there is no subset $S \subseteq X$ such that $|N(S)| < |S|$, then there is a matching that saturates X.

8. *Vertex Cover.* A set of vertices such that at least one of them is incident with each edge of a graph G is called a *vertex cover of the edges* of G, or a *vertex cover* of G for short. In any graph, the size a matching is less than or equal to the size of any vertex cover.

9. *Alternating Path, Augmenting Path.* A simple path is called an *alternating path* for a matching M if, as we move along the path, the edges alternate between edges in M and edges not in M. An *augmenting path* is an alternating path that begins and ends at unmatched vertices.

10. *Berge's Lemma.* If M_1 and M_2 are matchings of a graph G then the connected components of $M_1 \Delta M_2$ are cycles with an even number of vertices and simple paths. Further, the cycles and paths are alternating cycles and paths for both M_1 and M_2.

11. *Berge's Corollary.* If M_1 and M_2 are matchings of a graph $G = (V, E)$ and $|M_1| > |M_2|$, then there is an alternating path for M_1 and M_2 that starts and ends with vertices saturated by M_1 but not by M_2.

12. *Berge's Theorem.* A matching M in a graph is of maximum size if and only if M has no augmenting path. Further, if a matching M has an augmenting path P with edge set $E(P)$, then we can create a larger matching by deleting the edges in $M \cap E(P)$ from M and adding in the edges of $E(P) - M$.

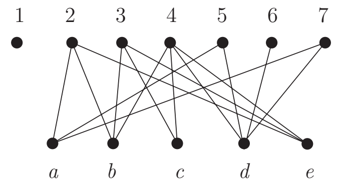
13. *Augmentation-Cover Algorithm.* The Augmentation-Cover algorithm is an algorithm that begins with a bipartite graph and a matching of that graph and produces either an augmenting path or a vertex cover whose size equals that of the matching, thus proving that the matching is a maximum matching.

14. *König-Egerváry Theorem.* In a bipartite graph with parts X and Y, the size of a maximum sized matching equals the size of a minimum-sized vertex cover.

Problems

1. Either find a maximum matching or a subset S of the set $X = \{a, b, c, d, e\}$ such that $|S| > |N(S)|$ in the graph of Figure 6.27

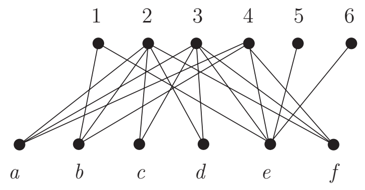
Figure 6.27: A bipartite graph



2. Find a maximum matching and a minimum vertex cover in the graph of Figure 6.27

3. Either find a matching which saturates the set $X = \{a, b, c, d, e, f\}$ in Figure 6.28 or find a set S such that $|N(S)| < |X|$.

Figure 6.28: A bipartite graph



4. Find a maximum matching and a minimum vertex cover in the graph of Figure 6.28.

5. In the previous exercises, when you were able to find a set S with $|S| > |N(S)|$, how did $N(S)$ relate to the vertex cover? Why did this work out as it did?

6. A star is a another name for a tree with one vertex connected to each of n other vertices. (So a star has $n + 1$ vertices.) What are the size of a maximum matching and a minimum vertex cover in a star with $n + 1$ vertices?

7. In Theorem 6.17 is it true that if there is an augmenting path P with edge set $E(P)$ for a matching M, then $M \Delta E(P)$ is a larger matching than M?

8. Find a maximum matching and a minimum vertex cover in graph (b) of Figure 6.26.

9. In a bipartite graph, is one of the parts always a maximum-sized independent set? What if the graph is connected?

10. Find infinitely many examples of graphs in which a maximum-sized matching is smaller than a minimum-sized vertex cover.

11. Find an example of a graph in which the maximum size of a matching is less than one quarter of the size of a minimum vertex cover.

12. Prove or give a counter-example: Every tree is a bipartite graph. (Note, a single vertex with no edges is a bipartite graph; one of the two parts is empty.)

13. Prove or give a counter-example. A bipartite graph has no odd cycles.

14. Let G be a connected graph with no odd cycles. Let x be a vertex of G. Let X be all vertices at an even distance from x, and let Y be all vertices at an odd distance from x. Prove that G is bipartite with parts X and Y.

15. What is the sum of the maximum size of an independent set and the minimum size of a vertex cover in a graph G? Hint: it is useful to think both about the independent set and its complement (relative to the vertex set).

6.5 Coloring and planarity

The idea of coloring

Graph coloring was one of the origins of graph theory. It arose from a question from Francis Guthrie, who noticed that that four colors were enough colors to color the map of the counties of England so that if two counties shared a common boundary line, then they got different colors. He wondered whether this was the case for any map. Through his brother he passed it on to Agustus DeMorgan, and in this way it seeped into the consciousness of the mathematical community. If we think of the counties as vertices and draw an edge between two vertices if their counties share some boundary line, we get a representation of the problem that is independent of such things as the shape of the counties, the amount of boundary line they share, etc. so that it captures the part of the problem we need to focus on. We now color the vertices of the graph, and for this problem we want to do so in such a way that adjacent vertices get different colors. We will return to this problem later in the section; we begin our study with another application of coloring.

Exercise 6.5-1 The executive committee of the board of trustees of a small college has seven members, Kim, Smith, Jones, Gupta, Ramirez, Wang, and Chernov. It has six subcommittees with the following membership

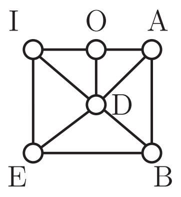
- Investments: W, R, G
- Operations: G, J, S, K
- Academic affairs: S, W, C
- Fund Raising: W, C, K
- Budget: G, R, C
- Enrollment: R, S, J, K

Each time the executive committee has a meeting, first each of the subcommittees meets with appropriate college officers, and then the executive committee gets together as a whole to go over subcommittee recommendations and make decisions. Two committees cannot meet at the same time if they have a member in common, but committees that don't have a member in common can meet at the same time. In this exercise you will figure out the minimum number of time slots needed to schedule all the subcommittee meetings. Draw a graph in which the vertices are named by the initials of the committee names and two vertices are adjacent if they have a member in common. Then assign numbers to the vertices in such a way that two adjacent vertices get different numbers. The numbers represent time slots, so they need not be distinct unless they are on adjacent vertices. What is the minimum possible number of numbers you need?

Because the problem of map coloring motivated much of graph theory, it is traditional to refer to the process of assigning labels to the vertices of a graph as coloring the graph. An assignment of labels, that is a function from the vertices to some set, is called a *coloring*. The set of possible labels (the range of the coloring function) is often referred to as a set of *colors*. Thus in Exercise 6.5-1 we are asking for a coloring of the graph. However, as with the map problem, we want a coloring in which adjacent vertices have different colors. A coloring of a graph is called a *proper coloring* if it assigns different colors to adjacent vertices.

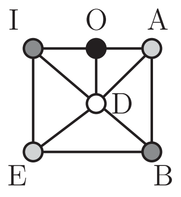
We have drawn the graph of Exercise 6.5-1 in Figure 6.29. We call this kind of graph an *intersection graph*, which means its vertices correspond to sets and it has an edge between two vertices if and only if the corresponding sets intersect.

Figure 6.29: The "intersection" graph of the committees.



The problem asked us to color the graph with as few colors possible, regarding the colors as 1,2,3, etc. We will represent 1 as a white vertex, 2 as a light grey vertex, 3 as a dark grey vertex and 4 as a black vertex. The triangle on the bottom requires three colors simply because all three vertices are adjacent. Since it doesn't matter which three colors we use, we choose arbitrarily to make them white, light grey, and dark grey. Now we know we need at least three colors to color the graph, so it makes sense to see if we can finish off a coloring using just three colors. Vertex I must be colored differently from E and D, so if we use the same three colors, it must have the same color as B. Similarly, vertex A would have to be the same color as E if we use the same three colors. But now none of the colors can be used on vertex O, because it is adjacent to three vertices of different colors. Thus we need at least four colors rather than 3, and we show a proper four-coloring in Figure 6.30.

Figure 6.30: A proper coloring of the committee intersection graph.



Exercise 6.5-2 How many colors are needed to give a proper coloring of the complete graph K_n?

Exercise 6.5-3 How many colors are needed for a proper coloring of a cycle C_n on $n = 3, 4, 5,$ and 6 vertices?

In Exercise 6.5-2 we need n colors to properly color K_n, because each pair of vertices is adjacent and thus must have two different colors. In Exercise 6.5-3, if n is even, we can just alternate two colors as we go around the cycle. However if n is odd, using two colors would require that they alternate as we go around the cycle, and when we colored our last vertex, it would be the same color as the first. Thus we need at least three colors, and by alternating two of them as we go around the cycle until we get to the last vertex and color it the third color we get a proper coloring with three colors.

The *chromatic number* of a graph G, traditionally denoted $\chi(G)$, is the minimum number of colors needed to properly color G. Thus we have shown that the chromatic number of the complete graph K_n is n, the chromatic number of a cycle on an even number of vertices is two, and the chromatic number of a cycle on an odd number of vertices is three. We showed that the chromatic number of our committee graph is 4.

From Exercise 6.5-2, we see that if a graph G has a subgraph which is a complete graph on n vertices, then we need at least n colors to color those vertices, so we need at least n colors to color G. this is useful enough that we will state it as a lemma.

Lemma 6.22 *If a graph G contains a subgraph that is a complete graph on n vertices, then the chromatic number of G is at least n.*

Proof: Given above.■

Interval Graphs

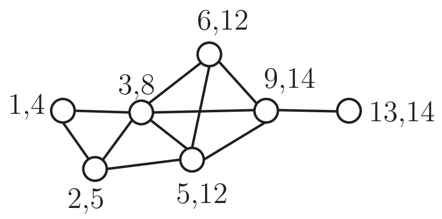
An interesting application of coloring arises in the design of optimizing compilers for computer languages. In addition to the usual RAM, a computer typically has some memory locations called registers which can be accessed at very high speeds. Thus values of variables which are going to be used again in the program are kept in registers if possible, so they will be quickly available when we need them. An optimizing compiler will attempt to decide the time interval in which a given variable may be used during a run of a program and arrange for that variable to be stored in a register for that entire interval of time. The time interval is not determined in absolute terms of seconds, but the relative endpoints of the intervals can be determined by when variables first appear and last appear as one steps through the computer code. This information is what is needed to set aside registers to use for the variables. We can think of coloring the variables by the registers as follows. We draw a graph in which the vertices are labeled with the variable names, and associated to each variable is the interval during which it is used. Two variables can use the same register if they are needed during non-overlapping time intervals. This is helpful, because registers are significantly more expensive than ordinary RAM, so they are limited in number. We can think of our graph on the variables as the intersection graph of the intervals. We want to color the graph properly with a minimum number of registers; hopefully this will be no more than the number of registers our computer has available. The problem of assigning variables to registers is called the *register assignment problem.*

An intersection graph of a set of intervals of real numbers is called an *interval graph.* The assignment of intervals to the vertices is called an *interval representation.* You will notice that so far in our discussion of coloring, we have not given an algorithm for properly coloring a graph efficiently. This is because the problem of whether a graph has a proper coloring with k colors, for any fixed k greater than 2 is another example of an **NP**-complete problem. However, for interval graphs, there is a very simple algorithm for properly coloring the graph in a minimum number of colors.

Exercise 6.5-4 Consider the closed intervals $[1,4], [2,5], [3,8], [5,12], [6,12], [7,14], [13,14]$.
Draw the interval graph determined by these intervals and find its chromatic number.

We have drawn the graph of Exercise 6.5-4 in Figure 6.31. (We have not included the square braces to avoid cluttering the figure.) Because of the way we have drawn it, it is easy to see a

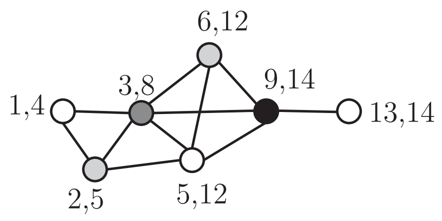
Figure 6.31: The graph of Exercise 6.5-4



subgraph that is a complete graph on four vertices, so we know by our lemma that the graph has chromatic number at least four. In fact, Figure 6.32 shows that the chromatic number is exactly four. This is no accident.

Figure 6.32: A proper coloring of the graph of Exercise 6.5-4 with four colors

.



Theorem 6.23 *In an interval graph G, the chromatic number is the size of the largest complete subgraph.*

Proof: List the intervals of an interval representation of the graph in order of their left endpoints. Then color them with the integers 1 through some number n by starting with 1 on the first interval in the list and for each succeeding interval, use the smallest color not used on any neighbor of the interval earlier in the list. This will clearly give a proper coloring. To see that the number of colors needed is the size of the largest complete subgraph, let n denote the largest color used, and choose an interval I colored with color n. Then, by our coloring algorithm, I must intersect with earlier intervals in the list colored 1 through $n-1$; otherwise we could have used a smaller color on I. All these intervals must contain the left endpoint of I, because they intersect I and come earlier in the list. Therefore they all have a point in common, so they form a complete graph on n vertices. Therefore the minimum number of colors needed is the size of a complete subgraph of G. But by Lemma 6.22, G can have no larger complete subgraph. Thus the chromatic number of G is the size of the largest complete subgraph of G. ■

Corollary 6.24 *An interval graph G may be properly colored using $\chi(G)$ consecutive integers as colors by listing the intervals of a representation in order of their left endpoints and going through* *the list, assigning the smallest color not used on an earlier adjacent interval to each interval in the list.*

Proof: This is the coloring algorithm we used in the proof of Theorem 6.23. ■

Notice that using the correspondence between numbers and grey-shades we used before, the coloring in Figure 6.32 is the one given by this algorithm. An algorithm that colors an (arbitrary) graph G with consecutive integers by listing its vertices in some order, coloring the first vertex in the list 1, and then coloring each vertex with the least number not used on any adjacent vertices earlier in the list is called a *greedy coloring algorithm.* We have just seen that the greedy coloring algorithm allows us to find the chromatic number of an interval graph. This algorithm takes time $O(n^2)$, because as we go through the list, we might consider every earlier entry when we are considering a given element of the list. It is good luck that we have a polynomial time algorithm, because even though we stated in Theorem 6.23 that the chromatic number is the size of the largest complete subgraph, determining whether the size of a largest complete subgraph in a general graph (as opposed to an interval graph) is k (where k may be a function of the number of vertices) is an **NP-complete** problem.

Of course we assumed that we were given an interval representation of our graph. Suppose we are given a graph that happens to be an interval graph, but we don't know an interval representation. Can we still color it quickly? It turns out that there is a polynomial time algorithm to determine whether a graph is an interval graph and find an interval representation. This theory is quite beautiful,[11] but it would take us too far afield to pursue it now.

Planarity

We began our discussion of coloring with the map coloring problem. This problem has a special aspect that we did not mention. A map is drawn on a piece of paper, or on a globe. Thus a map is drawn either on the plane or on the surface of a sphere. By thinking of the sphere as a completely elastic balloon, we can imagine puncturing it with a pin somewhere where nothing is drawn, and then stretching the pinhole until we have the surface of the balloon laid out flat on a table. This means we can think of all maps as drawn in the plane. What does this mean about the graphs we associated with the maps? Say, to be specific, that we are talking about the counties of England. Then in each county we take an important town, and build a road to the boundary of each county with which it shares more than a single boundary point. We can build these roads so that they don't cross each other, and the roads to a boundary line between two different counties join together at that boundary line. Then the towns we chose are the vertices of a graph representing the map, and the roads are the edges. Thus given a map drawn in the plane, we can draw a graph to represent it in such a way that the edges of the graph do not meet at any point except their endpoints.[12] A graph is called *planar* if it has a drawing in the plane such that edges do not meet except at their endpoints. Such a drawing is called a *planar drawing* of the graph. The famous four color problem asked whether all planar graphs have proper four colorings. In 1976, Apel and Haken, building on some of the early attempts at proving the theorem, used a computer to demonstrate that four colors are sufficient to color

[11]See, for example, the book *Algorithmic Graph Theory and the Perfect Graph Conjecture*, by Martin Golumbic, Academic Press, New York, 1980.

[12]We are temporarily ignoring a small geographic feature of counties that we will mention when we have the terminology to describe it

any planar graph. While we do not have time to indicate how their proof went, there is now a book on the subject that gives a careful history of the problem and an explanation of what the computer was asked to do and why, assuming that the computer was correctly programmed, that led to a proof.[13]

What we will do here is derive enough information about planar graphs to show that five colors suffice, giving the student some background on planarity relevant to the design of computer chips.

We start out with two problems that aren't quite realistic, but are suggestive of how planarity enters chip design.

Exercise 6.5-5 A circuit is to be laid out on a computer chip in a single layer. The design includes five terminals (think of them as points to which multiple electrical circuits may be connected) that need to be connected so that it is possible for a current to go from any one of them to any other without sending current to a third. The connections are made with a narrow layer of metal deposited on the surface of the chip, which we will think of as a wire on the surface of the chip. Thus if one connection crosses another one, current in one wire will flow through the other as well. Thus the chip must be designed so that no two wires cross. Do you think this is possible?

Exercise 6.5-6 As in the previous exercise, we are laying out a computer circuit. However we now have six terminals, labeled a, b, c, 1, 2, and 3, such that each of a, b, and c must be connected to each of 1, 2, and 3, but there must be no other connections. As before, the wires cannot touch each other, so we need to design this chip so that no two wires cross. Do you think this is possible?

The answer to both these exercises is that it is not possible to design such a chip. One can make compelling geometric arguments why it is not possible, but they require that we visualize simultaneously a large variety of configurations with one picture. We will instead develop a few equations and inequalities relating to planar graphs that will allow us to give convincing arguments that both these designs are impossible.

The Faces of a Planar Drawing

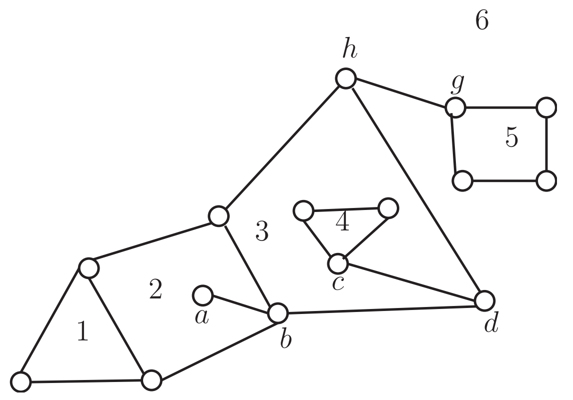
If we assume our graphs are finite, then it is easy to believe that we can draw any edge of a graph as a broken line segment (i.e. a bunch of line segments connected at their ends) rather than a smooth curve. In this way a cycle in our graph determines a polygon in our drawing. This polygon may have some of the graph drawn inside it and some of the graph drawn outside it. We say a subset of the plane is *geometrically connected* if between any two points of the region we can draw a curve.[14] (In our context, you may assume this curve is a broken line segment, but a careful study of geometric connectivity in general situations is less straightforward.) If we remove all the vertices and edges of the graph from the plane, we are likely to break it up into a number of connected sets.

Such a connected set is called a *face* of the drawing if it not a proper subset of any other connected set of the plane with the drawing removed. For example, in Figure 6.33 the faces are

[13]Robin Wilson, *Four Colors Suffice.* Princeton University Press, Princeton NJ 2003.

[14]The usual thing to say is that it is connected, but we want to distinguish this kind of connectivity form graphical connectivity. The fine point about counties that we didn't point out earlier is that they are geometrically connected. If they were not, the graph with a vertex for each county and an edge between two counties that share some boundary line would not necessarily be planar.

Figure 6.33: A typical graph and its faces.



marked 1, a triangular face, 2, a quadrilateral face that has a line segment and point removed for the edge $\{a, b\}$ and the vertex z, 3, another quadrilateral that now has not only a line but a triangle removed from it as well, 4, a triangular face, 5, a quadrilateral face, and 6 a face whose boundary is a heptagon connected by a line segment to a quadrilateral. Face 6 is called the "outside face" of the drawing and is the only face with infinite area. Each planar drawing of a graph will have an *outside face*, that is a face of infinite area in which we can draw a circle that encloses the entire graph. (Remember, we are thinking of our graphs as finite at this point.) Each edge either lies between two faces or has the same face on both its sides.The edges $\{a, b\}$, $\{c, d\}$ and $\{g, h\}$ are the edges of the second type. Thus if an edge lies on a cycle, it must divide two faces; otherwise removing that edge would increase the number of connected components of the graph. Such an edge is called a *cut edge* and cannot lie between two distinct faces. It is straightforward to show that any edge that is not a cut edge lies on a cycle. But if an edge lies on only one face, it is a cut edge, because we can draw a broken line segment from one side of the edge to the other, and this broken line segment plus part of the edge forms a closed curve that encloses part of the graph. Thus removing the edge disconnects the enclosed part of the graph from the rest of the graph.

Exercise 6.5-7 Draw some planar graphs with at least three faces and experiment to see if you can find a numerical relationship among v, the number of vertices, e, the number of edges, and f the number of faces. Check your relationship on the graph in Figure 6.33.

Exercise 6.5-8 In a simple graph, every face has at least three edges. This means that the number of pairs of a face and an edge bordering that face is at least $3f$. Use the fact that an edge borders either one or two faces to get an inequality relating the number of edges and the number of faces in a simple planar graph.

Some playing with planar drawings usually convinces people fairly quickly of the following theorem known as *Euler's Formula*.

Theorem 6.25 (Euler) *In a planar drawing of a graph G with v vertices, e edges, and f faces,*

$$v - e + f = 2.$$

Proof: We induct on the number of cycles of G. If G has no cycles, it is a tree, and a tree has one face because all its edges are cut-edges. Then $v - e + f = v - (v - 1) + 1 = 2$. Now suppose G has $n > 0$ cycles. Choose an edge which is between two faces, so it is part of a cycle. Deleting that edge joins the two faces it was on together, so the new graph has $f' = f - 1$ faces. The new graph has the same number of vertices and one less edge. It also has fewer cycles than G, so we have $v - (e - 1) - (f - 1) = 2$ by the inductive hypothesis, and this gives us $v - e + f = 2$. ■

ForExercise 6.5-8 let's define an edge-face pair to be an edge and a face such that the edge borders the face. Then we said that the number of such pairs is at least $3f$ in a simple graph. Since each edge is in either one or two faces, the number of edge-face pairs is also no more than $2e$. This gives us

$$3f \leq \# \text{ of edge-face pairs} \leq 2e,$$

or $3f \leq 2e$, so that $f \leq \frac{2}{3}e$ in a planar drawing of a graph. We can combine this with Theorem 6.25 to get

$$2 = v - e + f \leq v - e + \frac{2}{3}e = v - e/3$$

which we can rewrite as

$$e \leq 3v - 6$$

in a planar graph.

Corollary 6.26 *In a simple planar graph, $e \leq 3v - 6$.*

Proof: Given above.■

In our discussion of Exercise 6.5-5 we said that we would see a simple proof that the circuit layout problem was impossible. Notice that the question in that exercise was really the question of whether the complete graph on 5 vertices, K_5, is planar. If it were, the inequality $e \leq 3v - 6$ would give us $10 \leq 3 \cdot 5 - 6 = 9$, which is impossible, so K_5 can't be planar. The inequality of Corollary 6.26 is not strong enough to solve Exercise 6.5-6. This exercise is really asking whether the so-called "complete bipartite graph on two parts of size 3," denoted by $K_{3,3}$, is planar. In order to show that it isn't, we need to refine the inequality of Corollary 6.26 to take into account the fact that in a simple bipartite graph there are no cycles of size 3, so there are no faces that are bordered by just 3 edges. You are asked to do that in Problem 13.

Exercise 6.5-9 Prove or give a counter-example: Every planar graph has at least one vertex of degree 5 or less.

Exercise 6.5-10 Prove that every planar graph has a proper coloring with six colors.

In Exercise 6.5-9 suppose that G is a planar graph in which each vertex has degree six or more. Then the sum of the degrees of the vertices is at least 6v, and also is twice the number of edges. Thus $2e \geq 6v$, or $e \geq 3v$, contrary to $e \leq 3v - 6$. This gives us yet another corollary to Euler's formula.

Corollary 6.27 *Every planar graph has a vertex of degree 5 or less.*

Proof: Given above.■

The Five Color Theorem

We are now in a position to give a proof of the five color theorem, essentially Heawood's proof, which was based on his analysis of an incorrect proof given by Kempe to the four color theorem about ten years earlier in 1879. First we observe that in Exercise 6.5-10 we can use straightforward induction to show that any planar graph on n vertices can be properly colored in six colors. As a base step, the theorem is clearly true if the graph has six or fewer vertices. So now assume $n > 6$ and suppose that a graph with fewer than n vertices can be properly colored with six colors. Let x be a vertex of degree 5 or less. Deleting x gives us a planar graph on $n - 1$ vertices, so by the inductive hypothesis it can be properly colored with six colors. However only five or fewer of those colors can appear on vertices which were originally neighbors of x, because x had degree 5 or less. Thus we can replace x in the colored graph and there is at least one color not used on its neighbors. We use such a color on x and we have a proper coloring of G. Therefore, by the principle of mathematical induction, every planar graph on $n \geq 1$ vertices has a proper coloring with six colors.

To prove the five color theorem, we make a similar start. However, it is possible that after deleting x and using an inductive hypothesis to say that the resulting graph has a proper coloring with 5 colors, when we want to restore x into the graph, five distinct colors are already used on its neighbors. This is where the proof will become interesting.

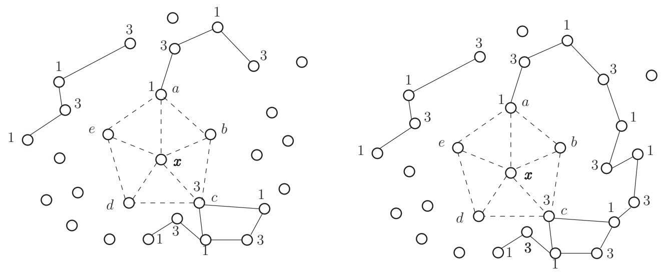
Theorem 6.28 *A planar graph G has a proper coloring with at most 5 colors.*

Proof: We may assume that every face except perhaps the outside face of our drawing is a triangle for two reasons. First, if we have a planar drawing with a face that is not a triangle, we can draw in additional edges going through that face until it has been divided into triangles, and the graph will remain planar. Second, if we can prove the theorem for graphs whose faces are all triangles, then we can obtain graphs with non-triangular faces by removing edges from graphs with triangular faces, and a proper coloring remains proper if we remove an edge from our graph. Although this appears to muddy the argument at this point, at a crucial point it makes it possible to give an argument that is clearer than it would otherwise be.

Our proof is by induction on the number of vertices of the graph. If G has five or fewer vertices then it is clearly properly colorable with five or fewer colors. Suppose G has n vertices and suppose inductively that every planar graph with fewer than n vertices is properly colorable with five colors. G has a vertex x of degree 5 or less. Let G' be the graph obtained by deleting x form G. By the inductive hypothesis, G' has a coloring with five or fewer colors. Fix such a coloring. Now if x has degree four or less, or if x has degree 5 but is adjacent to vertices colored with just four colors in G', then we may replace x in G' to get G and we have a color available to use on x to get a proper coloring of G.

Thus we may assume that x has degree 5, and that in G' five different colors appear on the vertices that are neighbors of x in G. Color all the vertices of G other than x as in G'. Let the five vertices adjacent to x be a, b, c, d, e in clockwise order, and assume they are colored with colors 1, 2, 3, 4, and 5. Further, by our assumption that all faces are triangles, we have that $\{a, b\}$, $\{b, c\}$4,{c,d},$\{d, e\}$, and $\{e, a\}$ are all edges, so that we have a pentagonal cycle surrounding x. Consider the graph $G_{1,3}$ of G which has the same vertex set as G but has only edges with endpoints colored 1 and 3. (Some possibilities are shown in Figure 6.34. We show only edges connecting vertices colored 1 and 3, as well as dashed lines for the edges from x to its neighbors and the edges between successive neighbors. There may be many more vertices and edges in G.)

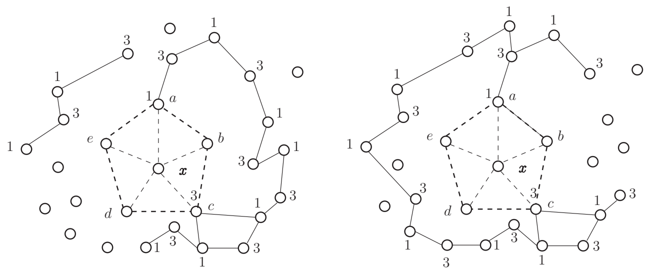
Figure 6.34: Some possibilities for the graph $G_{1,3}$.



The graph $G_{1,3}$ will have a number of connected components. If a and c are not in the same component, then we may exchange the colors on the vertices of the component containing a without affecting the color on c. In this way we obtain a coloring of G with only four colors, 3,2,3,4,5 on the vertices a, b, c, d, e. We may then use the fifth color (in this case 1) on vertex x and we have properly colored G with five colors.

Otherwise, as in the second part of Figure 6.34, since a and c are in the same component of $G_{1,3}$, there is a path from a to c consisting entirely of vertices colored 1 and 3. Now temporarily color x with a new color, say color 6. Then in G we have a cycle C of vertices colored 1, 3, and 6. This cycle has an inside and an outside. Part of the graph can be on the inside of C, and part can be on the outside. In Figure 6.35 we show two cases for how the cycle could occur, one in

Figure 6.35: Possible cycles in the graph $G_{1,3}$.



which vertex b is inside the cycle C and one in which it is outside C. (Notice also that in both cases, we have more than one choice for the cycle because there are two ways in which we could use the quadrilateral at the bottom of the figure.)

In G we also have the cycle with vertex sequence a, b, c, d, e which is colored with five different colors. This cycle and the cycle C can intersect only in the vertices a and c. Thus these two cycles divide the plane into four regions: the one inside both cycles, the one outside both cycles, and the two regions inside one cycle but not the other. If b is inside C, then the area inside both cycles is bounded by the cycle $a\{a, b\}b\{b, c\}c\{c, x\}x\{x, a\}a$. Therefore e and d are not inside the cycle

C. If one of d and e is inside C, then both are (because the edge between them cannot cross the cycle) and the boundary of the region inside both cycles is $a\{a,e\}e\{e,d\}d\{d,c\}c\{c,x\}x\{x,a\}a$. In this case b cannot be inside C. Therefore one of b and d is inside the cycle c and one is outside it. Therefore if we look at the graph $G_{2,4}$ with the same vertex set as G and just the edges connecting vertices colored 2 and 4, the connected component containing b and the connected component containing d must be different, because otherwise a path of vertices colored 2 and 4 would have to cross the cycle C colored with colors 1, 3, and 6. Therefore in G' we may exchange the colors 2 and 4 in the component containing d, and we now have only colors 1, 2, 3, and 5 used on vertices a, b, c, d, and e. Therefore we may use this coloring of G' as the coloring for the vertices of G different from x and we may change the color on x from 6 to 4, and we have a proper five coloring of G. Therefore by the principle of mathematical induction, every finite planar graph has a proper coloring with 5 colors. ■

Kempe's argument that seemed to prove the four color theorem was similar to this, though where we had five distinct colors on the neighbors of x and sought to remove one of them, he had four distinct colors on the five neighbors of x and sought to remove one of them. He had a more complicated argument involving two cycles in place of our cycle C, and he missed one of the ways in which these two cycles can interact.

Important Concepts, Formulas, and Theorems

1. *Graph Coloring.* An assignment of labels to the vertices of a graph, that is a function from the vertices to some set, is called a *coloring* of the graph. The set of possible labels (the range of the coloring function) is often referred to as a set of *colors*.

2. *Proper Coloring.* A coloring of a graph is called a *proper coloring* if it assigns different colors to adjacent vertices.

3. *Intersection Graph.* We call a graph an *intersection graph* if its vertices correspond to sets and it has an edge between two vertices if and only if the corresponding sets intersect.

4. *Chromatic Number.* The *chromatic number* of a graph G, traditionally denoted $\chi(G)$, is the minimum number of colors needed to properly color G.

5. *Complete Subgraphs and Chromatic Numbers.* If a graph G contains a subgraph that is a complete graph on n vertices, then the chromatic number of G is at least n.

6. *Interval Graph.* An intersection graph of a set of intervals of real numbers is called an *interval graph.* The assignment of intervals to the vertices is called an *interval representation.*

7. *Chromatic Number of an Interval Graph.* In an interval graph G, the chromatic number is the size of the largest complete subgraph.

8. *Algorithm to Compute the Chromatic number and a proper coloring of an Interval Graph.* An interval graph G may be properly colored using $\chi(G)$ consecutive integers as colors by listing the intervals of a representation in order of their left endpoints and going through the list, assigning the smallest color not used on an earlier adjacent interval to each interval in the list.

9. *Planar Graph and Planar Drawing.* A graph is called *planar* if it has a drawing in the plane such that edges do not meet except at their endpoints. Such a drawing is called a *planar drawing* of the graph.

10. *Face of a Planar Drawing.* A geometrically connected connected subset of the plane with the vertices and edges of a planar graph taken away is called a *face* of the drawing if it not a proper subset of any other connected set of the plane with the drawing removed.

11. *Cut Edge.* An edge whose removal from a graph increases the number of connected components is called a *cut edge* of the graph. A cut edge of a planar graph lies on only one face of a planar drawing.

12. *Euler's Formula.* Euler's formula states that in a planar drawing of a graph with v vertices, e edges and f faces, $v - e + f = 2$. As a consequence, in a planar graph, $e \leq 3v - 6$.

Problems

1. What is the minimum number of colors needed to properly color a path on n vertices if $n > 1$?

2. What is the minimum number of colors needed to properly color a bipartite graph with parts X and Y.

3. If a graph has chromatic number two, is it bipartite? Why or why not?

4. Prove that the chromatic number of a graph G is the maximum of the chromatic numbers of its components.

5. A *wheel* on n vertices consists of a cycle on $n - 1$ vertices together with one more vertex, normally drawn inside the cycle, which is connected to every vertex of the cycle. What is the chromatic number of a wheel on 5 vertices? What is the chromatic number of a wheel on an odd number of vertices?

6. A *wheel* on n vertices consists of a cycle on $n - 1$ vertices together with one more vertex, normally drawn inside the cycle, which is connected to every vertex of the cycle. What is the chromatic number of a wheel on 6 vertices? What is the chromatic number of a wheel on an even number of vertices?

7. The usual symbol for the maximum degree of any vertex in a graph is Δ. Show that the chromatic number of a graph is no more than $\Delta + 1$. (In fact Brooks proved that if G is not complete or an odd cycle, then $\chi(G) \leq \Delta$. Though there are now many proofs of this fact, none are easy!)

8. Can an interval graph contain a cycle with four vertices and no other edges between vertices of the cycle?

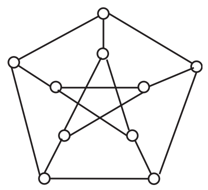
9. The Petersen graph is in Figure 6.36. What is its chromatic number?

10. Let G consist of a five cycle and a complete graph on four vertices, with all vertices of the five-cycle joined to all vertices of the complete graph. What is the chromatic number of G?

11. In how many ways can we properly color a tree on n vertices with t colors?

12. In how many ways may we properly color a complete graph on n vertices with t colors?

Figure 6.36: The Petersen Graph.



13. Show that in a simple planar graph with no triangles, $e \leq 2v - 4$.

14. Show that in a simple bipartite planar graph, $e \leq 2v - 4$, and use that fact to prove that $K_{3,3}$ is not planar.

15. Show that in a planar graph with no triangles there is a vertex of degree three or less.

16. Show that if a planar graph has fewer than twelve vertices, then it has at least one vertex of degree 4.

17. The Petersen Graph is in Figure 6.36. What is the size of the smallest cycle in the Petersen Graph? Is the Petersen Graph planar?

18. Prove the following Theorem of Welsh and Powell. If a graph G has degree sequence $d_1 \geq d_2 \geq \cdots \geq d_n$, then $\chi(G) \leq 1 + max_i[min(d_i, i-1)]$. (That is the maximum over all i of the minimum of d_i and $i - 1$.)

19. What upper bounds do Problem 18 and Problem 7 and the Brooks bound in Problem 7 give you for the chromatic number in Problem 10. Which comes closest to the right value? How close?

Index