CHAPTER 6

A Sampler of Graph Topics

Introduction

A tree is a very important type of graph. For this reason, we've devoted quite a bit of space to them in this text. In Chapter 3, we used decision trees to study the listing, ranking and unranking of functions and to briefly study backtracking. In the next section, we'll focus on "spanning trees." Various types of spanning trees play important roles in many algorithms; however, we will barely touch these applications.

"Graph coloring" problems have been studied by mathematicians for some time and there are a variety of interesting results, some of which we'll discuss. The subject originated from the problem of coloring the countries on a map so that no adjacent countries have the same color. The subject of map colorings is discussed in the Section 6.3, where we consider planar graphs.

If you attempt to draw a graph on a piece of paper, you will often find that you can't do it unless you allow some edges to cross. There are some graphs which can be drawn in the plane without any edges crossing; e.g., all trees. These are called *planar graphs*. Drawing a graph without edges crossing is called *embedding the graph*. K_5, the five vertex complete graph (all edges present) cannot be embedded in the plane. Try it. Can you *prove* that it can't be embedded? It's not clear how to go about *proving* that you haven't somehow missed a clever way to embed it. The impossiblity of embedding K_5 is one of the things that we will prove in Section 6.2.

Aspects of planar graphs of interest to us are coloring, testing for planarity and circuit design. Our discussion of coloring planar maps relies slightly on Section 6.2 and our discussion of a planarity algorithm relies on Section 6.1.

The edges of a graph can be thought of as pipes that hold a fluid. This leads to the idea of *network flows*. We can also interpret the edges as roads, telephone lines, etc. The central practical problem is to maximize, in some sense, the flow through a network. This has important applications in industry. In Section 6.4, we will discuss the underlying concepts and develop, with some gaps, an algorithm for maximizing flow. The theory of flows in networks has close ties with "linear programming" and "matching theory." We will explore the latter briefly.

We can ask what "typical" large graphs are like. For example,

- How many leaves does a typical tree have?
- How many "triangles"—three vertices all joined by edges—does a typical graph have?

- How small can we make the function $q(n)$ and still have most n-vertex, q-edge simple graphs connected?

Questions like these are discussed in Section 6.5.

Finally, we introduce the subject of "finite state machines" in Section 6.6. They provide a theoretical basis for studying questions related to the computability and complexity of functions and algorithms. Various classes of finite state machines are closely associated with various classes of grammars, an association we'll explore briefly in Section 9.2. In Example 11.2 (p. 310), we'll see how some machines provide a method for solving certain types of enumeration problems.

The sections in this chapter are largely independent of each other. Other parts of the book do not require the material in this chapter. If you are not familiar with $\Theta(\ )$ and $O(\ )$ notation, you will need to read Appendix B for some of the examples in this chapter.

6.1 Spanning Trees

Here's the definition of what we'll be studying in this section.

Definition 6.1 Spanning tree *A* **spanning tree** *of a (simple) graph $G = (V, E)$ is a subgraph $T = (V, E')$ which is a tree and has the same set of vertices as G.*

Example 6.1 Since a tree is connected, a graph with a spanning tree must be connected. On the other hand, you were asked to prove in Exercise 5.4.2 (p. 139) that every connected graph has a spanning tree. Thus we have: A graph is connected if and only if it has a spanning tree. It follows that, if we had an algorithm that was guaranteed to find a spanning tree whenever such a tree exists, then this algorithm could be used to decide if a graph is connected. □

In this section, we study minimum weight spanning trees and lineal spanning trees.

Minimum Weight Spanning Trees

Suppose we wish to install "lines" to link various sites together. A site may be a computer installation, a town or a spy. A line may be a digital communication channel, a rail line or a contact arrangement. We'll assume that

- a line operates in both directions;
- it must be possible to get from any site to any other site using lines;
- each possible line has a cost (rental rate, construction costs or likelihood of detection) independent of each other line's cost;
- we want to choose lines to minimize the total cost.

We can think of the sites as vertices V in a graph, the lines as edges E and the costs as a function λ from the edges to the real numbers. Let $T = (V, E')$ be a subgraph of $G = (V, E)$. Define $\lambda(T)$, the *weight* of T, to be the sum of $\lambda(e)$ over all $e \in E'$. Minimizing total cost means choosing T so that $\lambda(T)$ is a minimum. Getting from one site to another means choosing T so that it is connected. It follows that we should choose T to be a spanning tree—if T had more edges than in a spanning tree, we could delete some; if T had less, it would not be connected. (See Exercise 5.4.2 (p. 139).) We call such a T a *minimum weight spanning tree* of (G, λ), or simply of G, with λ understood from context.

How can we find a minimum weight spanning tree T? One approach is to construct T by adding an edge at a time in a greedy way. Since we want to minimize the weight, "greedy" means keeping the weight of each edge we add as low as possible. Here's such an algorithm.

Theorem 6.1 Prim's Algorithm (Minimum Weight Spanning Tree) *Let $G = (V, E)$ be a simple graph with edge weights given by λ. If the following algorithm stops with $V' \neq V$, G has no spanning tree; otherwise, (V, E') is a minimum weight spanning tree for G.*

1. Start: *Let $E' = \emptyset$ and let $V' = \{v_0\}$ where v_0 is any vertex in V.*
2. Possible Edges: *Let $F \subseteq E$ be those edges $\{x, y\}$ with $x \in V'$ and $y \notin V'$. If $F = \emptyset$, stop.*
3. Choose Edge Greedily: *Let $f = \{x, y\}$ be such that $\lambda(f)$ is a minimum over all $f \in F$. Replace V' with $V' \cup \{y\}$ and E' with $E' \cup \{f\}$. Go to Step 2.*

Proof: We begin with the first part; i.e, if the algorithm stops with $V' \neq V$, then G has no spanning tree. Suppose that $V' \neq V$ and that there is a spanning tree. We will prove that the algorithm does not stop at V'. Choose $u \in V - V'$ and $v \in V'$. Since G is connected, there must be a path from u to v. Each vertex on the path is either in V' or not. Since $u \notin V'$ and $v \in V'$, there must be an edge f on the path with one end in V' and one end not in V'. But then $f \in F$ and so the algorithm does not stop at V'.

We now prove that, if G has a spanning tree, then (V, E') is a minimum weight spanning tree. One way to do this is by induction: We will prove that at each step there is a minimum weight spanning tree of G that contains E'.

The starting case for the induction is the first step in the algorithm; i.e., $E' = \emptyset$. Since G has a spanning tree, it must have a minimum weight spanning tree. The edges of this tree obviously contain the empty set, which is what E' equals at the start.

We now carry out the inductive step of the proof. Let V' and E' be the values going into Step 3 and let $f = \{x, y\}$ be the edge chosen there. By the induction hypothesis, there is a minimum weight spanning tree T of G that contains the edges E'. If it also contains the edge f, we are done. Suppose it does not contain f. We will prove that we can replace an edge in the minimum weight tree with f and still achieve minimum weight.

Since T contains all the vertices of G, it contains x and y and, also, some path P from x to y. Since $x \in V'$ and $y \notin V'$, this path must contain an edge $e = \{u, v\}$ with $u \in V'$ and $v \notin V'$. We now prove that removing e from T and then adding f to T will still give a minimum spanning tree.

By the definition of F in Step 2, $e \in F$ and so, by the definition of f, $\lambda(e) \geq \lambda(f)$. Thus the weight of the tree does not increase. If we show that the result is still a tree, this will complete the proof.

The path P together with the edge f forms a cycle in G. Removing e from P and adding f still allows us to reach every vertex in P and so the altered tree is still connected. It is also still a tree because it contains no cycles—adding f created only one cycle and removing e destroyed it. This completes the proof that the algorithm is correct. □

This proof illustrates an important technique for proving that algorithms are correct:

Make an assertion about the algorithm and then prove it inductively.

In this case the assertion was the existence of a minimum weight spanning tree having certain edges. Induction on the number of those edges started easily and the inductive step was not too difficult.

We could construct an even greedier algorithm: At each time add the lowest weight edge that does not create a cycle. The intermediate graphs (V', E') that are built in this way may not be connected; however, if (V, E) was connected, the end result will be a minimum weight spanning tree. We leave it to you to formulate the algorithm carefully and prove that it works in Exercise 6.1.5. This algorithm, with some tricky, efficient handling of the data structures is called *Kruskal's Algorithm.*

Example 6.2 A more general spanning tree algorithm The discussion so far has centered around choosing edges that will be in our minimum weight spanning tree. We could also choose edges that will *not be in* our minimum weight spanning tree. This can be done by selecting a cycle of edges, none of which have been rejected, and then rejecting an edge for which λ is largest among the edges in the cycle. When no more cycles remain, the remaining edges form a minimum weight spanning tree. We leave it to you to prove this. (Exercise 6.1.1)

These ideas can be combined: We have two sets A and R of edges that begin by both being empty. At each step, we somehow add an edge to either A or R and claim that there exists a minimum weight spanning tree that contains all of the edges in A and none of the edges in R. Of course, this will be true at the start. The proof that it is true in general will use induction and will depend on the specific algorithm used for adding edges to A and R.

What sort of algorithms can be built using this idea? We could of course simply use the greedy algorithm for adding edges to A all the time, or we could use the greedier algorithm for adding edges to A all the time, or we could use the cycle approach mentioned at the start of this example to add edges to R. Something new: we could sometimes add edges to A and sometimes to R, whichever is more convenient. This can be useful if we are finding the tree by hand. □

Example 6.3 How fast is the algorithm? We'll analyze the minimum weight (or cost) spanning tree algorithm. Here's a brief description of it. We let $G = (V, E)$ be the given simple graph.

1. Initialize: Select a vertex $v \in V$ and let the tree T consist of v.
2. Select an edge: If there are no edges between V_T (the vertices of T) and $V - V_T$, stop; otherwise, add the one of minimum cost to T and go to Step 2.

Of course, when we add an edge to T, we also add the vertex on it that was not already in T. When the algorithm stops, V_T is the vertex set of the component of G containing v. If G is connected, T is a minimum cost spanning tree.

Suppose that T currently has k vertices. How much work is required to add the next edge to T in the worst case? If the answer is t_k, then the worst case running time is $O(t_1 + \cdots + t_{|V|-1})$. We can't determine t_k since we didn't specify enough details in Step 2. We'll fill them in now. For each vertex u in T, we look at all edges containing it. If $\{u, x\}$ is such an edge, we check to see if $x \in V_T$ and, if not, we check to see if $\{u, x\}$ is the least cost edge found so far in the execution of Step 2. Both these checks can be performed in constant time; i.e, the time does not depend on $|V_T|$ or the size of G. Since we examine at most $|E|$ edges, $t_k = O(|E|)$. Since k ranges from 1 to $|V| - 1$ for a connected graph G, the worst case running time is $O(|V|\,|E|)$.

We cannot say that the worst case running time is $\Theta(|V|\,|E|)$ because we've said "at most" in our argument.

Are there faster algorithms than this? Assuming no one has organized our data according to edge costs, we must certainly look at each edge cost at least once, so any algorithm must have best case running time at least $\Theta(|E|)$. Algorithms with worst case running times $\Theta(|E| \ln\ln |V|)$ and $\Theta(|V|^2)$ are known. If G has a lot of edges, we could have $|E| = \Theta(|V|^2)$ and so $\Theta(|V|^2)$ is $\Theta(|E|)$.

Does this mean that the $\Theta(|V|^2)$ algorithm is best for a graph with about $|V|^2/4$ edges, the typical number in a "random" graph? Not necessarily. To illustrate why not, *suppose* that the running time of the first algorithm is close to $4|E|\,\ln\ln|V|$ and that of the second is close to $3|V|^2$. The first algorithm will be better as long as

$$3|V|^2 \;>\; 4|E|\,\ln\ln|V| \;=\; |V|^2\,\ln\ln|V|.$$

Solving for $|V|$, we obtain $|V| > \exp(e^3) \;=\; 5 \times 10^8$. This means that the $\Theta(|V|^2 \ln\ln|V|)$ algorithm, which is slower when $|V|$ is very large, would actually be faster for all practical values of $|V|$. (Remember, this is *hypothetical* because we assumed the values of the constants in the $\Theta(\cdots)$ expressions.) See Example B.4 (p. 373) for further discussion. □

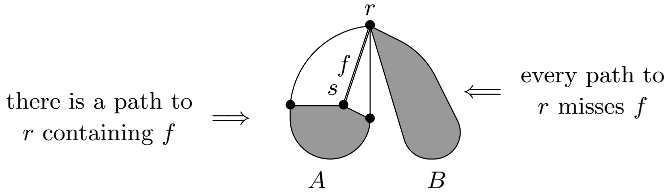


Figure 6.1 An example of the division of vertices for the lineal spanning tree induction. The subgraphs A and B are shaded.

Lineal Spanning Trees

If we simply want to find any spanning tree of G, we can choose any values for the function λ, and use the minimal weight spanning tree algorithm of Theorem 6.1. Put another way, in Step 3 we may choose any edge $f \in F$. Sometimes it is important to restrict the choice of f in some way so that the spanning tree will have some special property other than being minimal.

An important example of such a special property concerns certain rooted spanning trees. To define the trees we are interested in, we borrow some terminology from genealogy.

Definition 6.2 Lineal spanning tree *Let x and y be two vertices in a rooted tree with root r. If x is on the path connecting r to y, we say that y is a* **descendant** *of x. (In particular, all vertices are descendants of r.) If one of u and v is a descendant of the other, we say that $\{u, v\}$ is a* **lineal pair**. *A* **lineal spanning tree** *or* **depth first spanning tree** *of a connected graph $G = (V, E)$ is a rooted spanning tree of G such that each edge $\{u, v\}$ of G is a lineal pair.*

To see some examples of a lineal spanning tree, look back at Figure 5.5 (p. 139). It is the lineal spanning tree of a graph, namely itself. We can add some edges to this graph, for example $\{a, f\}$ and $\{b, j\}$ and still have Figure 5.5 as a lineal spanning tree. On the other hand, if we added the edge $\{e, j\}$, the graph would not have Figure 5.5 as a lineal spanning tree.

How can we find a lineal spanning tree of a graph? That question may be a bit premature—we don't even know when such a tree exists. We'll prove

Theorem 6.2 Lineal spanning tree existence *Every connected graph G has a lineal spanning tree. In fact, given any vertex r of G, there is a lineal spanning tree of G with root r.*

Proof: Our proof will be by induction on the number of vertices in G. The theorem is trivially true for a graph with one vertex. Suppose we know that the claim is true for graphs with less than n vertices. Let $G = (V, E)$ have n vertices and let $f = \{r, s\} \in E$. We will prove that G has a lineal spanning tree with root r.

Let $S \subseteq V$ be those vertices of G that can be reached by a path starting at r and containing the edge f. Note that $r \notin S$ because a path cannot contain repeated vertices; however, $s \in S$. Let $R = V - S$. If $x \in R$, every path from r to x misses s, for if not we could simply go from r to s on f and then follow the path to x.

Let A be the subgraph of G induced by S and let B be the subgraph of G induced by R. (Recall that the subgraph induced by S is the set of all edges in G whose end points both lie in S.) We now prove:

$$\text{Each edge of } G \text{ that does not contain } r \text{ lies in either } A \text{ or } B. \tag{6.1}$$

Suppose we had an edge $\{u, v\}$ with u in A and $v \neq r$ in B. There is a path from r to u using f. By adding v to the path, we conclude that $v \in S$, a contradiction.

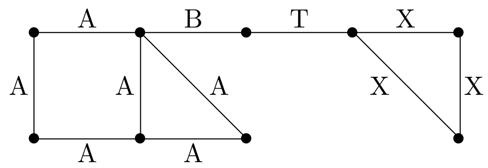
We claim that A is connected and B is connected. Suppose x and y are vertices that are both in A or both in B. Since G is connected, there is a path joining them in G. If an edge of G does not lie in A or in B, then, by (6.1), r is one of its vertices. A path starting in A or B, leaving it and then returning would therefore contain r twice, contradicting the definition of a path. Thus the path lies entirely in A or entirely in B.

Since neither R nor S is empty, each of A and B have less vertices than G. Thus, since A and B are connected, we may apply the induction hypothesis to A and to B. Let $T(A)$ be the lineal spanning tree of A rooted at s and let $T(B)$ be the lineal spanning tree of B rooted at r.

Join $T(A)$ to $T(B)$ by f to produce a connected subgraph T of G with vertex set V and root r. Since $T(A)$ and $T(B)$ have no cycles, it follows that T is a spanning tree of G.

To complete the proof, we must show that T is lineal. Let $e = \{u, v\} \in E$. If one of u and v is r, then e is a lineal pair of T. Suppose that $r \notin e = \{u, v\}$. By (6.1), e lies in A or B. Since $T(A)$ and $T(B)$ are lineal spanning trees, e is a lineal pair of either $T(A)$ or $T(B)$ and hence of T. □

Example 6.4 Bicomponents of graphs Let $G = (V, E)$ be a simple graph For $e, f \in E$ write $e \sim f$ if either $e = f$ or there is a cycle of G that contains both e and f. We claim that this is an equivalence relation. To see what we're talking about, let's look at an example.



6.2

The edges fall into four equivalence classes, which we've arbitrarily called A, B, T and X. Each edge has the letter of its equivalence class next to it. Notice that the vertices *do not* fall into equivalence classes because some of them would have to belong to more than one equivalence class.

Now we'll prove that we have an equivalence relation by using Theorem 5.1 (p. 127). The reflexive and symmetric parts are easy. Suppose that $e \sim f \sim g$. If $e = g$, then $e \sim g$, so suppose that $e \neq g$. Let $e = \{v_1, v_2\}$. Let $C(e, f)$ be the cycle containing e and f and $C(f, g)$ the cycle containing f and g. In $C(e, f)$ there is a path P_1 from v_1 to v_2 that does not contain e. Let x and $y \neq x$ be the first and last vertices on P_1 that lie on the cycle containing f and g. We know that there must be such points because the edge f is on P_1. Let P_2 be the path in $C(e, f)$ from y to x containing e. In $C(f, g)$ there is a path P_3 from x to y containing g. We have shown that P_2 followed by P_3 defines a cycle containing e and g. Hence $e \sim g$.

Since $\sim$ is an equivalence relation on the edges of G, it partitions them. If the partition has only one block, then we say that G is a *biconnected graph.* If E' is a block in the partition, the subgraph of G induced by E' is called a *bicomponent* of G. Note that the bicomponents of G are not necessarily disjoint: Bicomponents may have vertices in common (but *never* edges). The picture (6.2) has four bicomponents.

Finding the bicomponents of a graph is important when we wish to decide if the graph can be drawn in the plane so that no edges cross. We discuss this briefly at the end of Section 6.3.

Biconnectivity is closely related to lineal spanning trees. Suppose T is a lineal spanning tree of G and that the vertices x and y are in the same bicomponent of G. Then either

$$\{x, y\} \text{ is an edge that is a bicomponent by itself}$$

or

$$\text{there is a cycle containing } x \text{ and } y.$$

In either case, since T is a lineal spanning tree, $\{x, y\}$ is a lineal pair. This leads to an algorithm for finding bicomponents: Suppose $e = \{x, y\}$ is an edge that is not in T. If f is an edge on the path from x to y in T, write $e \sim f$. As it stands, $\sim$ is not an equivalence relation; however, it can be made into one by adding what is needed to insure reflexivity, symmetry and transitivity. In the resulting relation it turns out that, $e \sim f$ if and only if e and f are in the same bicomponent. (This requires proof, which we omit.) You might like to experiment with this idea. □

Exercises

6.1.1. A cycle approach to forming a minimum weight spanning tree was discussed in Example 6.2: Throw away largest weight edges that do not disconnect the graph. Prove that it actually leads to a minimum weight spanning tree as follows.

(a) Let T be a minimum weight spanning tree and let e be the first edge that is removed by the algorithm but is contained in T. Prove that T with e deleted consists of two components, T_1 and T_2.

(b) Call any edge in the original graph that has one end in T_1 and one in T_2 a *connector*. Prove that, if f is a connector, then $\lambda(f) \geq \lambda(e)$.

(c) Let T^* be the spanning tree produced by the algorithm. Prove that, if e is added to T^*, then the resulting graph has a cycle containing e and some connector f.

(d) Let f be the edge in (c). Prove that $\lambda(f) \geq \lambda(e)$.

(e) Let f be the edge in (c). Prove that T with e removed and f added is also a minimum weight spanning tree.

*(f) Complete the proof.

6.1.2. Let G be a connected simple graph and let B_1 and $B_2 \neq B_1$ be two bicomponents of G. Prove that B_1 and B_2 have at most one vertex in common.

6.1.3. Let G be a connected simple graph. Let $Q(G)$ be the set of bicomponents of G and let $P(G)$ be the set of all vertices of G that belong to more than one bicomponent; i.e., $P(G)$ is the union of the sets $H \cap K$ over all pairs $H \neq K$ with $H, K \in Q(G)$. Define a simple bipartite graph $\mathcal{B}(G)$ with vertex set $W = P(G) \cup Q(G)$ and $\{u, X\}$ an edge if $u \in P(G)$ and $u \in X \in Q(G)$. (See Exercise 5.3.2 for a definition of bipartite.)

(a) Construct three examples of $\mathcal{B}(G)$, each containing at least four vertices.

(b) Prove that $\mathcal{B}(G)$ is a connected simple graph.

(c) Prove that $\mathcal{B}(G)$ is a tree.
Hint. Prove that a cycle in $\mathcal{B}(G)$ would lead to a cycle in G that involved edges in different bicomponents.

(d) Prove that $P(G)$ is precisely the articulation points of G. (See Exercise 5.3.3 for a definition.)

6.1.4. Using the proof of Theorem 6.1, prove: If λ is an *injection* from E to $\mathbb{R}$ (the real numbers), then the minimum weight spanning tree is unique.

*6.1.5. We will study the greedier algorithm that was mentioned in the text. Suppose the graph $G = (V, E, \varphi)$ has n vertices. From previous work on trees, we know that any spanning tree has $n-1$ edges. Let $g_1, g_2, \ldots, g_{n-1}$ be the edges in the order chosen by the greedier algorithm. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges in any spanning tree of G, ordered so that $\lambda(e_i) \leq \lambda(e_{i+1})$. Our goal is to prove that $\lambda(g_i) \leq \lambda(e_i)$ for $1 \leq i < n$. It follows immediately from this that the greedier algorithm produces a minimum weight spanning tree.

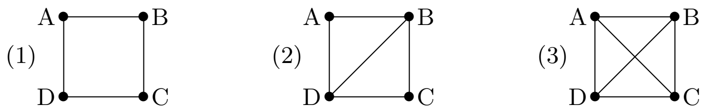
(a) Prove that the vertices of G together with any k edges of G that contain no cycles is a graph with $n-k$ components each of which is a tree.

(b) Let G_k be the graph with vertices V and edges $g_1, \ldots, g_k$. Let H_k be the graph with vertices V and edges $e_1, \ldots, e_k$. Prove that one of the edges of H_{k+1} can be added to G_k to give a graph with no cycles.
Hint. Prove that there is some component of H_{k+1} that contains vertices from more than one component of G_k and then find an appropriate edge in that component of H_{k+1}.

(c) Prove that $\lambda(g_i) \leq \lambda(e_i)$ for $1 \leq i < n$.

6.1.6. Using the result in Exercise 6.1.5, prove that whenever λ is an injection the minimum weight spanning tree is unique.

6.1.7. For each of the following graphs:



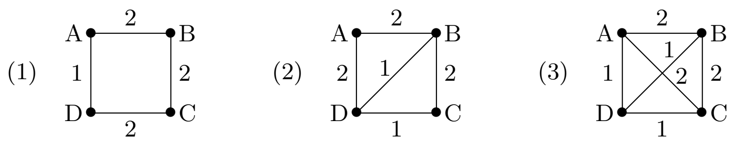
(a) Find all spanning trees.

(b) Find all spanning trees up to isomorphism; that is, find all distinct trees when the vertex labels are removed.

(c) Find all depth-first spanning trees rooted at A.

(d) Find all depth-first spanning trees rooted at B.

6.1.8. For each of the following graphs:



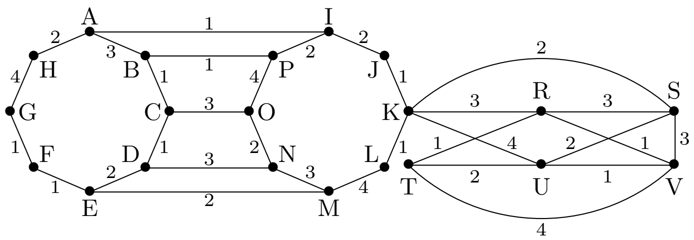
(a) Find all minimal spanning trees.

(b) Find all minimal spanning trees up to isomorphism; that is, find all distinct trees when the vertex labels are removed.

(c) Find all minimal depth-first spanning trees rooted at A.

(d) Find all minimal depth-first spanning trees rooted at B.

6.1.9. In the following graph, the edges are weighted either 1, 2, 3, or 4.



(a) Find a minimal spanning tree using the method of Theorem 6.1.

(b) Find a minimal spanning tree using the method of Example 6.2.

(c) Find a minimal spanning tree using the method of Exercise 6.1.5.

(d) Find a depth-first spanning trees rooted at K.

6.2 Coloring Graphs

Example 6.5 Register allocation Optimizing compilers use a variety of techniques to produce faster code. One obvious way to produce faster code is to keep variables in registers so that memory references are eliminated. Unfortunately, there are often not enough registers available to do this, so choices must be made. For simplicity, assume that the registers and variables are all the same size. Suppose that, by some process, we have gotten a list of variables that we would like to keep in registers.

Can we keep them in registers? If the number of variables does not exceed the number of available registers, we can obviously do it. This sufficient condition is not necessary: We may have two variables that are only used in two separate parts of the program. They could share a register.

This suggests that we can define a binary relation among variables. We could say that two variables are "compatible" if they may share a register. Alternatively, we could say that two variables "conflict" if they cannot share a register. Two variables are either compatible or in conflict, but not both. Thus we can derive one relation from the other and it is rather arbitrary which we focus on. For our purposes, the conflict relation is better.

Construct a simple graph whose vertices are the variables. Two variables are joined by an edge if and only if they conflict. A register assignment can be found if and only if we can find a function λ from the set of vertices to the set of registers such that whenever $\{v, w\}$ is an edge $\lambda(v) \neq \lambda(w)$. (This just says that if v and w conflict they must have different registers assigned to them.) This section studies such ""vertex labelings" λ. □

Definition 6.3 Graph coloring *Let $G = (V, E)$ be a simple graph and C a set. A* ***proper coloring*** *of G using the "colors" in C is a function $\lambda: V \to C$ such that $\lambda(v) \neq \lambda(w)$ whenever $\{v, w\} \in E$.*

Some people omit "proper" and simply refer to a "coloring." A solution to the register allocation problem is a proper coloring of the "conflict graph" of the variables using the set of registers as colors.

Given G and C, we can ask for reasonably fast algorithms to answer various questions about proper colorings:

1. Does there exist a proper coloring of G with C?
2. What is a good way to find a proper coloring of G with C, if one exists?
3. How many proper colorings of G with C are there?

Question 1 could be answered by an algorithm that attempts to construct a proper coloring and fails only if none exist. It could also be answered by calculating the number of proper colorings and discovering that this number is zero. Before trying to answer these questions, let's look at more examples.

Example 6.6 Scheduling problems Register allocation is an example of a simple *scheduling problem*. In this terminology, variables are scheduled for storage in registers. A scheduling problem can involve a variety of constraints. Some of these can be sequential in nature: Problem definition must occur before algorithm formulation, which must in turn occur before division of programming tasks. Others can be conflict avoidance like register allocation. The simplest sort of conflict avoidance conditions are of the form "v and w cannot be scheduled together," which we encountered with register allocation. These can be phrased as graph coloring problems.

Here's an example. Suppose we want to make up a course schedule that avoids obvious conflicts. We could let the vertices of our graph be the courses. Two courses are connected by an edge if we expect a student in one course will want to enroll in the other during the same term. The colors are the times at which courses meet. □

Example 6.7 Map coloring In loose terms, a *map* is a collection of regions, called countries and water, that partition a sphere. (A sphere is the *surface* of a ball.) To make it easy to distinguish regions that have a common boundary, they should be different colors. ("Common boundary" means an actual line segment or curve and not just a single point.) This can be formulated as a graph coloring problem by letting the regions be vertices and by joining two vertices with an edge if the corresponding regions have a common boundary. What problems, if any, are caused by our loose definition of a map? A country may consist of several pieces, like the United States which includes Alaska and Hawaii. This is ruled out in a careful definition of a map.

It is easy to find a map that requires four colors. Try to find such a map yourself. Later we will prove that any map can be colored with five colors. How many colors are needed? From the past few sentences, at least four and at most five. Four colors suffice. This fact is known as the *Four Color Theorem*. At present, the only way this can be proved is by extensive computer calculations. This was done by Appel and Haken in 1976.

Maps can be defined on more complicated surfaces (like a torus—the surface of a doughnut). For each such surface S there is a number $n(S)$ such that some map requires $n(S)$ colors and no map requires more. A fairly simple formula has been found and proved for $n(S)$. It is somewhat amusing that computer calculations are needed only in what appears to be the simplest case—when S is equivalent to a sphere. □

How can we construct a proper coloring of a graph? Suppose we have n vertices and c colors. We could systematically try all c^n possible assignments of colors to vertices until we find one that works or we find that there is no proper coloring. Backtracking on a decision tree can save considerable time. A decision corresponds to assigning a color to a vertex. Suppose that we are at some node t in the decision tree where we have already colored vertices $v_1, \ldots, v_k$ of the graph. The edges leading out of t correspond to the different ways of coloring v_{k+1} so that it is not the same color as any of $v_1, \ldots, v_k$ that are adjacent to it. It is not clear how fast this algorithm is or if we could find a substantially better one.

We'll abandon the construction problem in favor of the counting problem. We will prove

> **Theorem 6.3 Chromatic polynomial existence** *Let G be a simple graph with n vertices. There is a polynomial $P_G(x)$ of degree n such that for each positive integer m, the number of ways to properly color G using $\underline{m}$ is $P_G(m)$.*

$P_G(x)$ is called the *chromatic polynomial* of G. Various properties of it are found in the exercises.

We'll give two proofs of the theorem. The first is simple but it does not provide a useful method for determining $P_G(x)$. The second is more complicated, but the steps of the proof provide a recursive method for calculating chromatic polynomials. We'll explore the method after the proof.

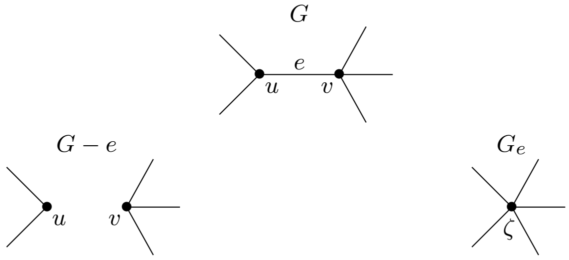


Figure 6.2 Forming $G-e$ and G_e from G by deletion and contraction.

Proof: (Nonconstructive) Let $d_G(k)$ be the number of ways to properly color the graph using k colors such that each color is used at least once. Clearly $d_G(k)=0$ when $k>n$, the number of vertices of G. If we are given x colors, then the number of ways we can use exactly k of them to properly color G is $\binom{x}{k}d_G(k)$. (Choose k colors AND use them.) If $d_G(k)\neq 0$, this is a polynomial in x of degree k because $\binom{x}{k}=x(x-1)\cdots(x-k+1)/k!$, a polynomial in x of degree k. Since the number of colors actually used is between 1 and n,

$$P_G(x) \;=\; \sum_{k=1}^{n}\binom{x}{k}d_G(k), \qquad 6.3$$

a sum of a finite number of polynomials in x. Note that

- $d_G(n)\neq 0$, since it is always possible to color a graph with each vertex colored differently and
- the $k=n$ term in the sum (6.3) is the only term in the sum that has degree n.

Thus, $P_G(x)$ is a polynomial in x of degree n. (We needed to know that there was only one term of degree n because otherwise there might be cancellation, giving a polynomial of lower degree.) □

Proof: (Constructive) Let $G=(V,E)$. We'll use induction on $|E|$, the number of edges in the graph. If the graph has no edges, then $P_G(x)=x^n$ because any function from vertices to colors is acceptable as a coloring in this case.

You may find Figure 6.2 helpful. Suppose $e=\{u,v\}\in E$. Let $G-e=(V,E-e)$, a subgraph of G. This is called *deleting* the edge e. Every proper coloring of G is a proper coloring of $G-e$, but not conversely—a proper coloring λ of $G-e$ is a proper coloring of G if and only if $\lambda(u)\neq\lambda(v)$. A proper coloring λ of $G-e$ with $\lambda(u)=\lambda(v)$ can be thought of as a proper coloring of a graph G_e in which u and v have been identified. This is called *contracting* the edge e. Let's define G_e precisely. Choose $\zeta\notin V$, let $V_e=V\cup\{\zeta\}-\{u,v\}$ and let E_e be all two element subsets of V_e in E together with all sets $\{\zeta,y\}$ for which either $\{u,y\}\in E$ or $\{v,y\}\in E$ or both. The proper colorings of $G_e=(V_e,E_e)$ are in one-to-one correspondence with the proper colorings λ of $G-e$ for which $\lambda(u)=\lambda(v)$—we simply have $\lambda(\zeta)=\lambda(u)=\lambda(v)$. Thus every proper coloring of $G-e$ is a proper coloring of either G or G_e, but not both. By the Rule of Sum, $P_{G-e}(x)=P_G(x)+P_{G_e}(x)$ and so

$$P_G(x) \;=\; P_{G-e}(x)-P_{G_e}(x). \qquad 6.4$$

Since $G-e$ and G_e have less edges than G, it follows by induction that $P_{G-e}(x)$ is a polynomial of degree $|V|=n$ and that $P_{G_e}(x)$ is a polynomial of degree $|V_e|=n-1$. Thus (6.4) is a polynomial of degree n. □

Example 6.8 Some chromatic polynomial calculations What is the chromatic polynomial of the graph with all $\binom{n}{2}$ possible edges present? In this case each vertex is connected to every other vertex by an edge so each vertex has a different color. Thus we get $x(x-1)(x-2)\cdots(x-n+1)$. The graph with all edges present is called the *complete graph* and is denoted by K_n.

What is the chromatic polynomial of the n vertex graph with no edges? We can color each vertex any way we choose, so the answer is x^n. By using the first proof of the theorem, we will obtain another formula for this chromatic polynomial. The graph can be colored using k colors (with each color used) by first partitioning the vertices into k blocks and then assigning a color to each block. By the Rule of Product, $d_G(k) = S(n,k)k!$, where $S(n,k)$ is a Stirling number of the second kind, introduced in Example 1.27. By the first proof and the fact that $P_G(x) = x^n$, we have

$$x^n \;=\; \sum_{k=1}^{n} \binom{x}{k} S(n,k)k! \;=\; \sum_{k=1}^{n} x(x-1)\cdots(x-k+1)\, S(n,k). \qquad 6.5$$

What is the chromatic polynomial of a path containing n vertices? Let the vertices be $\underline{n}$ and the edges be $\{i, i+1\}$ for $1 \le i < n$. Color vertex 1 using any of x colors. If the first i vertices have been colored, color vertex $i+1$ using any of the $x-1$ colors different from the color used on vertex i. Thus we see that the chromatic polynomial of the n vertex path is $x(x-1)^{n-1}$.

We now consider a more complicated problem. What is the chromatic polynomial of the graph that consists of just one cycle? Let n be the length of the cycle and let the answer be $C_n(x)$. Call the graph Z_n. Since Z_2 is just two connected vertices, $C_2(x) = x(x-1)$. It is easy to calculate $C_3(x)$: color one vertex arbitrarily, color the next vertex in a different color and color the last vertex different from the first two. Thus $C_3(x) = x(x-1)(x-2)$. What is $C_4(x)$? We use (6.4). If we delete an edge e from Z_4, we obtain a path on 4 vertices, which we have dealt with. If we contract e, we obtain Z_3, which we have dealt with. Thus

$$C_4(x) \;=\; x(x-1)^3 - x(x-1)(x-2).$$

What is the general formula? The previous argument generalizes to

$$C_n(x) \;=\; x(x-1)^{n-1} - C_{n-1}(x) \quad \text{for } n > 2. \qquad 6.6$$

How can we solve this recursion? This is not so clear. It is easier to see if we repeatedly use Figure 6.2 to expand things out until we obtain paths. The result for Z_5 is shown in the left side of Figure 6.3. In the right side of Figure 6.3, we have replaced each leaf by its chromatic polynomial. We can now work upwards from the leaves to compute the chromatic polynomial of the root.

This is a good way to do it for a graph that has no nice structure. In this case, however, we can write down the chromatic polynomial of the root directly. Notice that the kth leaf from the left (counting the leftmost as 0) has chromatic polynomial $x(x-1)^{n-k-1}$. If k is even it appears in the chromatic polynomial of the root with a plus sign, while if k is odd it appears with a minus sign. This shows that, for $n > 1$,

$$C_n(x) \;=\; \sum_{k=0}^{n-2} (-1)^k x(x-1)^{n-k-1} \;=\; x(x-1)^{n-1} \sum_{k=0}^{n-2} \left(\frac{1}{1-x}\right)^k,$$

which is a geometric series. Thus

$$C_n(x) \;=\; x(x-1)^{n-1} \frac{1 - \left(\frac{1}{1-x}\right)^{n-1}}{1 - \frac{1}{1-x}}.$$

You should show that this simplifies to

$$C_n(x) \;=\; (x-1)^n + (-1)^n (x-1) \qquad 6.7$$

for $n > 1$. □

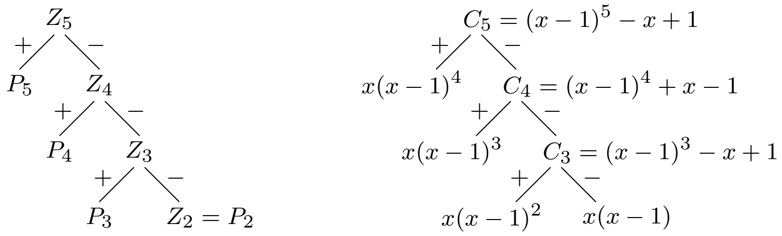


Figure 6.3 The calculation for $C_5(x)$ expanded. P_n is the n vertex path.

Exercises

6.2.1. Give an alternate proof of (6.7) by induction as follows: Prove by substitution that (6.7) satisfies (6.6) and show that (6.7) is correct for $n = 2$.

6.2.2. Conjecture and prove a formula for the chromatic polynomial of a tree. Your formula may include the number of vertices, the degrees of the vertices and anything else that you need. Be sure to indicate how you arrived at your conjecture. This formula can be useful in computing chromatic polynomials by the recursive method.
Hint. There is a simple formula.

6.2.3. The results in this exercise make it easier to calculate some chromatic polynomials using the recursive method.

(a) Suppose that G consists of two graphs H and K with no vertices in common. Prove that $P_G(x) = P_H(x)P_K(x)$.

(b) Suppose that G consists of two graphs H and K sharing exactly one vertex. Prove that $P_G(x) = P_H(x)P_K(x)/x$.

(c) Suppose that G is formed by taking two graphs H and K with no vertices in common, choosing vertices $v \in H$ and $w \in K$, and adding the edge $\{v, w\}$. Express $P_G(x)$ in terms of $P_H(x)$ and $P_K(x)$.

6.2.4. Let n and k be integers such that $1 < k < n-1$. Let G have $V = \underline{n}$ and edges $\{i, i+1\}$ for $1 \le i < n$, $\{1, n\}$, and $\{k, n\}$. Thus G is Z_n with one additional edge. Obtain a formula for $P_G(x)$.

6.2.5. Let L_n be the simple graph with $V = \underline{n} \times \underline{2}$ and $\{(i,j),(i',j')\}$ an edge if and only if $i = i'$ or $j = j'$ and $|i - i'| = 1$. The graph looks somewhat like a ladder and has $3n - 2$ edges. Prove that the chromatic polynomial of L_n is $(x^2 - 3x + 3)^{n-1}x(x-1)$.

6.2.6. Let G be the simple graph with $V = \underline{3} \times \underline{3}$ and $\{(i,j),(i',j')\}$ an edge if and only if $|i-i'|+|j-j'| = 1$. It looks like a 2×2 board. Compute its chromatic polynomial.
Hint. It appears that any way it is done requires some computation. Using the edges joined to $(2,2)$ for deletion and contraction is helpful.

*6.2.7. Construct a graph from the cube by removing the interior and the faces and leaving just the vertices and edges. What is the chromatic polynomial of this graph? (There is not some quick neat way to do this. Quite a bit of careful calculation is involved. A bit of care in selecting edges for removal and contraction will help.)

6.2.8. Give a proof of (6.5) by counting all functions from $\underline{n}$ to $\underline{x}$ in two ways.

6.2.9. Adapt the second proof that $P_G(x)$ is a polynomial of degree n to prove that the coefficients of the polynomial alternate in sign; that is, the coefficient of x^k is a nonnegative multiple of $(-1)^{n-k}$.

6.3 Planar Graphs

Recall that, drawing a graph in the plane without edges crossing is called embedding the graph in the plane. Any graph that can be embedded in the plane can be embedded in the sphere (i.e., the surface of a ball) and vice versa. The idea is simple: Cut a little hole out of the sphere in such a way that you don't remove any of the graph, then, pretending the sphere is a rubber sheet, stretch it flat to form a disc. Conversely, any map on the plane is bounded, so we can cut a disc containing a map out of the plane and curve it around to fit on a sphere. Thus, studying maps on the plane is equivalent to studying maps on the sphere.

Sometimes fairly simple concepts in mathematics lead to a considerable body of research. The research related to planar graphs is among the most accessible such bodies for someone without extensive mathematical training. Here are some of the research highlights and what we'll be doing about them.

1. The earliest is probably Euler's relation, which we'll discuss soon. If the sphere is cut along the edges of an embedded connected graph, we obtain pieces called *faces*. Euler discovered that the number of vertices and faces together differed from the number of edges by a constant. This has been extended to graphs embedded in other surfaces and to generalizations of graphs in higher dimensions. The result is an important number associated with a generalized surface called its *Euler characteristic*.
2. The four color problem has already been mentioned in the section on chromatic polynomials. As noted there, it has been generalized to other surfaces. We'll use Euler's relation to prove that five colors suffice on the plane.
3. A description of those graphs which can be drawn in the plane was obtained some time ago by Kuratowski: A graph is planar if and only if it does not "contain" either
   - K_5, the five vertex complete graph, or
   - $K_{3,3}$, the graph with $V = \{a_1, a_2, a_3, b_1, b_2, b_3\}$ and all nine edges of the form $\{a_i, b_j\}$.

   We say that G contains H if, except for labels we can obtain H from G by repeated use of the three operations:

   (a) delete an edge,

   (b) delete a vertex that lies on no edges and

   (c) if v lies only on the edges $e_1 = \{v, a_1\}$ and $e_2 = \{v, a_2\}$, delete v, e_1 and e_2 and add the edge $\{a_1, a_2\}$.

   Research has developed in two directions. One is algorithmic: Find good algorithms for deciding if a graph is planar and, if so, for embedding it in the plane. We'll discuss the algorithmic approach a bit. The other direction is more theoretical: Develop criteria like Kuratowski's for other surfaces. It has recently been proved that such criteria exist: There is always a finite list of graphs, like K_5 and $K_{3,3}$, that are bad. We will not be able to pursue this here; in fact, we will not even prove Kuratowski's Theorem.
4. Let's allow loops in our graphs. A graph embedded in the plane has a *dual*. A dual is constructed as follows for an embedded graph G. Place a vertex of $D(G)$ in each face of G. Thus there is a bijection between faces of G and vertices of $D(G)$. Every edge e of G is crossed by exactly one edge e' of $D(G)$, and vice versa, as follows. Let the face on one side of e be f_1 and the face on the other side be f_2. (Possibly, $f_1 = f_2$.) If v_1' and v_2' are the corresponding vertices in $D(G)$, then e' connects v_1' and v_2'. Attempting to extend the idea of a dual to other graphs leads to what are called "matroids" or "combinatorial geometries." We won't discuss this subject at all.

The algorithmic subsection does not require the earlier material, but it does require some knowledge of spanning trees, which were discussed in Section 6.1.

Our terminology "G contains H" in Item 3 is not standard. People are likely to say "G contains H homeomorphically." You should note that this is not the same as H being a subgraph of G. Repeated application of Rules (a) and (b) gives a subgraph of G. Conversely, all subgraphs of G can be obtained this way. Rule (c) allows us to contract an edge if it has an endpoint of degree 2. The result is not a subgraph of G. For example, applying (c) to a cycle of length 4 produces a cycle of length 3.

Euler's Relation

We'll state and prove Euler's relation and then examine some of its consequences.

Theorem 6.4 Euler's relation *Let $G = (V, E, \varphi)$ be a connected graph. Suppose that G has been embedded in the plane (or sphere) and that the embedding has f faces. Then*

$$|V| - |E| + f \;=\; 2. \tag{6.8}$$

This remains true if we extend the notion of a graph to allow loops.

Proof: In Exercise 5.4.3 (p. 140) you were asked to prove that $v = e + 1$ for trees. Since cutting along the edges of a tree does not make the plane (or sphere) fall apart, $f = 1$. Thus Euler's relation holds for all trees.

Is there some way we can prove the general result by using the fact that it is true for trees? This suggests induction, but on what should we induct? By Exercise 5.4.2 (p. 139), a tree has the least number of edges of any connected v-vertex graph. Thus we should somehow induct on the number of edges. With care, we could do this without reference to v, but there are better ways. One possibility is to induct on $d = e - v$, where $d \geq -1$. Trees correspond to the case $d = -1$ and so start the induction.

Another approach to the induction is to consider v to be fixed but arbitrary and induct on e. From this viewpoint, our induction starts at $e = v - 1$, which is the case of trees.

The two approaches are essentially the same. We'll take the latter approach.

Let $G = (V, E, \varphi)$ be any connected graph embedded in the plane. From now on, we will work in the plane, removing edges from this particular embedding of G. By Exercise 5.4.2, G contains a spanning tree $T = (V, E')$, say. Let $x \in E - E'$; that is, some edge of G not T. The subgraph G' induced by $E - \{x\}$ is still connected since it contains T.

Let $e' = e - 1$ and f' be the number of edges and faces of G'. By the induction assumption, G' satisfies (6.8) and so $v - e' + f' = 2$. We will soon prove that the opposite sides of x are in different faces of G. Thus, removing x merges these two faces and so $f' = f - 1$. This completes the inductive step.

We now prove our claim that opposite sides of x lie in different faces. Suppose $\varphi(x) = \{a, b\}$. Let P be a path from a to b in T. Adding x to the path produces a cycle C. Any face of the embedding must lie entirely on one side of C. Since one side of x is inside C and the other side of x is outside C, the two sides of x must lie in different faces. $\square$

One interesting consequence of Euler's relation is that it tells us that no matter how we embed a graph $G = (V, E)$ in the plane—and there are often many ways to do so—it will always have $|E| - |V| + 2$ faces. We'll derive more interesting results.

Corollary 1 *If G is a planar connected simple graph with e edges and $v > 2$ vertices, then $e \leq 3v - 6$.*

Proof: When we trace around the boundary of a face of G, we encounter a sequence of vertices and edges, finally returning to our starting position. Call the sequence $v_1, e_1, v_2, e_2, \ldots, v_d, e_d, v_1$. We call d the *degree of the face*. Some edges may be encountered twice because both sides of them are on the same face. A tree is an extreme example of this: Every edge is encountered twice. Thus the degree of the face of a tree with e edges is $2e$. Let f_k be the number of faces of degree k. Since there are no loops or multiple edges, $f_1 = f_2 = 0$. If we trace around all faces, we encounter each edge exactly twice. Thus

$$2e \;=\; \sum_{k \ge 3} k f_k \;\ge\; \sum_{k \ge 3} 3 f_k = 3f$$

and so $f \le 2e/3$. Consequently, $2 = v - e + f \le v - e + 2e/3$. Rearrangement gives the corollary. □

Example 6.9 Nonplanarity of K_5 The graph K_5 has 5 vertices and 10 edges and so $3v - 6 = 9 < e$. Thus, it cannot be embedded in the plane. □

The following result will be useful in discussing coloring.

Corollary 2 *If G is a planar connected simple graph, then at least one vertex of G has degree less than 6.*

Proof: We suppose that the conclusion of the corollary is false and derive a contradiction. Let v_k be the number of vertices of degree k. Since each edge contains two vertices, $2e = \sum k v_k$. Since no vertex has degree less than 6, $v_k = 0$ for $k < 6$ and so

$$2e \;=\; \sum_{k \ge 6} k v_k \ge 6v.$$

Thus $e \ge 3v$, contradicting the previous corollary when $v \ge 3$. If $v \le 3$, there are at most 3 edges, so the result is trivial. □

Exercises

6.3.1. Suppose that G is a planar connected graph with e edges and $v > 2$ vertices and that it contains no cycles of length 3. Prove that $e \ge 2v - 4$.

6.3.2. Prove that $K_{3,3}$ is not a planar graph. ($K_{3,3}$ was defined on page 162.)

6.3.3. We will call a connected graph embedded in the sphere a *regular graph* if all vertices have the same degree, say d_v and all faces have the same degree, say d_f.

(a) If e is the number of edges of the graph, prove that

$$e\left(\frac{2}{d_v} + \frac{2}{d_f} - 1\right) \;=\; 2. \tag{6.9}$$

(b) The possible graphs with $d_v = 2$ are simple to describe. Do it.

(c) By the previous part, we may as well assume that $d_v \ge 3$. Since both sides of (6.9) are positive, conclude that one of d_v and d_f must be 3 and that the other is a most 5.

(d) Draw all embedded regular graphs with $d_v > 2$.

6.3.4. Chemists have discovered a compound whose molecule is shaped like a hollow ball and consisting of **sixty** carbon atoms. It is called *buckminsterfullerene*. (This is *not* a joke.) There is speculation that this and similar carbon molecules may be common in outer space. A chemical compound can be thought of as a graph with the atoms as vertices and the chemical bonds as edges. Thus buckminsterfullerene can be viewed as a graph on the sphere. Because of the properties of carbon, it is reasonable to suppose that each atom is bound to exactly 3 others and that the faces of the associated embedded graph are either hexagons or pentagons. How many hexagons are there? How many pentagons are there?
Hint. One method is to determine the number of edges and then obtain two relations involving the number of pentagons and number of hexagons, one by counting edges and another from Euler's relation.

*6.3.5. A graph can be embedded on other finite surfaces besides the sphere. In this case, there is usually another condition: If we cut along all the edges of the graph, we get faces of the embedded graph and they all look like stretched polygons. This is called *properly* embedding the graph. To see that all embeddings are not proper, consider a torus (surface of a donut). A 3-cycle can be embedded around the torus like a bracelet. When we cut along the edges and straighten out the result, we have a cylinder, not a stretched polygon.

For proper embeddings in any surface, there is a relation like Euler's relation (6.8): $|V| - |E| + f = c$, but the value of c depends on the surface. For the sphere, it is 2.

(a) Properly embed some graph on the torus and compute c for it.

(b) Prove that your value in (a) is the same for all proper embeddings of graphs on the torus.
Hint. Cut around the torus like a bracelet, avoiding all vertices. Fill in each of the holes with a circle, introducing edges and vertices along the cuts.

The Five Color Theorem

Our goal is to prove the five color theorem:

Theorem 6.5 Heawood's Theorem *Every planar graph $G = (V, E)$ can be properly colored with five colors (i.e., adjacent vertices have distinct colors).*

Although four colors are enough, we will not prove that since the only known method is quite technical and requires considerable computing resources. On the other hand, if we were satisfied with six colors, the proof would much easier. We'll begin with it because it lays the foundation for five colors.

Proof: (Six colors) The proof will be by induction on the number of vertices of G.

A graph with at most six vertices can obviously be properly colored: Give each vertex a different color. Thus we can assume G has more than six vertices. We can also assume that G is connected because otherwise each component, which has less vertices than G, could be properly colored by the induction hypothesis. This would give a proper coloring of G.

Let $x \in V$ be a vertex of G with smallest degree. By Corollary 2, $d(x) \leq 5$. Let $G - x$ be the graph induced by $V - \{x\}$. By the induction hypothesis, $G - x$ can be properly 6-colored. Since there are at most 5 vertices adjacent to x in G, there must be a color available to use for x. $\square$

This proof works for proving that G can be properly 5-colored except for one problem: The induction fails if x has degree 5 and the coloring of $G - x$ is such that the 5 vertices adjacent to x in G are all colored differently. We now show two different ways to get around this problem.

Proof: (Five colors, first proof) As noted above, we may assume that $d(x) = 5$. Label the vertices adjacent to x as $y_1, \ldots, y_5$, not necessarily in any particular order. Not all of the y_i's can be joined by edges because we would then have K_5 as a subgraph of G and, by Example 6.9, K_5 is not planar.

Suppose that y_1 and y_2 are not joined by an edge. Erase the edges $\{x, y_j\}$ from the picture in the plane for $j = 3, 4, 5$. Contract the edges $\{x, y_1\}$ and $\{x, y_2\}$. This merges x, y_1 and y_2 into a single vertex which we call y. We now have a graph H in the plane with two less vertices than G. By induction, we can properly 5-color H. Do so.

Color all vertices of G using the same coloring as for H, except for x, y_1 and y_2, which are colored as follows. Give y_1 and y_2 the same color as y. Give x a color different from all the colors used on the four vertices y, y_3, y_4 and y_5. (There must be one since we have five colors available.) Since H was properly colored and $\{y_1, y_2\}$ is not an edge of G, we have properly colored G. □

*Proof: (Five colors, second proof) As noted above, we may assume that $d(x) = 5$. Label the vertices adjacent to x as $y_1, \ldots, y_5$, reading clockwise around x. Properly color the subgraph induced by $V - x$. Let c_i be the color used for y_i. As noted above we may assume that the c_i's are distinct, for otherwise we could simply choose a color for x different from $c_1, \ldots, c_5$.

Call this position in the text HERE for future reference. Let H be the the subgraph of $G - x$ induced by the those vertices that have the same colors as y_1 and y_3. Either y_1 and y_3 belong to different components of H or to the same component. We consider the cases separately.

Suppose y_1 and y_3 belong to different components. Interchange c_1 and c_3 in the component containing y_1. We claim this is still a proper coloring of $G - x$. Why? First, it is still a proper coloring of the component of H containing y_1 and hence of H. Second, the vertices not in H are colored with colors other than c_1 and c_3, so those vertices adjacent to vertices in H remain properly colored. We have reduced this situation to the earlier case since only 4 colors are now used for the y_i.

Suppose that y_1 and y_3 belong to the same component. Then there is a path in H from y_1 to y_3. Add the edges $\{x, y_1\}$ and $\{x, y_3\}$ to the path to obtain a cycle. This cycle can be viewed as dividing the plane into two regions, the inside and the outside. Since y_2 and y_4 are on opposite sides of the cycle, any path joining them must contain a vertex on the cycle. Since all vertices on the cycle except y are colored c_1 or c_3, there is no path of vertices colored c_2 and c_4 in $G - x$ joining y_2 and y_4. Now go back to HERE, using the subscripts 2 and 4 in place of 1 and 3. Since y_2 and y_4 will belong to different components, we will not return to this paragraph. Thus, the proof is complete. □

Exercises

6.3.6. Let G be the simple graph with $V = \underline{7}$ and edge set

$$\begin{aligned} E &= \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\}, \\ &\qquad \{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}. \end{aligned}$$

(a) Embed G in the plane.

(b) The vertices of V have a natural ordering. Thus a function specifying a coloring of V can be written in one-line form. In this one-line form, what is the lexicographically least proper coloring of G when the available "colors" are a, b, c, d, e and f?

(c) What is the lexicographically least proper coloring of G using the colors a, b, c and d?
Notice that the lexicographically least proper coloring in (b) uses five colors but there is another coloring that uses only four colors in (c).

6.3.7. Prove that if G is a planar graph with $V = \underline{5}$, then the lexicographically least proper coloring of G using the colors a, b, c, d and e uses only 4 colors.

6.3.8. Suppose $V = \underline{6}$ and the available colors are a, b, c, d, e. Find a planar graph G with vertex set V such that the lexicographically least proper coloring of G is not a four coloring of G.

*6.3.9. What's wrong with the following idea for showing that 4 colors are enough? By the argument for 5 colors, our only problem is a vertex of degree 5 or less such that the adjacent vertices require 4 colors. Use the idea in second argument in the text. With a vertex of degree 4, the argument can be used as stated in the text with all mention of y_5 deleted.

Now suppose the vertex has degree 5. The argument in the text guarantees that $y_1, \ldots, y_5$ can be colored with 4 colors. Do so. We can still select two vertices that are colored differently and are not adjacent in the clockwise listing. Thus the argument given in the text applies and we can reduce the number of colors needed for $y_1, \ldots, y_5$ from 4 to 3.

*6.3.10. In Exercise 6.3.5 we looked at proper embeddings of graphs in a torus. It can be shown that every embedding (proper or not) of a graph in a torus can be properly colored using at most seven colors. Find a graph embedded in a torus that requires seven colors.
Hint. There is one with just seven vertices.

*Algorithmic Questions

We do not know of an algorithm for 4 coloring a planar graph in a reasonable time. The first proof of the Five Color Theorem leads to a reasonable algorithm for coloring a planar graph with 5 colors. At each step, the number of vertices is decreased by 2. Once the reduced graph is colored, it is a simple matter to color the given graph: Adjust the previous coloring in[1] $O^+(1)$ time by assigning colors to the three vertices x, y_1 and y_2 that were merged to form y. The time required is $O^+(1)$. If $R(n)$ is the maximum time needed to reduce an n vertex planar graph, then The total time is[2]

$$R(n) + R(n-2) + \cdots + O^+(n),$$

where the $O^+(n)$ is due to the roughly $n/2$ times the coloring must be adjusted. Note that $R(n)$ comes primarily from finding a vertex of degree 5 or less. It should be fairly clear to you that $R(n)$ is $O^+(n)$ and so the time for coloring is $O^+(n^2)$. One would expect that using a sophisticated data structure to represent the graph would allow us to improve this time. We will not pursue this here.

We now turn our attention to the problem of deciding if a graph is planar. At first glance, there is no obvious algorithm for doing this except to use Kuratowski's Theorem. This involves a tremendous amount of computer time because there are so many possible choices for the operations (a)–(c) described at the start of Section 6.3. Because of this, it is somewhat surprising that there are algorithms with worst case running times that are $\Theta(|V|)$. We'll examine some of the ideas associated with one such algorithm here but will not develop the complex data structures needed to achieve $\Theta(|V|)$ running time.

To check if a graph is planar, it suffices to merge multiple edges and to check that each connected component is planar. This should be obvious. A bit less obvious is the fact that it suffices to check each biconnected component. (This was defined in Example 6.4 (p. 154).) To see this, note that we can begin by embedding one bicomponent in the plane. Suppose that some subgraph H of the given graph has been already embedded. Any unembedded bicomponent C that shares a vertex v with H can then be embedded in a face of H adjacent to v in such a way that the v of C and the v of H can be merged into one vertex. This process can be iterated. Because of this, we limit our attention to simple biconnected graphs.

---

[1] The notation for $O^+(\,)$ is discussed in Appendix B.
[2] Recall that $f(n) + O^+(n)$ means $f(n)$ plus some function that is in $O^+(n)$.

Definition 6.4 *st*-**labeling** *Let $G = (\underline{n}, E)$ be a simple biconnected graph with $\{s,t\} \in E$, an st-***labeling** *of G is a permutation λ of $\underline{n}$ such that*

(a) $\lambda(s) = 1$,

(b) $\lambda(t) = n$ *and,*

(c) *whenever $1 < \lambda(v) < n$, there are vertices u and w adjacent to v such that $\lambda(u) < \lambda(v) < \lambda(w)$.*

We will soon prove that such a vertex labeling exists and give a method for finding it, but first, we explain how to use it.

Start embedding G in the plane by drawing the edge $\{s,t\}$. Suppose that we have managed to embed the subgraph H_k induced by the vertices with $\lambda(v) < k$ together with t. If $\lambda(x) = k$, then we must embed x in the same face as t. Why is this? By (c), there exist vertices $x = w_1, w_2, \ldots = t$ such that $\{w_i, w_{i+1}\}$ is an edge and $\lambda(w_i)$ is an increasing function of i. Since none of these vertices except t have been embedded, they must all lie in the same face of H_k. Thus we know which face of H_k to put x in. Unfortunately, parts of our embedding of H_k may be wrong in the sense that we cannot now embed x to construct H_{k+1}. How can we correct that?

The previous observation implies that we can start out by placing the vertices of G in the plane so that v is at $(\lambda(v), 0)$. Correcting the problems with H_k can be done by redrawing edges without moving the vertices. There are systematic, but rather complicated, ways of finding a good redrawing of H_k (or proving that none exists if the graph is not planar). We will not discuss them. For relatively small graphs, this can be done by inspection.

We now present an algorithm for finding an st-labeling. The validity of the algorithm implies that such a labeling exists, thus completing the proof of our planarity algorithm. We'll find an injection $\lambda\colon \underline{n} \to \mathbb{R}$ that satisfies (a)–(c) in Definition 6.4. It is easy to construct an st-labeling from such a λ by replacing the kth smallest value of λ with k. Suppose that we specified λ for a subgraph G_Z of G induced by some subset Z of the vertices of G. We assume that $s, t \in Z$ and that λ satisfies (a)–(c) on G_Z. Suppose that $\{x,y\}$ is not in G_Z. Since G is biconnected, there is a cycle in G containing $\{x,y\}$ and $\{s,t\}$. This means that there are disjoint paths in G either

(i) from s to x and from t to y or

(ii) from s to y and from t to x.

If (ii) holds, interchange the meanings of x and y to convert it to (i). Thus we may assume that (i) holds.

Let the paths in (i) be $s = u_1, u_2, \ldots = x$ and $t = w_1, w_2, \ldots = y$. Suppose that u_i and w_j are the last vertices in Z on these two paths. Let v_k be the kth vertex on the path

$$u_i, u_{i+1}, \ldots, x, y, \ldots, w_{j+1}, w_j \qquad 6.10$$

and let m be the total number of vertices on the path. Except for u_i and w_j, none of the vertices in (6.10) are in Z. Add them to Z and define $\lambda(v_k)$ for $k \neq 1, m$ such that λ is still an injection and $\lambda(v_k)$ is monotonic on the path. In other words, if $\lambda(u_i) < \lambda(w_j)$, then λ will be strictly increasing on the path, and otherwise it will be strictly decreasing. Making λ an injection is easy since there are an infinite number of real numbers between $\lambda(u_i)$ and $\lambda(w_j)$. We leave it to you to show that this function satisfies (c) for the vertices that were just added to Z.

Example 6.10 Let G be the simple graph with $V = \underline{7}$ and edge set

$$\begin{aligned} E &= \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\}, \\ &\qquad \{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}. \end{aligned}$$

We'll use the algorithm to construct a 2,5-labeling for it. The labeling λ will be written in two line form with blanks for unassigned values; i.e., for vertices not yet added to Z. To begin with $Z = \{2,5\}$ and

$$\lambda = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ & 1 & & & 7 & & \end{pmatrix}.$$

Let $\{x,y\} = \{2,6\}$. In this case, a cycle is 2,6,5. Thus $s = u_1 = x$, $t = w_1$ and $w_2 = y$. Hence $i = 1$, $j = 1$ and the path (6.10) is $s = x, y, t = 2, 6, 5$. We must choose $\lambda(6)$ between 1 and 7, say 4. Thus

$$\lambda = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ & 1 & & & 7 & 4 & \end{pmatrix}.$$

Let $\{x,y\} = \{3,4\}$. A cycle is $3,4,7,6,5,2$ and the path (6.10) is $2,3,4,7,6$. We must choose $\lambda(x)$, $\lambda(y)$ and $\lambda(7)$ between 1 and 4 and monotonic increasing. We take

$$\lambda = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ & 1 & 2 & 3 & 7 & 4 & 3.5 \end{pmatrix}.$$

Finally, with $\{x,y\}$, a cycle is $2,1,4,7,6,5$ and the path (6.10) is 2,1,4. We take

$$\lambda = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2.5 & 1 & 2 & 3 & 7 & 4 & 3.5 \end{pmatrix}.$$

Adjusting to preserve the order and get a permutation, we finally have

$$\lambda = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 1 & 2 & 4 & 7 & 6 & 5 \end{pmatrix}. \quad \square$$

This algorithm for deciding planarity can be adapted to produce an actual embedding in the plane, if the graph is planar. In practical problems, one usually imposes other constraints on the embedding. For example, if one is looking at pictures of graphs, the embedding should spread the vertices and edges out in a reasonably nice fashion. On the other hand, in VLSI design, one often assumes that the maximum degree of the vertices is at most four and requires that the edges be laid out on a regular grid. Other VLSI layout problems involve vertices which take up space on the plane and various edge constraints. In VLSI design, one would also like to keep the edge lengths as small as possible. A further complication that arises in VLSI is that the graph may not be planar, but we would like to draw it in the plane with relatively few crossings. These are hard problems—often they are "NP-hard," a term discussed in Section B.3 (p. 377).

Exercises

6.3.11. Let G be the simple graph it $V = \underline{7}$ and edge set

$$\begin{aligned} E &= \{\{1,2\},\{1,3\},\{1,4\},\{1,7\},\{2,3\},\{2,4\}, \\ &\qquad \{2,5\},\{2,6\},\{2,7\},\{3,4\},\{4,7\},\{5,6\},\{6,7\}\}. \end{aligned}$$

(a) Use the algorithm in the text to construct a $1,7$-labeling for G.

(b) Use the algorithm in the text to construct a $7,1$-labeling for G.

6.3.12. Construct an st-labeling for K_5. (Since K_5 is completely symmetric, it doesn't matter what vertices you choose for s and t.)

6.3.13. Construct an st-labeling for $K_{3,3}$. (Again, the symmetry guarantees that it doesn't matter which edge is chosen for $\{s,t\}$.)

1	2, 7, 12	11	9, 12, 13, 14	21	20, 22, 23, 24
2	1, 6, 3, 10	12	11, 9, 1	22	21, 20, 3
3	2, 4, 17, 22, 25	13	11, 14, 15	23	21, 24
4	3, 5, 29, 24, 27	14	13, 11	24	23, 21, 4
5	4, 6, 28, 30	15	13, 7	25	6, 26, 27
6	5, 20, 25, 7, 2	16	7, 18, 19, 17	26	3, 25
7	6, 8, 16, 1, 19, 15	17	16, 18, 3	27	4, 25
8	7, 9, 10	18	17, 16	28	5, 29, 30
9	10, 12, 11, 8	19	16, 7	29	4, 28
10	8, 9, 1	20	6, 21, 22	30	5, 28

Figure 6.4 The adjacency lists of a simple graph for Exercise 6.3.16.

6.3.14. We return to Exercise 5.5.11: Suppose that G is a connected graph with no isthmuses. We want to prove that the edges of G can be directed so that the resulting directed graph is strongly connected.

(a) Suppose that G is biconnected and has at least two edges. Use st-labelings to prove that the result is true in this case.

(b) Prove if a graph has no isthmuses, then every bicomponent has at least two edges.

(c) Complete the proof of this exercise.

6.3.15. Prove that a graph is biconnected if and only if it has an st-labeling.
Hint. Given any edge different from $\{s,t\}$, use the properties of an st-labeling to find a cycle containing that edge and $\{s,t\}$.

*6.3.16. $G = (\underline{30}, E)$ is a simple graph given by Figure 6.4. Row k lists the vertices of G that are adjacent to k. Embed G in the plane using the algorithm described in the text. Produce a 5 coloring of G and, if you can, make it a 4 coloring.

6.4 Flows in Networks

We now discuss "flows in networks," an application of directed graphs. Examples of this concept are fluid flowing in pipes, traffic moving on freeways and telephone conversations through telephone company circuits. We'll use fluid in pipes to motivate and interpret the mathematical concepts.

The Concepts

In Figure 6.5 we see a simple directed graph. Note that its edges are labeled; e.g., the directed edge (D_1, P_1) has label 2. Imagine that the directed edges of the graph represent pipes through which a fluid is flowing. A label on an edge represents the rate (measured in liters per second) at which the fluid is flowing along the pipe represented by that edge. We denote this flow rate function by f. Thus, $f(D_1, P_1) = 2$ liters/sec is an example of a value of f in Figure 6.5.

The vertices V in Figure 6.5 are divided into two classes,

$$\mathcal{D} = \{D_1, D_2, D_3, D_4\} \quad \text{and} \quad \mathcal{P} = \{P_1, P_2, P_3, P_4\}.$$

Think of the vertices in $\mathcal{D}$ as depots and those in $\mathcal{P}$ as pumps. Fluid can enter or leave the system at a depot but not at a pump. This corresponds to practical experience with pumps: If the rate at which fluid is flowing into a pump exceeds that at which it is flowing out, the pump will rupture, while, if the inflow is less than the outflow, the pump must be creating fluid.

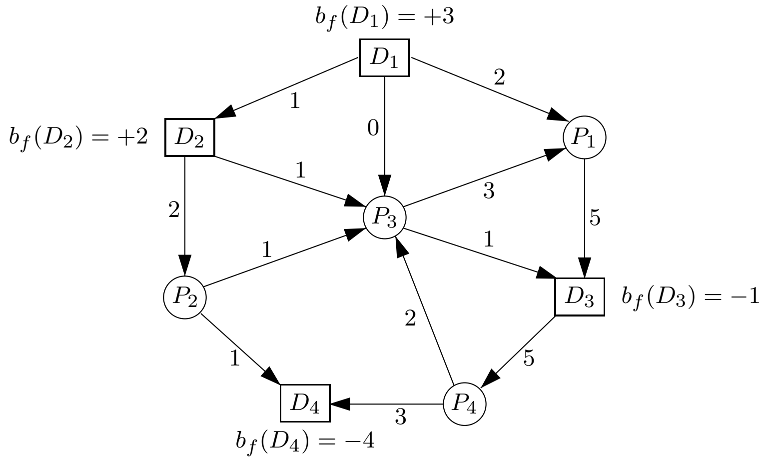


Figure 6.5 A network of pipes with depots D_i, pumps P_i and flow function f (on edges).

We now associate with each vertex $v \in V$ a number $b_f(v)$ which measures the balance of fluid flow at the vertex: $b_f(v)$ equals the sum of all the flow rates for pipes out of v minus the sum of all the flow rates for pipes into v. By the previous paragraph, $b_f(v) = 0$ if $v \in \mathcal{P}$. In Figure 6.5, the nonzero values of b_f are written by the depots. The fact that $b_f(D_2) = +2$ means that we must constantly pump fluid into D_2 from outside the system at a rate of 2 liters/sec to keep the flow rates as indicated. Likewise, we must constantly extract fluid from D_4 at the rate of 4 liters/sec to maintain stability.

It is useful to summarize some of the above ideas with a precise definition. Note that we do not limit ourselves to directed graphs which are simple.

Definition 6.5 Flow in a digraph *Let $G = (V, E, \varphi)$ be a directed graph. For $v \in V$, define*

$$\mathrm{IN}(v) \;=\; \{e \in E : \varphi(e) = (x, v) \text{ for some } x \in V\}$$

and

$$\mathrm{OUT}(v) \;=\; \{e \in E : \varphi(e) = (v, y) \text{ for some } y \in V\}.$$

Let f be a function from E to the nonnegative real numbers; i.e., $f : E \to \mathbb{R}^+$. Define $b_f : V \to \mathbb{R}$ by

$$b_f(v) \;=\; \sum_{e \in \mathrm{OUT}(v)} f(e) \;-\; \sum_{e \in \mathrm{IN}(v)} f(e).$$

Let $(\mathcal{D}, \mathcal{P})$ be an ordered partition of V into two sets. The function f will be called a **flow** *with respect to this partition if $b_f(v) = 0$ for all $v \in \mathcal{P}$. We call the function b_f the* **balance** *of the flow f.*

You may have noticed that our discussion of flows in networks is missing something important, namely the capacities of the pipes to carry fluid. In Figure 6.6, we have included this information. Attached to each edge is a dotted semicircle containing the maximum amount of fluid in liters/sec that can flow through that pipe. This is the *capacity* of the edge (pipe) and is denoted by c; e.g., $c(P_1, D_3) = 6$ in Figure 6.6. The capacity c is a function from E to the set of positive real numbers. We are interested in flows which do not exceed the capacities of the edges. Realistically, it would also be necessary to specify capacities for the pumps and depots. We will do that in the exercises, but for now we'll assume they have a much larger capacity than the pipes and so can handle any flow that the pipes can.

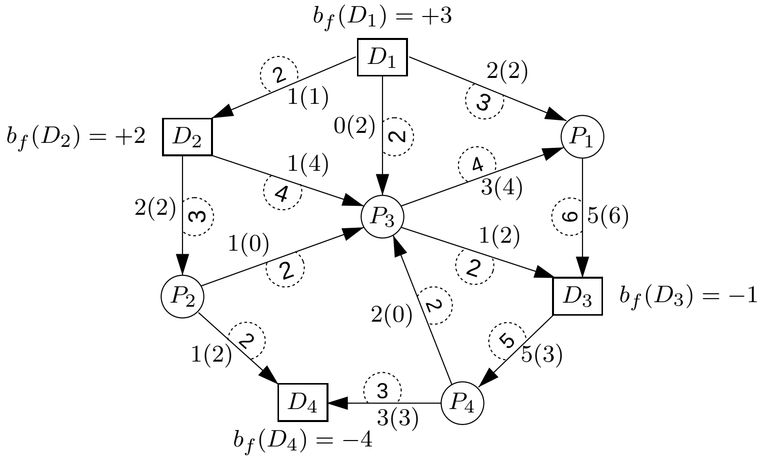


Figure 6.6 The network in Figure 6.5 with capacities c (in dotted semicircles), the flow from Figure 6.5, and another admissible flow p (in parentheses).

The set $\mathcal{D}$ will now be divided into two subsets called the *sources*, where fluid can enter the network, and the *sinks*, where fluid can leave the network. Our goal is to maximize the rate at which fluid passes through the network; that is to maximize the sum of $b_f(v)$ over all sources. We'll present a formal definition and then look at how it applies to Figure 6.6. Since we will spend some time working with Figure 6.6, you may find it useful to make some copies of it for scratch work.

Definition 6.6 Some network flow terminology *Let $G = (V, E, \varphi)$ be a directed graph. Let c be a function from E to the nonnegative reals, called the* **capacity** *function for G. Let f be a flow function on the directed graph $G = (V, E, \varphi)$ with vertex partition $(\mathcal{D}, \mathcal{P})$ and balance function b_f, as defined in Definition 6.5. The flow f will be called* **admissible** *with respect to the capacity function c if $f(e) \leq c(e)$ for all edges e. Let $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$ be an arbitrary ordered partition of $\mathcal{D}$ into two nonempty sets. We call $\mathcal{D}_{\text{in}}$ the set of* **source vertices** *for this partition and $\mathcal{D}_{\text{out}}$ the set of* **sink vertices**. *We define the* **value** *of f with respect to this partition to be*

$$\text{value}(f) \;=\; \sum_{v \in \mathcal{D}_{\text{in}}} b_f(v).$$

An admissible flow f will be called **maximum** *with respect to $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$ if value$(g) \leq$ value(f) for all other admissible flows g.*

In general, the partitioning you are given of the set $\mathcal{D}$ of depots into the two subsets $\mathcal{D}_{\text{in}}$ and $\mathcal{D}_{\text{out}}$ is completely arbitrary. Once this is done, the two sets will be kept the same throughout the problem of maximizing value(f). It is sometimes convenient to write $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ to refer to the graph G with the capacity function c and the "source-sink" partition $(\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}})$. Our basic problem is, given $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$, find an admissible flow that is maximum.

In Figure 6.6 let $\{D_1, D_2\} = \mathcal{D}_{\text{in}}$ and $\{D_3, D_4\} = \mathcal{D}_{\text{out}}$. Intuitively, value$(f)$ is the amount of fluid in liters/sec that must be added to D_1 and D_2 to maintain the flow f. (The same amount overflows at D_3 and D_4.) You have to be careful though! If you just pick some admissible flow f without worrying about maximizing value(f), you might pick one with value$(f) < 0$, in which case fluid will have to be extracted from the source $\mathcal{D}_{in}$ to maintain the flow. It is only for the flow f that maximizes value(f) that we can be sure that fluid is added to the source vertices (or at least not extracted).

An Algorithm for Constructing a Maximum Flow

Now we'll study Figure 6.6 more carefully to help us formulate an algorithm for finding an admissible flow function f which maximizes $\text{value}(f) = b_f(D_1) + b_f(D_2)$, where b_f is the balance function of the function f.

In addition to the capacities shown on the edges of Figure 6.6 there are two sets of numbers. One number has parentheses around it, such as (4) on the edge (D_2, P_3). The other number does not have parentheses, such as 1 on (D_2, P_3). The parenthesized numbers define a flow, which we call p for "parentheses," and the other numbers define a flow which we call f. Referring to the edge $e = (D_2, P_3)$, $f(e) = 1$ and $p(e) = 4$. You should check that f and p satisfy the definitions of a flow with respect to the depots $\mathcal{D}$ and pumps $\mathcal{P}$. Computing the values of f and p with respect to $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ we obtain $\text{value}(f) = 3 + 2 = 5$ and $\text{value}(p) = 5 + 5 = 10$. Thus p has the higher value. In fact, we will later prove that p is a flow of maximum value with respect to the set of sources $\mathcal{D}_{\text{in}}$.

To begin with, concentrate on the flow f. Go to Figure 6.6 and follow the edges connecting the sequence of vertices (D_2, P_3, P_4, D_3). They form an (undirected) path on which you first go forward along the edge (D_2, P_3), then backwards along the edge (P_4, P_3), then backwards along the edge (D_3, P_4). Note that for each forward edge e, $f(e) < c(e)$ and for each backward edge e, $f(e) > 0$. These conditions, $f(e) < c(e)$ on forward edges and $f(e) > 0$ on backward edges are very important for the general case, which we discuss later.

For each forward edge e, define $\delta(e) = c(e) - f(e)$. For each backward edge e, define $\delta(e) = f(e)$. In our particular path we have $\delta(D_2, P_3) = \delta(P_4, P_3) = \delta(D_3, P_4) = 2$. Let δ denote the minimum value of $\delta(e)$ over all edges in the path. In our case $\delta = 2$. We now define a new flow g based on f, the path, and δ:

- For each forward edge e on the path, add δ to $f(e)$ to get $g(e)$.
- For each backward edge e on the path, subtract δ from $f(e)$ to get $g(e)$.
- For all edges e not on the path, $g(e) = f(e)$.

This process is called "augmenting the flow" along the path. You should convince yourself that in our example, $\text{value}(g) = \text{value}(f) + \delta = \text{value}(f) + 2$. This type of relation will be true in general.

If you now study Figure 6.6, you should be able to convince yourself that there is no path from $\mathcal{D}_{\text{in}}$ to $\mathcal{D}_{\text{out}}$ along which p can be augmented. This observation is the key to proving that p is a maximum flow: We will see that maximum flows are those which cannot be augmented in this way.

We are now ready for some definitions and proofs. The next definition is suggested by our previous discussion of augmenting flows.

Definition 6.7 *Let f be an admissible flow function for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. Suppose that $(v_1, v_2, \ldots, v_k)$ is an undirected path in G with $v_1 \in \mathcal{D}_{\text{in}}$ and $v_i \notin \mathcal{D}_{\text{in}}$ for $i > 1$. If for each forward edge e in this path, $f(e) < c(e)$ and for each backward edge e, $f(e) > 0$ then we say that the path is* **augmentable**. *If in addition $v_k \in \mathcal{D}_{\text{out}}$ then we say that the path is a* **complete augmentable path**. *Let $\delta(e) = f(e)$ if e is a backward edge on the path and let $\delta(e) = c(e) - f(e)$ if e is a forward edge. The minimum value of $\delta(e)$ over all edges of the path, denoted by δ, will be called the* **increment of the path**. *Let $\mathcal{A}(f)$ be those vertices that lie on some augmentable path of f, together with all the vertices in $\mathcal{D}_{\text{in}}$.*

In Figure 6.6, with respect to the admissible flow f, the path (D_1, P_1, P_3) is augmentable. The path $(D_2, P_2, P_3, P_4, D_3)$ is a complete augmentable path. So is the path (D_2, P_3, P_4, D_3).

Theorem 6.6 Augmentable Path Theorem *A flow f is a maximum flow if and only if it has no complete augmentable path; that is, if and only if $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}} = \emptyset$.*

Proof: If such a complete augmentable path exists, use it to augment f by δ and obtain a new flow p. Since the first vertex on the path lies in $\mathcal{D}_{\text{in}}$, it follows that $\text{value}(p) = \text{value}(f) + \delta > \text{value}(f)$. Therefore f was not a maximum flow.

Now suppose that no complete augmentable path exists for the flow f. Let $A = \mathcal{A}(f)$ and $B = V - A$. We will now consider what happens to flows on edges between A and B. For this purpose, it will be useful to have a bit of notation. For C and D subsets of V, let $\text{FROM}(C, D)$ be all $e \in E$ with $\varphi(e) \in C \times D$; i.e., the edges from C to D. We claim:

1. For *any* flow g,
$$\text{value}(g) \;=\; \sum_{e \in \text{FROM}(A,B)} g(e) \;-\; \sum_{e \in \text{FROM}(B,A)} g(e). \qquad 6.11$$

2. For the flow f, if $e \in \text{FROM}(A, B)$ then $f(e) = c(e)$, and if $e \in \text{FROM}(B, A)$ then $f(e) = 0$.

The proofs of the claims are left as exercises. Suppose the claims are proved. Since $0 \le g(e) \le c(e)$ for any flow g, it follows that

$$\begin{aligned}\text{value}(g) &= \sum_{e \in \text{FROM}(A,B)} g(e) - \sum_{e \in \text{FROM}(B,A)} g(e)\\ &\le \sum_{e \in \text{FROM}(A,B)} c(e) - \sum_{e \in \text{FROM}(B,A)} 0\\ &= \text{value}(f).\end{aligned}$$

Since g was *any* flow whatsoever, it follows that f is a maximum flow. □

This theorem contains the general idea for an algorithm that computes a maximum flow for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. The first thing to do is to choose an admissible flow f. The flow $f(e) = 0$ for all e will always do. Usually, by inspection, you can do better than that. The general idea is that, given a flow f such that $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}} \ne \emptyset$, we can find a complete augmentable path and use that path to produce a flow of higher value. Here's the procedure

```
/* The main procedure */
Procedure maxflow
     Set f(e) = 0 for all e.
     While A(f) ∩ D_out ≠ ∅, augment(f).
     Return f.
End

/* Replace f with a bigger flow. */
Procedure augment(f)
     Find a complete augmentable path (v_1, v_2, ..., v_k).
     Compute the increment δ of this path.
     If e is a forward edge of the path, set f(e) = f(e) + δ.
     If e is a backward edge of the path, set f(e) = f(e) − δ.
     Return f.
End
```

We have left it up to you to do such nontrivial things as decide if $\mathcal{A}(f) \cap \mathcal{D}_{\text{out}}$ is empty and to find a complete augmentable path. In the examples and exercises in this book it will be fairly easy to do these things. On larger scale problems, the efficiency of the algorithms used for these things can be critical. This is a topic for a course in data structures and/or algorithm design and is beyond the scope of this book.

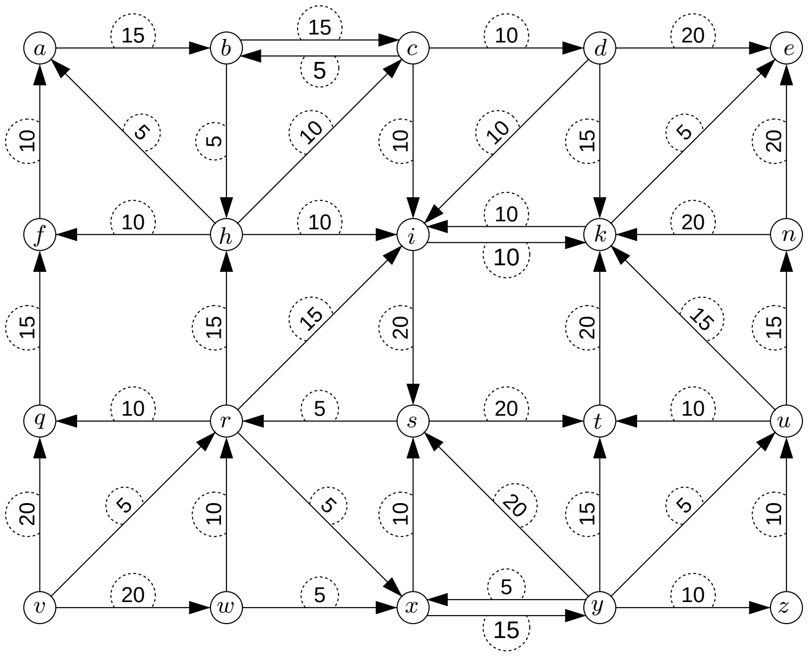


Figure 6.7 The network for Exercise 6.4.1.

It is possible that, like Zeno's paradox, our algorithm will run forever: `augment` could simply produce a flow f with value(f) halfway between the value of the flow it was given and the value of a maximum flow. In fact, the algorithm will always stop. We'll prove a weaker version:

Theorem 6.7 Integer Flow Theorem *If the capacities of a network are all integers, then the* `maxflow` *algorithm stops after a finite number of steps and produces a maximum flow which assigns integer flows to the edges.*

Proof: We claim that the calculations in the algorithm involve only integer values for f and δ. This can be proved by induction: Before any iterations, f is an integer valued function. Suppose that we call `augment`(f) with an integer valued f. Since δ is a minimum of numbers of the form $f(e)$ and $c(e) - f(e)$, which are all integers, δ is a positive integer. Thus the new f is integer valued and has a value at least one larger than the old f. Thus, after n steps, value$(f) \geq n$. If a maximum flow has value F, then a maximum flow is reached after at most F steps. $\square$

Although this algorithm stops, it is a poor algorithm. Quite a bit of work has been done on finding fast network flow algorithms. Unfortunately, improvements usually lead to more complex algorithms that use more complicated data structures. One easy improvement is to use a shortest complete augmentable path in `augment`. This leads to an easily programmed algorithm which often runs fairly quickly. Another poor feature of our algorithm lies in the fact that all the calculations needed to find an augmentable path are thrown away after the path has been found. With just a little more work, one may be able to do several augmentations at the same time. The worst case behavior of the resulting algorithm is good, namely $O(|V|^3)$. Unfortunately, it is rather complicated so we will not discuss it here.

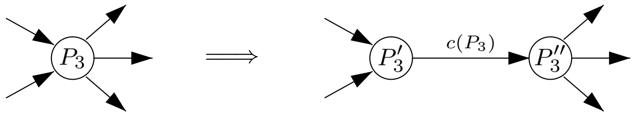


Figure 6.8 Converting pump capacity to edge capacity for Exercise 6.4.4.

Exercises

6.4.1. The parts of this problem all refer to Figure 6.7.

(a) With $\mathcal{D}_{\text{in}} = \{v, w, x, y, z\}$ and $\mathcal{D}_{\text{out}} = \{a, b, c, d, e\}$, find a maximum flow. Also, find a minimum cut set and verify that its capacity is equal to the value of your maximum flow.

(b) Returning to the previous part, find a different maximum flow and a different minimum cut set.

(c) With $\mathcal{D}_{\text{in}} = \{v\}$ and $\mathcal{D}_{\text{out}} = \{e\}$, find a maximum flow. Also, find a minimum cut set and verify that its capacity is equal to the value of your maximum flow.

(d) In the previous part, is the maximum flow unique? Is the minimum cut set unique?

6.4.2. Prove Claim 2 in the proof of the Augmentable Path Theorem.
Hint. Prove that $f(e) < c(e)$ for $e \in \text{FROM}(A, B)$ implies that the ends of e are in A, a contradiction. Do something similar for $\text{FROM}(B, A)$.

6.4.3. Prove (6.11).
Hint. Prove that $\text{value}(g) = \sum_{v \in A} b(v)$ and that each edge e with both ends in A contributes both $c(e)$ and $-c(e)$ to $\sum_{v \in A} b(v)$.

6.4.4. We didn't consider the capacities of the pumps in dealing with Figure 6.6. Essentially, we assumed that the pumps could handle any flow rates that might arise. Suppose that pumps P_1, P_2 and P_3 are having mechanical problems and can only pump 3 liters/sec. What is a maximum flow for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ with $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ and $\mathcal{D}_{\text{out}} = \{D_3, D_4\}$? Figure 6.8 shows how to convert a pump capacity into an edge capacity.

6.4.5. Consider again the network flow problem of Figure 6.6. The problem was defined by $N = (G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$ where $\mathcal{D}_{\text{in}} = \{D_1, D_2\}$ and $\mathcal{D}_{\text{out}} = \{D_3, D_4\}$. Imagine that two new depots are created as shown in Figure 6.9, and all of the original depots are converted into pumping stations. Let $N' = (G', c', (\mathcal{D}'_{\text{in}}, \mathcal{D}'_{\text{out}}))$ denote this new problem, where $\mathcal{D}'_{\text{in}} = \{D_0\}$ and $\mathcal{D}'_{\text{out}} = \{D_5\}$.

(a) What are the smallest values of $c'(D_0, P'_1) = c'_1$, $c'(D_0, P'_2) = c'_2$, $c'(P'_3, D_5) = c'_3$ and $c'(P'_4, D_5) = c'_4$ that guarantees that if p' is a maximum flow for N' then p' restricted to the edges of G is a maximum flow for N? Explain.

(b) With your choices of c'_i, will it be true that any maximum flow on N can be used to get a maximum flow on N'? Explain.

*Cut Partitions and Cut Sets

The "Max-Flow Min-Cut Theorem" is closely related to our augmentable path theorem; however, unlike that theorem, it does not lead immediately to an algorithm for finding the maximum flow. Instead, its importance is primarily in the theoretical aspects of the subject. It is an example of a "duality" theorem, many of which are related to one another. If you are familiar with linear programming, you might like to know that this duality theorem can be proved from the linear programming duality theorem. We need a definition.

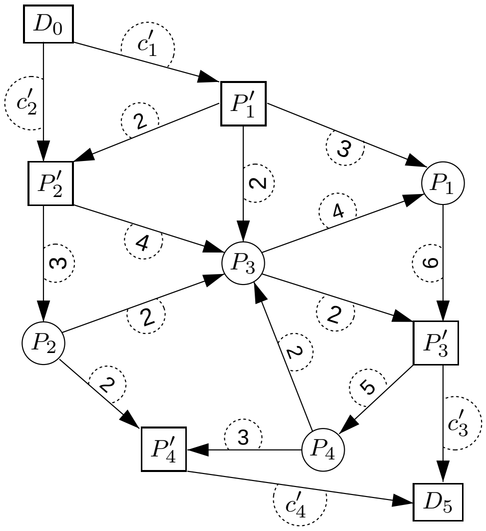


Figure 6.9 New depots for Exercise 6.4.5.

Definition 6.8 Cut partition *Given $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$, any ordered partition (A, B) of V with $\mathcal{D}_{\text{in}} \subseteq A$ and $\mathcal{D}_{\text{out}} \subseteq B$ will be called a* **cut partition**. *A* **cut set** *is a subset F of the edges of G such that every directed path from $\mathcal{D}_{\text{in}}$ to $\mathcal{D}_{\text{out}}$ contains an edge of F. If F is a set of edges in G, the sum of $c(e)$ over all $e \in F$ is called the* **capacity** *of F. If (A, B) is a cut partition, we write $c(A, B)$ instead of $c(\text{FROM}(A, B))$ and call it the capacity of the cut partition.*

The following lemma shows that cut partitions and cut sets are closely related.

Lemma *If (A, B) is a cut partition,* $\text{FROM}(A, B)$ *is a cut set. Conversely, if F is a cut set, then there is a cut partition (A, B) with* $\text{FROM}(A, B) \subseteq F$.

Proof: This is left as an exercise. □

Theorem 6.8 Max-Flow Min-Cut Theorem *Let f be any flow and (A, B) any cut partition for $(G, c, (\mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}))$. Then*

$$\text{value}(f) \leq c(A, B)$$

and, if f is a maximum flow, then there is a cut partition (A, B) such that $\text{value}(f) = c(A, B)$. *The results are valid if we replace the cut partition (A, B) with the cut set F.*

Proof: The inequality $\text{value}(f) \leq c(A, B)$ follows immediately from (6.11) and the fact that f takes on only nonnegative values. Suppose that f is a maximum flow. Let $A = \mathcal{A}(f)$ (and, therefore, $B = V - \mathcal{A}(f)$). It follows from the claim following (6.11) that $\text{value}(f) = c(A, B)$. To change from cut partition to cut set, apply the lemma. □

Why is this called the Max-Flow Min-Cut Theorem? The inequality $\text{value}(f) \leq c(A, B)$ implies that the maximum $\text{value}(f)$ over all possible admissible flows f is less than or equal to minimum value of $c(A, B)$ over all possible cut partitions (A, B). The fact that equality holds for maximum flows and certain cut partitions says that "The maximum value over all flows is equal to the minimum capacity over all cut partitions."

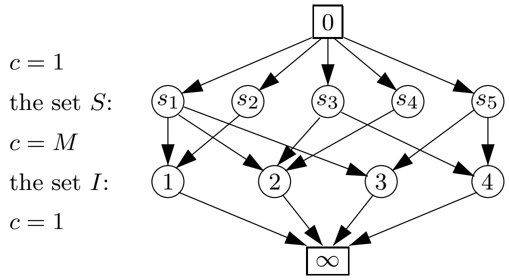


Figure 6.10 The network for $n = 4$, $A_1 = \{s_1, s_2\}$, $A_2 = \{s_1, s_3, s_4\}$, $A_3 = \{s_1, s_5\}$ and $A_4 = \{s_3, s_5\}$. Capacities appear on the left side.

Example 6.11 Systems of distinct representatives Let S be a finite set and suppose that $A_i \subseteq S$ for $1 \le i \le n$. A list $a_1, \ldots, a_n$ is called a system of representatives for the A_i's if $a_i \in A_i$ for all i. If the a_i's are distinct, we call the list a *system of distinct representatives* for the A_i's.

Systems of distinct representatives are useful in a variety of situations. A classical example is the *marriage problem*: There are n tasks and a set S of resources (e.g., employees, computers, delivery trucks). Each resource can be used to carry out some of the tasks; however, we must devote one resource to each task. If A_i is the set of resources that could carry out the ith task, then a system of distinct representatives is an assignment of resources to tasks. An extension of this, which we will not study, assigns a value for each resource-task pair. A higher value means a greater return from the resource-task pairing due to increased speed, capability, or whatever. The *assignment problem* asks for an assignment that maximizes the sum of the values.

We will use the Max-Flow Min-Cut Theorem to prove the following result, which is also called the Philip Hall Theorem and the SDR Theorem.

Theorem 6.9 Marriage Theorem *With the notation given above, a system of distinct representatives (SDR) exists if and only if*

$$\left|\bigcup_{i\in I} A_i\right| \ge |I| \qquad 6.12$$

for every subset of indices $I \subseteq \underline{n}$. In other words, every collection of A_i's contains at least as many distinct a_j's as there are A_i's in the collection.

Proof: By renaming the elements of S if necessary, we may assume that S contains no integers or ∞. Let G be the simple digraph with $V = S \cup \underline{n} \cup \{0, \infty\}$ and edges of three kinds:

- $(0, s)$ for all $s \in S$;
- (i, ∞) for all $i \in \underline{n}$;
- (s, i) for all $s \in S$ and $i \in \underline{n}$ such that $s \in A_i$.

Let all edges of the form (s, i) have capacity M, a *very large* integer and let all other edges have capacity 1. Let $\mathcal{D}_{\text{in}} = \{0\}$ and $\mathcal{D}_{\text{out}} = \{\infty\}$. Such a network is shown in Figure 6.10.

Consider a flow f which is integer valued. Since a vertex $s \in S$ has one edge directed in and that edge has capacity 1, the flow out of s cannot exceed 1. Similarly, since a vertex $i \in \underline{n}$ has one edge directed out and that edge has capacity 1, the flow into i cannot exceed 1. It also follows that f takes on only the values 0 and 1.

We can interpret the edges that have $f(e) = 1$:

- $f(0, s) = 1$ for $s \in S$ means s is used as a representative;
- $f(i, \infty) = 1$ for $i \in \underline{n}$ means A_i has a representative;

- $f(s,i) = 1$ for $s \in S$ and $i \in \underline{n}$ means s is the representative of A_i.

You should convince yourself that this interpretation provides a bijection between integer valued flows f and systems of distinct representatives for *some* of the A_i's, viz., those for which $f(i,\infty) = 1$. To do this, note that for a given $s \in S$, $f(s,i) = 1$ for at most one $i \in \underline{n}$ because the flow into s is at most 1. Experiment with integer flows and partial systems of distinct representatives in Figure 6.10 to clarify this.

From the observation in the previous paragraph, value(f) is the number of sets for which distinct representatives have been found. Thus a system of distinct representatives is associated with a flow f with value(f) $= n$. If we can understand what a minimum capacity cut set looks like, we may be able to use the Max-Flow Min-Cut Theorem to complete the proof.

What can we say about a minimum capacity cut set F? Note that F contains no edges of the form (s,i) because of their large capacity. Thus $c(e) = 1$ for all $e \in F$ and so $c(F) = |F|$. Consequently, we are concerned with the minimum of $|F|$ over all cut sets F containing no edges of the form (s,i). Thus F contains edges of the form $(0,s)$ and/or (i,∞).

Let I be those $i \in \underline{n}$ such that $(i,\infty) \notin F$. What edges of the form $(0,s)$ are needed to form a cut set? If $i \in \underline{n}$ and $s \in A_i$, then we must have an edge from the path $0, s, i, \infty$ in the cut set. Thus, $i \in I$ implies that $(0,s) \in F$. It follows that $(0,s) \in F$ for every $s \in \bigcup_{i \in I} A_i$.

This is enough to form a cut set: Suppose $0, s, i, \infty$ is a path. If $i \notin I$, then $(i,\infty) \in F$. If $i \in I$, then $(0,s) \in \bigcup_{i \in I} A_i \subseteq F$.

What is $|F|$ in this case? (Figure 6.10 may help make the following discussion clearer.) We have $n - |I|$ edges of the form (i,∞) and $\Bigl|\bigcup_{i \in I} A_i\Bigr|$ edges of the form $(0,s)$. Thus $|F|$ is the sum of these and so the minimum capacity is

$$\min_{I \subseteq \underline{n}} \Bigl\{ \Bigl| \bigcup_{i \in I} A_i \Bigr| + n - |I| \Bigr\} \;=\; n + \min_{I \subseteq \underline{n}} \Bigl\{ \Bigl| \bigcup_{i \in I} A_i \Bigr| - |I| \Bigr\}. \qquad 6.13$$

By the Max-flow Min-cut Theorem, a system of distinct representatives will exist if and only if this is at least n. Consequently, the expression in the right hand set of braces of (6.13) must be nonnegative for all $I \subseteq \underline{n}$. $\square$

Exercises

6.4.6. Prove the lemma about cut partitions.

*6.4.7. Prove that for a given max-flow problem, $\mathcal{A}(f)$ is the same for all maximum flows f.

6.4.8. For $r \le n$, and $r \times n$ *Latin rectangle* is an $r \times n$ array in which each row is a permutation of $\underline{n}$ and each column contains no element more than once. If $r = n$, each column must therefore be a permutation of $\underline{n}$. Such a configuration is called a Latin square. The goal of exercise is to prove that it is always possible to add $n - r$ rows to such an $r \times n$ Latin rectangle to obtain a Latin square.

(a) Suppose we are given and $r \times n$ Latin rectangle L with $r < n$. In the notation for systems of distinct representatives, let $S = \underline{n}$ and let A_i be those elements of S that do not appear in the ith column of L. Prove that a system of distinct representatives could be appended to L to obtain and $(r+1) \times n$ Latin rectangle.

(b) Prove that each $s \in S$ appears in exactly $n - r$ of the A_i's.

(c) Use the previous result and $|A_i| = n - r$ to prove that $|A_I| \ge |I|$ and so conclude that a system of distinct representatives exists.

(d) Use induction on $n - r$ to prove that an $r \times n$ Latin rectangle can be "completed" to a Latin square.

6.4.9. The purpose of this exercise is to prove the Marriage Theorem without using flows in networks. The proof will be by induction on n.

(a) Prove the theorem for $n = 1$.

(b) For the induction step, consider two cases, either (6.12) is strict for all $I \neq \underline{n}$ or it is not. Prove the induction step in the case of strictness by proving that we may choose any $a \in A_n$ as the representative of A_n.

(c) Suppose that equality holds in (6.12) for $I \neq \underline{n}$. Let $X = \cup_{i \in I} A_i$ and $B_i = A_i - X$, the set of those elements of A_i which are not in X. Prove that

$$\left| \bigcup_{i \in R} B_i \right| \;\geq\; |R|$$

for all $R \subseteq (\underline{n} - X)$. Use the induction hypothesis twice to obtain a system of distinct representatives.

*6.4.10. Let G be a directed graph and let u and v be two distinct vertices in G. Suppose that (u, v) is not an edge of G. A set of directed paths from u to v in G is called "edge disjoint" if none of the paths share an edge. A set F of edges of G is called an "edge cutset" for u and v if every directed path from u to v in G contains an edge in F. Prove that the cardinality of the largest set of edge disjoint paths equals the cardinality of the smallest edge cutset.
Hint. Make a network of G with source u and sink v.

*6.4.11. State and prove a result like that in Exercise 6.4.10 for graphs.

*6.4.12. Using the idea in Exercise 6.4.4 state and prove results like the two previous exercises with "edge" replaced by "vertex other that u and v" in the definitions of disjoint and cutset. The undirected result is called *Menger's theorem*.

*6.5 Probability and Simple Graphs

Probability theory is used in two different ways in combinatorics.

It can be used to show that, if something is large enough, then it must have some property. For example, it was shown in Example 1.25 (p. 29) that certain error correcting codes must exist if the code words were long enough. Estimates obtained this way are often quite far from best possible. On the other hand, better estimates may be hard to find.

Probability theory is also used to study how random objects behave. For example, what is the probability that a "random graph" with n vertices and $n - 1$ edges is a tree?

What do we mean by random graphs? It may be more instructive to ask "What are some questions asked about random graphs?" Here are some examples, sometimes a bit vaguely stated. You should think of n as large and the graphs as simple.

- What is the probability that a random n-vertex, $(n - 1)$-edge graph is a tree?
- How many edges must an n-vertex random graph have so that it is likely to be connected?
- On average, how many leaves does an n-vertex tree possess and how far is a random tree likely to differ from this average?
- How many colors are we likely to need to color a random n-vertex graph that has kn-edges?
- How can we generate graphs at random so that they resemble the graph of connections of computers on the internet?

To answer questions like these, we must be clear on what is meant by "random" and "likely". Sometimes this can get rather technical; however, there are simple examples.

To begin, we usually consider a set S_n of (simple) graphs with n-vertices and make it into a probability space. An obvious way to do this is with the uniform probability. For example, let S_n be the set of $(n-1)$-edge simple graphs with vertex set $\underline{n}$ and let Pr be the uniform distribution. What is the probability that a random such graph is a tree? By Exercise 5.1.2 (p. 124), there are $\binom{N}{k}$ k-edge graphs, where $N = \binom{n}{2}$. Since a tree has $n-1$ edges, we want $k = n-1$. By Example 5.10 (p. 143), there are n^{n-2} trees. Thus the answer is $n^{n-2} \big/ \binom{N}{n-1}$.

Example 6.12 Graphs with few edges Suppose a graph has few edges. What are some properties we can expect to see?

To answer such questions, we need a probability space. Let $\mathcal{G}(n,k)$ be the probability space gotten by taking the uniform probability on the set of k-edge simple graphs with vertex set $\underline{n}$. For convenience, let $N = \binom{n}{2}$.

If our graph has $n-1$ edges, it could be a tree; however, we'll show that this is a rare event. (You were asked to estimate this probability in Exercise 5.5.15(b). We'll do it here.) The number of such graphs is

$$\begin{aligned}
\binom{N}{n-1} &= \frac{N(N-1)\cdots(N-(n-1)+1)}{(n-1)!} > \frac{\big(N-(n-1)\big)^{n-1}}{(n-1)!} \\
&= \frac{\big((n-2)(n-1)/2\big)^{n-1}}{(n-1)!} \\
&\sim \frac{\big((n-2)(n-1)/2\big)^{n-1}}{\sqrt{2\pi(n-1)}\big((n-1)/e\big)^{n-1}} \qquad \text{by Stirling's formula} \\
&> \frac{\big(e(n-2)/2\big)^{n-1}}{\sqrt{2\pi n}}.
\end{aligned}$$

Since there are n^{n-2} trees, for large n the probability that a random graph in $\mathcal{G}(n, n-1)$ is a tree is less than

$$\frac{n^{n-2}\sqrt{2\pi n}}{\big(e(n-2)/2\big)^{n-1}} = \sqrt{2\pi/n}\,(2/e)^{n-1}\left(1+\frac{2}{n-2}\right)^{n-1}. \qquad 6.14$$

From calculus, $\lim_{x\to 0}(1+x)^{a/x} = e^a$. With $x = \frac{2}{n-2}$ and $a = 2$, we have $\left(1+\frac{2}{n-2}\right)^{n-2} \sim e^2$. Since $2/e < 1$, (6.14) goes to zero rapidly as n gets large. Thus trees are rare.

You should be able to show that a simple graph with n vertices and $n-1$ edges that is not a tree is not connected and has cycles. That leads naturally to two questions:

- How large must k be so that most graphs in $\mathcal{G}(n,k)$ have cycles?
- How large must k be so that most graphs in $\mathcal{G}(n,k)$ are connected?

The first question will be looked at some in the exercises. We'll look at the second question a bit later. $\square$

Of course, the uniform distribution is not the only possible distribution, but why would anyone want to choose a different one? Suppose we are studying graphs with n vertices and k edges. The fact that the number of edges is fixed can be awkward. For example, suppose u, v and w are three distinct vertices. If we know whether or not $\{u,v\}$ is an edge, this information will affect the probability that $\{u,w\}$ is an edge:

- If $\{u,v\}$ is not an edge, there must be k edges among the remaining $N-1$ possible edges, so the probability that $\{u,w\}$ is an edge is equal to $\frac{k}{N-1}$.
- If $\{u,v\}$ is an edge, there must be $k-1$ edges among the remaining $N-1$ possible edges, so the probability that $\{u,w\}$ is an edge is equal to $\frac{k-1}{N-1}$.

Here is a way to avoid that. For each $e \in \mathcal{P}_2(\underline{n})$, make the set $\{$"$e \in G$", "$e \notin G$"$\}$ into a probability space by setting $\Pr(e \in G) = p$ and $\Pr(e \notin G) = 1-p$. We can think of "$e \in G$" as the event in which e is in a randomly chosen graph. Let $\mathcal{G}_p(n)$ be the product of these probability spaces over all $e \in \mathcal{P}_2(\underline{n})$. Let $X(G)$ be the number of edges in G. It can be written as a sum of the N independent random variables

$$X_e(G) \;=\; \begin{cases} 1 & \text{if } e \in G, \\ 0 & \text{if } e \notin G. \end{cases} \qquad 6.15$$

By independence, $\mathbf{E}(X) = pN$ and $\mathbf{var}(X) = Np(1-p)$ because $p(1-p)$ is the variance of a (0,1)-valued random variable whose probability of being 1 is p. With $p = k/N$, we expect to see k edges with variance $kp(1-p) < k$. By Chebyshev's inequality (C.3) (p. 385)

$$\Pr(|X-k| > Ck^{1/2}) \;<\; 1/C^2.$$

Thus, with $C = k^{1/3}$, dividing $|X-k| > k^{1/6}$ by k, and using $\Pr(A') = 1 - \Pr(A)$, we have

$$\Pr\left(\frac{|X-k|}{k} \le k^{-1/6}\right) \;>\; 1 - k^{-2/3}. \qquad 6.16$$

Since $(|X-k|/k) \times 100$ is the percentage deviation of the X from k, (6.16) tells us that this deviation is very likely to be small when k is large.[3]

Because a random graph in $\mathcal{G}_p(n)$ has very nearly pN edges, results for $\mathcal{G}_p(n)$ with $p = k/N$ almost always hold for $\mathcal{G}(n,k)$ as well. Since $\mathcal{G}_p(n)$ is usually easier to study than $\mathcal{G}(n,k)$, people often study it.

If we want to consider all graphs with vertex set $\underline{n}$ with each of the 2^N graphs equally likely, we simply study $\mathcal{G}_{1/2}(n)$ because any particular graph with q edges occurs with probability

$$(1/2)^q(1-1/2)^{N-q} \;=\; (1/2)^N \;=\; 2^{-N},$$

a value that is the same for all n-vertex graphs.

[3] For those familiar with the normal approximation to the binomial distribution, the number of edges is binomially distributed. Using this, one can avoid using Chebyshev's inequality and derive stronger results than (6.16).

Example 6.13 The clique number of a random graph A *clique* in a graph G is a subgraph H such that every pair of vertices of H is connected by an edge. The size of the clique is the number of vertices of H. The *clique number* of a graph is the size of its largest clique. A k-vertex clique is called a k-clique.

What can we say about the clique number of a random graph; that is, a graph chosen using $\mathcal{G}_{1/2}(n)$? We'll get an upper bound on this number for most graphs.

Notice that if a graph contains a K-clique, then it contains an k-clique for all $k \le K$. Thus if a graph does not contain a k-clique, its clique number must be less than k. If we can show that most n-vertex graphs do not have a k-clique, it will follow that the clique number of most n-vertex graphs is less than k.

We'll begin by looking at the expected number of k-cliques in an n-vertex graph. When $W \subseteq \underline{n}$, let X_W be 1 if the vertices W form a clique and 0 otherwise. The probability that $X_W = 1$ is $(1/2)^{\binom{|W|}{2}}$ since there are $\binom{|W|}{2}$ pairs of vertices in W, each of which must be connected by an edge to form a clique. Edges that do not connect two vertices in W don't matter—they can be present or absent. Summing over all k-element subsets of $\underline{n}$, we have

$$\begin{aligned}
\mathbf{E}(\text{number of } k\text{-cliques}) &= \mathbf{E}\Bigg(\sum_{\substack{W\subset\underline{n}\\|W|=k}} X_W\Bigg)\\
&= \sum_{\substack{W\subset\underline{n}\\|W|=k}} \mathbf{E}(X_W) \;=\; \sum_{\substack{W\subset\underline{n}\\|W|=k}} \Pr(X_W=1)\\
&= \sum_{\substack{W\subset\underline{n}\\|W|=k}} (1/2)^{\binom{k}{2}} \;=\; \binom{n}{k}2^{-\binom{k}{2}}.
\end{aligned}$$

Since the number of k-cliques in a graph is a nonnegative integer,

$$\begin{aligned}
\Pr(\text{at least one } k\text{-clique}) &= \sum_{j>0} \Pr(\text{exactly } j\ k\text{-cliques})\\
&\le \sum_{j\ge 0} j\ \Pr(\text{exactly } j\ k\text{-cliques}) \;=\; \mathbf{E}(\text{number of } k\text{-cliques})\\
&= \binom{n}{k}2^{-\binom{k}{2}} \;\le\; \frac{n^k}{k!}2^{-\binom{k}{2}} \;=\; \frac{2^{k(\log_2 n - k/2)}}{2^{-k/2}k!}.
\end{aligned}$$

Since $2^{-k/2}k!$ is large when k is large, the probability of a k-clique will be small when $k/2 \ge \log_2 n$. Thus almost all graphs have clique number less than $2\log_2 n$.

What can be said in the other direction? A lower bound is given in the Exercise 6.5.6. □

Example 6.14 Triangles in random graphs Using $\mathcal{G}_p(n)$ for our probability space, we want to look at the number of triangles in a random graph. For $u, v, w \in \underline{n}$, let

$$X_{u,v,w} = \begin{cases} 1 & \text{if the edges } \{u,v\},\ \{u,w\},\ \{v,w\} \text{ are present,}\\ 0 & \text{otherwise.}\end{cases}$$

We claim $\Pr(X_{u,v,w}=1) = p^3$. How can we see this? Intuitively, each edge has probability p of being present and they are independent, so we get p^3. More formally, since our probability space is a product space,

$$X_{u,v,w} = X_{\{u,v\}}X_{\{u,w\}}X_{\{v,w\}} = p^3,$$

where the $X_{x,y}$ are the random variable defined in (6.15).

Thus the expected value of $X_{u,v,w}$ is p^3. Since there are $\binom{n}{3}$ choices for $\{u,v,w\}$, the expected number of triangles in $\binom{n}{3}p^3$.

Let's compute the variance in the number of triangles. We have to be careful: Triangles are not independent because they may share edges. The safest approach is to define a random variable T that equals the number of triangles:

$$T = \sum_{t \in \mathcal{P}_3(\underline{n})} X_t, \quad \text{where} \quad X_{\{u,v,w\}} = X_{u,v,w}.$$

Since $\mathbf{var}(T) = \mathbf{E}(T^2) - \mathbf{E}(T)^2$ and we know $\mathbf{E}(T) = \binom{n}{3}p^3$, we need to compute $\mathbf{E}(T^2)$. It equals $\sum \mathbf{E}(X_s X_t)$, the sum ranging over all $s, t \in \mathcal{P}_3(\underline{n})$. There are three cases to consider:

- $s = t$ There are $\binom{n}{3}$ terms like this and $\mathbf{E}(X_s X_t) = \mathbf{E}(X_s^2) = \mathbf{E}(X_s) = p^3$.
- $|s \cap t| = 2$ There are $\binom{n}{3}\binom{3}{2}\binom{n-3}{1}$ terms like this since we have $\binom{n}{3}$ choices for s, $\binom{3}{2}$ ways to select two elements of s to include in t, and $\binom{n-3}{1}$ ways to complete t. Since there are a total of five edges in the two triangles, $\mathbf{E}(X_s X_t) = p^5$.
- $|s \cap t| < 2$ Since there are a total of $\binom{n}{3}^2$ terms in the sum $\sum \mathbf{E}(X_s X_t)$ and we have dealt with $\binom{n}{3} + \binom{n}{3}3(n-3)$ terms already, there are

$$\binom{n}{3}^2 - \binom{n}{3} - \binom{n}{3}3(n-3)$$

remaining. Since there are six edges in the two triangles, $\mathbf{E}(X_s X_t) = p^6$. Putting all this together

$$\begin{aligned}\mathbf{var}(T) &= \left(\binom{n}{3}^2 - \binom{n}{3} - \binom{n}{3}3(n-3)\right)p^6 + \binom{n}{3}3(n-3)p^5 + \binom{n}{3}p^3 - \left(\binom{n}{3}p^3\right)^2 \\ &= \binom{n}{3}3(n-3)p^5(1-p) + \binom{n}{3}p^3(1-p^3).\end{aligned}$$

Now we want to use Chebyshev's inequality to find out when most graphs have at least one triangle. Chebyshev's inequality (C.3) (p. 385) is

$$\Pr\Big(|T - \mathbf{E}(T)| > t\sqrt{\mathbf{var}(T)}\Big) < 1/t^2.$$

If $T = 0$, then $|T - \mathbf{E}(T)| = \mathbf{E}(T) > \mathbf{E}(T) - 1$. If we set $t\sqrt{\mathbf{var}(T)} = \mathbf{E}(T) - 1$, Chebyshev's inequality tells us the probability that there is no triangle is $1/t^2$, a number that we want to be small. Solving $t\sqrt{\mathbf{var}(T)} = \mathbf{E}(T) - 1$ and putting it all together with our previous calculations:

$$\Pr(T{=}0) < \frac{\mathbf{var}(T)}{(\mathbf{E}(T)-1)^2} = \frac{\binom{n}{3}3(n-3)p^5(1-p) + \binom{n}{3}p^3(1-p^3)}{\left(\binom{n}{2}p^3 - 1\right)^2}.$$

When p is such that p is small and $\binom{n}{3}p^3$ is large, we have $\binom{n}{3}p^3 \approx n^3p^3/6$ and so we are assuming that $L = np$ is large. Using this,

$$\frac{\mathbf{var}(T)}{(\mathbf{E}(T)-1)^2} \approx \frac{n^4p^5/2 + n^3p^3/6}{n^6p^6/36} = 6\frac{3Lp+1}{L^3} = 18p/L^2 + 6/L^3,$$

We've shown that, if np is large and p is small, then a random graph in $\mathcal{G}_p(n)$ almost certainly contains a triangle. If we let p be larger, then edges are more likely and so triangles are more likely. This we don't need "p is small." In summary, If np is large then a random graph in $\mathcal{G}_p(n)$ almost certainly contains a triangle. $\square$

Example 6.15 Growing random graphs Imagine starting out with vertices $V = \underline{n}$ and then growing a simple graph by randomly adding edges one by one. We'll describe the probable growth of such a graph.

In the first stage, we have a lot of isolated vertices (vertices on no edges) and pairs of vertices joined by an edge. As time goes by (more edges added), the single edges join up forming lots of small trees which continue to grow. Next the trees start developing cycles and so are no longer trees. Of course there are still a lot of isolated vertices and small trees around. Suddenly a threshold is passed and, in the blink of an eye, we have one large (connected) component and lots of smaller components, most of which are trees and isolated vertices. The large component starts "swallowing" the smaller ones, preferring to swallow the larger ones first. Finally, all that is left outside the large component is a few isolated vertices which are swallowed one by one and so the graph is connected. Growth continues beyond this point, but we'll stop here.

When does the graph get connected?

We can't answer this question; however, we can easily compute the expected number of isolated vertices in a random graph. When this number is near zero, we expect most random graphs to be connected. When it is not near zero, we expect a significant number of random graphs to still contain isolated vertices. We'll study this with the $\mathcal{G}_p(n)$ model. This isn't quite the correct thing to do since we're adding edges one by one. However, it's harder to use $\mathcal{G}(n,k)$ and we're not planning on proving anything—we just want to get an idea of what's true.

In $\mathcal{G}_p(n)$, a vertex v will be isolated if none of the possible $n-1$ edges connecting it to the rest of the graph are present. Thus the probability that v is isolated is $(1-p)^{n-1}$. Since there are n vertices, the expected number of isolated vertices is $n(1-p)^{n-1}$. When p is small, $1-p \approx e^{-p}$ and so the expected number of isolated vertices is about $ne^{-p(n-1)} = e^{\ln n - p(n-1)}$. This number will be near zero if $p(n-1) - \ln n$ is a large positive number. This is the same as $pn - \ln n$ being large and positive. In this case, isolated vertices are unlikely. In other words, they've all probably been swallowed by the big component and the graph is connected. On the other hand, if $pn - \ln n$ is large and negative, $e^{\ln n - p(n-1)}$ will be large. In other words, we expect a lot of isolated vertices. Thus $p \approx \frac{\ln n}{n}$ is the critical point when a graph becomes connected. Since we expect about $\binom{n}{2}p \approx \frac{n \ln n}{2}$ edges, a graph should become connected when it has around $\frac{n \ln n}{2}$ edges. $\square$

So far we've studied random graphs and asked what can be expected. Now we'll look at a different problem: we want to *guarantee* that something must happen in a graph.

Example 6.16 Bipartite subgraphs A graph (V', E') is *bipartite* if its vertices can be partitioned into two sets V_1' and $V_2' = V' - V_1'$ such that the edges of the graph only connect vertices in V_1' and V_2'. In other words, if $\{x,y\} \in E'$, then one of x, y is in V_1' and the other is in V_2'.

Given a graph $G = (V,E)$ with n vertices and k edges, we want to find a bipartite subgraph with as many edges as possible. How many edges can we guarantee being able to find? Since we want as many edges a possible, we may as well use all the vertices in G. Thus our bipartite graph will be (V, E') where V is partitioned into V_1' and V_2' and $E' \subseteq E$ are those edges which connect a vertex in V_1' to a vertex in V_2'. Thus, our partition V_1', V_2' determines E'.

Since the example is in this section, you can tell we're going to use probability somehow. But how? (Our previous methods won't work since we are given a particular graph G.) We can choose the partition of V randomly.

Our probability space will be the uniform probability on the set of all subsets of V. A subset will be V_1' and its complement $V - V_1'$ will be V_2'. For every edge $e = \{x,y\} \in E$, define a random variable X_e by

$$X_e(V_1') = \begin{cases} 0, & \text{if both ends of } e \text{ are in } V_1' \text{ or in } V_2', \\ 1, & \text{if one end of } e \text{ is in } V_1' \text{ and the other in } V_2'. \end{cases}$$

You should be able to see that the number of edges in the bipartite graph is $X(V_1')$, which we define by

$$X(V_1') = \sum_{e \in E} X_e(V_1').$$

We can think of the probability space as a product space as follows. For each $v \in V$, choose it with probability 1/2, independent of the choices made on the other vertices. The chosen vertices form V_1'. The probability of choosing V_1' is

$$(1/2)^{|V_1'|}(1/2)^{|V-V_1'|} = (1/2)^{|V|},$$

where $(1/2)^{|V_1'|}$ is the probability of choosing each of the vertices in V_1' and $(1/2)^{|V-V_1'|}$ is the probability of *not* choosing each of the vertices in $V - V_1'$. Thus, this probability space gives equal probability to every subset of V, just like our original space. Consequently, we can carry out calculations in either our original probability space or the product space we just introduced. Since the product space has a lot of independence built into it, calculation here is often easier. In particular, $\Pr(X_e = 1) = 1/2$ for any edge $\{u, v\}$ since $X_e = 1$ if and only if we make the same decisions for u and v (both included or both excluded) and this has probability $(1/2)^2 + (1/2)^2 = 1/2$. Thus $\mathbf{E}(X_e) = 1 \times (1/2) + 0 \times (1/2) = 1/2$.

Since a random variable must sometimes be at least as large as it's average, there must be a $V_1' \subseteq V$ with $X(V_1') \geq \mathbf{E}(X)$. Thus there is a bipartite subgraph with at least $\mathbf{E}(X)$ edges. Since $\mathbf{E}(X_e) = 1/2$, $\mathbf{E}(X) = k/2$. Since the expected number of edges in a randomly constructed bipartite subgraph is $k/2$, at least one of these subgraphs has at least $k/2$ edges. In other words, there is a bipartite subgraph containing at least half the edges of G. $\square$

Exercises

Remember the following!

- Expectation is linear: $\mathbf{E}(X_1 + \cdots + X_k) = \mathbf{E}(X_1) + \cdots + \mathbf{E}(X_k)$.
- $\Pr(A_1 \cap \cdots \cap A_m) \leq \Pr(A_1) + \cdots + \Pr(A_m)$, especially when $\Pr(A_1) = \cdots = \Pr(A_m)$.

6.5.1. Compute the following for a random graph in $\mathcal{G}_p(n)$.

(a) The expected number of vertices of degree d.
Hint. Let $X_v = 1$ if v has degree d and $X_v = 0$ otherwise. Study $\sum X_v$.

(b) The expected number of 4-cycles.

(c) The expected number of induced 4-cycles. (An induced subgraph of a graph G is a subset of vertices together with *all* the edges in G that connect the vertices.)

6.5.2. An *embedding* of a simple graph $H = (V_H, E_H)$ into a simple graph $G = (V_G, E_G)$ is an injection $\varphi : V_H \to V_G$ such that $\varphi(E_H) \subseteq E_G$, where we define $\varphi\{u, v\} = \{\varphi(u), \varphi(v)\}$. If $\varphi(E_H) = E_G \cap \mathcal{P}_2(V_H)$, we call the embedding *induced*.

(a) Prove that the expected number of embeddings of H in a random graph in $\mathcal{G}_p(n)$ when $n \geq |V_H|$ is

$$n(n-1)\cdots(n-|V_H|+1)p^{|E_H|} = \frac{n!\, p^{|E_H|}}{(n-|V_H|)!}.$$

(b) Repeat (a) for induced embeddings.

(c) In Example 6.14, we showed that the expected number of triangles in a random graph is $\binom{n}{3}p^3$. If part (a) of the present exercise is applied when H is a triangle, we obtain $6\binom{n}{3}p^3$ for the expected number of embeddings. Explain the difference.

6.5.3. In this exercise we'll find a bound on k (as a function of n) so that most graphs in $\mathcal{G}(n,k)$ do not have cycles. Since we know most graphs in $\mathcal{G}(n,n-1)$ have cycles, we'll assume $k \le n-1$. For $C \subseteq \underline{n}$, let $\mathcal{G}_C$ be those graphs in which C is a cycle. As usual, $N = \binom{n}{2}$.

(a) Show that

$$\Pr(G \text{ has a cycle}) \;\le\; \sum \Pr(\mathcal{G}_C),$$

where the sum is over all subsets of $\underline{n}$ that contain at least three elements.

(b) By arranging the vertices in C in a cycle and then inserting the remaining edges, prove that

$$\Pr(\mathcal{G}_C) \;=\; \frac{(c-1)!/2\,\binom{N-c}{k-c}}{\binom{N}{k}} \qquad \text{where } c = |C|.$$

Remember that, unlike cycles in permutations, cycles in graphs do not have a direction.

(c) Conclude that

$$\Pr(G \text{ has a cycle}) \;\le\; \sum_{c=3}^{k} \binom{n}{c} \frac{(c-1)!/2\,\binom{N-c}{k-c}}{\binom{N}{k}}.$$

(d) Show that the term c in the previous sum equals

$$\frac{k!}{2c\,(k-c)!} \prod_{i=0}^{c-1} \frac{n-i}{N-i} \;<\; \frac{k^c}{2c}\left(\frac{n}{N}\right)^c.$$

6.5.4. We'll redo the previous exercise using $\mathcal{G}_p(n)$.

(a) Let $v_1, \ldots, v_k$ be a list of vertices. Compute the probability that $v_1, \ldots, v_k, v_1$ is a cycle

(b) Show that the probability that a random graph in $\mathcal{G}_p(n)$ contains a k-cycle is less than $n^k p^k$.

(c) Show that the probability that a random graph in $\mathcal{G}_p(n)$ has a cycle is less than $(pn)^3$ when $pn < 2/3$.

6.5.5. This exercise relates to Example 6.16.

(a) A simple graph $G = (V,E)$ is *complete* if it contains all possible edges; that is, $E = \mathcal{P}_2(V)$. Prove that, if $|V| = 2n$, we can construct a bipartite subgraph with n^2 edges. Obtain a similar result if $|V| = 2n+1$.

(b) How close is this result to the lower bound in the example?

(c) Prove that when $|V|$ and k are large, there is a graph $G = (V,E)$ such that $|E| = k$ and the relative error between the lower bound and the best bipartite subgraph of G is $O(1/k^{1/2})$. (Of course we need $k \le |\mathcal{P}_2(V)|$ or there will be no simple graph.)
Hint. Use a complete graph in your construction.

(d) Prove that, if $G(V,E)$ can be properly colored using three colors, then it has a bipartite subgraph with at least $2|E|/3$ edges.

6.5.6. We want to find a number k (depending on n) such that most n-vertex graphs have a k-clique. Divide the vertices in $\lfloor n/k \rfloor$ sets of size k and one (possibly empty) smaller set.

(a) Show that the probability that none of these $\lfloor n/k \rfloor$ sets is a k-clique is $(1 - 2^{-\binom{k}{2}})^{\lfloor n/k \rfloor}$.

(b) It can be shown that $1 - x < e^{-x}$ for $x > 0$. Using this and the previous part, conclude that, for some constant A, almost all n-vertex graphs have a k-clique when $k \le A(\log n)^{1/2}$.

6.6 Finite State Machines

A “finite state machine” is simply a device that can be in any one of a finite number of situations and is able to move from one situation to another. The classic example (and motivation for the subject) is the digital computer. If no peripherals are attached, then the state at any instant is what is stored in the machine. You may object that this fails to take into account what instruction the machine is executing. Not so; that information is stored temporarily in parts of the machine’s central processing unit. We can expand our view by allowing input and output to obtain a finite state machine with I/O.

By formalizing the concept of a finite state machine, computer scientists hope to capture the essential features of some aspects of computing. In this section we’ll study a very restricted formalization. These restricted devices are called “finite automata” or “finite state machines.” The input to such machines is fed in one symbol at a time and cannot be reread by the machine.

Turing Machines

A *Turing machine*, introduced by A.M. Turing in 1937, is a more flexible concept than a finite automaton. It is equipped with an arbitrarily long tape which it can reposition, read and write. To run the machine, we write the input on a blank tape, position the tape in the machine and turn the machine on. We can think of a Turing machine as computing a function: the input is an element of the function’s domain and the output is an element of the function’s range, namely the value of the function at that input. The input and/or the output could be nothing. In fact, the domain of the function is any finite string of symbols, where each symbol must be from some finite alphabet; eg. $\{0,1\}$. Of course, the input might be something the machine wasn’t designed to handle, but it will still do something.

How complicated a Turing machine might we need to build? Turing proved that there exists a “universal” Turing machine $\mathcal{U}$ by showing how to construct it. If $\mathcal{U}$’s input tape contains

1. $\mathrm{D}(\mathcal{T})$, a description of any Turing machine $\mathcal{T}$ and
2. the input I for the Turing machine $\mathcal{T}$,

then $\mathcal{U}$ will produce the same output that would have been obtained by giving $\mathcal{T}$ the input I. This says that regardless of how complicated an algorithm we want to program, there is no need to build more than one Turing machine, namely the universal one $\mathcal{U}$. Of course, it might use a lot of time and a lot of tape to carry out the algorithm, so it might not be practical. Suprisingly, it can be shown that $\mathcal{U}$ will, in some sense, be almost as fast as the the Turing machine that it is mimicking. This makes it possible to introduce a machine independent measure of the complexity of a function.

Although Turing machines seem simple, it is believed that anything that can be computed by any possible computer can be computed by a Turing machine. (This is called *Church’s Thesis.*) Such computable functions are called *recursive functions.* Are there any functions which are not recursive?

Examples of nonrecursive functions are not immediately obvious. Here’s one. “Given a Turing machine $\mathcal{T}$ and input I, will the Turing machine eventually stop?” As phrased this isn’t quite a fair function since it’s not input for a Turing machine. We can change it slightly: “Given $\mathrm{D}(\mathcal{T})$ (a machine readable description of $\mathcal{T}$) and the input I, will the universal Turing machine $\mathcal{U}$ eventually stop?” For obvious reasons, this is called the *halting problem.* You may wonder why this can be thought of as a function. The domain of the function is all possible pairs D, I where D is any machine readable description of a machine and I is any possible input for a machine. The range is $\{$“yes”, “no”$\}$. The machine $\mathcal{U}$ computes the value of the function.

Theorem 6.10 *The halting problem is nonrecursive.*

Proof: We can't give a rigorous proof here; in fact, we haven't even defined our terms precisely. Nevertheless, we can give the idea for a proof. Suppose there existed a Turing machine $\mathcal{H}$ that could solve the halting problem. Create a Turing machine $\mathcal{B}$ that contains within itself what is essentially a subroutine equivalent to $\mathcal{H}$ and acts as follows. Whatever is written on the input tape, it makes a copy of it. It then "calls" $\mathcal{H}$ to process as input the original input together with the copy. If $\mathcal{H}$ says that the answer is "Doesn't stop," then $\mathcal{B}$ stops; otherwise $\mathcal{B}$ enters an infinite loop.

How does this rather strange machine behave? Suppose $\mathcal{B}$ is given $\mathrm{D}(\mathcal{T})$ as input. It then passes to $\mathcal{H}$ the input $\mathrm{D}(\mathcal{T})$ $\mathrm{D}(\mathcal{T})$ and so $\mathcal{H}$ solves the halting problem for $\mathcal{T}$ with $\mathrm{D}(\mathcal{T})$ as input. In other words:

$$\text{If } \mathcal{B} \text{ is given } \mathrm{D}(\mathcal{T}), \text{ it halts if and only if } \mathcal{T} \text{ with input } \mathrm{D}(\mathcal{T}) \text{ would run for ever.} \qquad 6.17$$

What does $\mathcal{B}$ do with the input $\mathrm{D}(\mathcal{B})$? We simply use (6.17) with $\mathcal{T}$ equal to $\mathcal{B}$. Thus, $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ halts if and only if $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ runs for ever. Since this is self-contradictory, there is either an error in the proof, an inconsistency in mathematics, or a mistake in assuming the existence of $\mathcal{H}$. We believe that the last is the case: There cannot be a Turing machine $\mathcal{H}$ to solve the halting problem. $\square$

We can describe the previous proof heuristically. If $\mathcal{H}$ exists, then it predicts the behavior of any Turing machine. $\mathcal{B}$ with input $\mathrm{D}(\mathcal{B})$ is designed to ask $\mathcal{H}$ how it will behave and then do the opposite of the prediction.

Finite State Machines and Digraphs

Consider a finite state machine that receives input one symbol at a time and enters a new state based on that symbol. We can represent the states of the machine by vertices in a digraph and the effect of the input i in state s by a directed edge that connects s to the new state and contains i and the associated output in its name. The following example should clarify this.

Example 6.17 Binary addition We would like to add together two nonnegative binary numbers and output the sum. The input is given as pairs of digits, one from each number, starting at the right ends (units digits) of the input. The pair 22 marks the end of the input. Thus to add 010 and 110 you would input the four pairs 00, 11, 01 and 22 in that order. In other words,

$$\text{the sum problem} \quad \begin{array}{rr} & A_nA_{n-1}\cdots A_1 \\ + & B_nB_{n-1}\cdots B_1 \\ \hline & C_{n+1}C_nC_{n-1}\cdots C_1 \end{array} \quad \text{becomes} \quad A_1B_1, \ldots, A_{n-1}B_{n-1}, A_nB_n, 22.$$

The output is given as single digits with 2 marking the end of the output, so the output for our example would be 00012. (The sum is backwards, $C_1 \ldots C_{n-1}, C_n, C_{n+1}, 2$, because the first output is the units digit.) We have two internal states: carry (C) and no carry (N) You should verify that the adder can be described by the table in Figure 6.11. The entry (o, s_2) in position (s_1, i) says that if the machine is in state s_1 and receives input i, then it will produce output o and move to state s_2. It is called the *state transition table* for the machine. Note that being in state C (carry) and receiving 22 as input causes two digits to be output, the carry digit and the termination digit 2.

We can associate a digraph (V, E, φ) with the tabular description, where $V = \{\text{N, C}\}$, each edge is a 4-tuple $e = (s_1, i, o, s_2)$, $\varphi(e) = (s_1, s_2)$ and i and o are the associated input and output, respectively. In drawing the picture, a shorthand is used: the label $00, 22 : 1, 12$ on the edge from C to N in Figure 6.11 stands for the two edges (C, 00, 1, N) and (C, 22, 12, N).

This example is slightly deficient. We tacitly assumed that everyone (and the machine!) somehow knew that the machine should start in state N. We should really indicate this by labeling N as the *starting state*.

	00	01	10	11	22
N	0, N	1, N	1, N	0, C	2, N
C	1, N	0, C	0, C	1, C	12, N

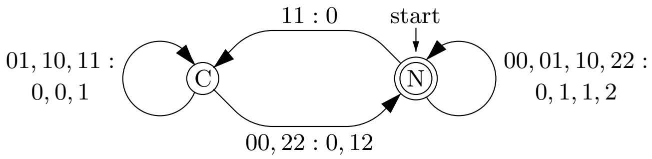


Figure 6.11 Tabular and graphical descriptions of a finite state machine for adding two binary numbers. The starting and accepting states are both N.

You can use the associated digraph to see easily what an automaton does with any given input string. Place your finger on the starting state and begin reading the input. Each time you read an input symbol, follow the directed edge that has that input symbol to whichever state (vertex) it leads and write down the output that appears on the edge. Keep this process up until you have used up all the input. □

Suppose we want a machine that simply recognizes a situation but takes no action. We could phrase this by saying that the machine either accepts (recognizes) or rejects an input string. In this case, we need not have any output; rather, we can label certain states of the machine as "accepting." This is represented pictorially by a double circle around an *accepting state*. If the machine ends up in an accepting state, then the input is accepted; otherwise, it is rejected.

Example 6.18 No adjacent ones Let's construct a machine to recognize (accept) all strings of zeroes and ones that contain no adjacent ones. The idea is simple: keep track of what you saw last and if you find a one followed by a one, reject the string. We use three states, labeled 0, 1 and R, where 0 and 1 indicate the digit just seen and R is the reject state. Both 0 and 1 are accepting states. What is the start state? We could add an extra state for this, but we get a smaller machine if we let 0 be the start state. This can be done because the string $s_1 \cdots s_n$ is acceptable if and only if $0s_1 \cdots s_n$ is. You should be able to draw a diagram of this machine. □

Example 6.19 Divisibility by five Let's construct a machine to recognize (accept) numbers which are divisible by 5. These numbers will be presented from left to right as binary numbers; i.e., starting with the highest order digits. To construct such a machine, we simply design it to carry out the usual long division algorithm that you learned many years ago. At each step in usual long division algorithm you produce a remainder to which you then append the next digit of the dividend. In effect, this multiplies the remainder by ten and adds the next digit to it. Since we are working in binary, we follow the same process but multiply by two instead of by ten. The digraph for the machine appears in Figure 6.12. No output is shown because there is none. □

Here's a formal definition of the concept of a finite automaton, which we've been using rather loosely so far.

> Definition 6.9 Finite automaton *A* **finite automaton** *is a quadruple* (S, I, f, s_0) *where* S *and* I *are finite sets,* $s_0 \in S$ *and* $f: S \times I \to S$. S *is called the set of* **states** *of the automaton,* I *the set of* **input symbols** *and* s_0 *the* **starting state**. *If the automaton has* **accepting states**, *we append them to the quadruple to give a quintuple. If the automaton has a set* **output symbols** O, *then we append it to the tuple and change the definition of* f *to* $f: S \times I \to O \times S$. *If an input string leaves the automaton in an accepting state, we say that the automaton* **accepts** *the string or that it* **recognizes** *the string.*

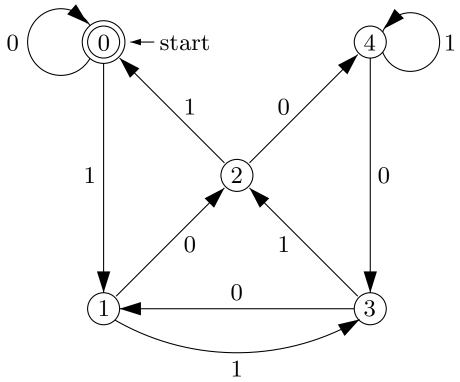


Figure 6.12 A machine to test divisibility of binary numbers by 5. The starting and accepting states are both 0. Input begins with the high order (leftmost) bit.

Example 6.20 An automaton grammar We can represent a finite automaton without output in another fashion. For each edge (s_1, i, s_2) we write $s_1 \to i, s_2$. If s_2 is an accepting state, we also write $s_1 \to i$. Suppose we begin with the starting state, say s_0 and replace it with the right side of some $s_0 \to$. If this leads to a string contains a state s, then replace s in the string with the right side of some $s \to$. After n such steps we will end up with either a string of n input symbols or a string of n input symbols followed by a state.

We claim that any string of input symbols (with no appended state) that can be produced in this fashion is accepted and conversely. Why is this true? Our replacement process mimics travelling along the digraph as dictated by the input string. We can only omit the state symbol at the end by moving to an accepting state. This is an example of a "grammar." We'll say more about grammars in Section 9.2. $\square$

We can attempt to endow our string recognizer with "free will" by allowing random choices. At present, for each state $s \in S$ and each $i \in I$ there is precisely one $t \in S$ such that (s, i, t) is an edge of the digraph. Remove the phrase "precisely one." We can express this in terms of the function f by saying that $f\colon S \times I \to 2^S$, the subsets of S, instead of $f\colon S \times I \to S$. Here $f(s, i)$ is the set of all states t such that (s, i, t) is an edge.

Since the successor of a state is no longer uniquely defined, what happens when the machine is in state s and receives input i? If $f(s, i) = \emptyset$, the empty set, the machine stops and does not accept the string; otherwise, the machine selects by its "free will" any $t \in f(s, i)$ as its next state.

Since the outcome of a given input string is not uniquely determined, how do we define acceptance? We simply require that acceptance be possible: If it is possible for the machine to get from its starting state to an accepting state by any sequence of choices when it receives the input string X, then we say the machine accepts X. Such a machine is called a *nondeterministic finite automaton.* What we have called finite automata are often called *deterministic* finite automata.

Theorem 6.11 No "free will" *Given a nondeterministic finite automaton, there exists a (deterministic) finite automaton that accepts exactly the same input strings.*

The conclusions of this theorem are not as sweeping as our name for it suggests: We are speaking about a rather restricted class of devices and are only concerned about acceptance. The rest of this section will be devoted to the proof.

Proof: Let $\mathcal{N} = (S, I, f, s_0, A)$, with $f\colon S \times I \to 2^S$, be a nondeterministic finite automaton. We will construct a deterministic finite automaton $\mathcal{D} = (T, I, g, t_0, B)$ that accepts the same input strings as $\mathcal{N}$.

Let the states T of $\mathcal{D}$ be 2^S, the set of all subsets of S. The initial state of $\mathcal{D}$ will be $t_0 = \{s_0\}$ and the set of accepting states B of $\mathcal{D}$ will be the set of all those subsets of S that contain at least one element from A. We now define $g(t, i)$. Let $g(\emptyset, i) = \emptyset$. For t a nonempty subset of S, let $g(t, i)$ be the union of $f(s, i)$ over all $s \in t$; that is,

$$g(t, i) = \bigcup_{s \in t} f(s, i). \tag{6.18}$$

This completes the definition of $\mathcal{D}$.

We must prove that $\mathcal{D}$ recognizes precisely the same strings that $\mathcal{N}$ does. Let t_n be the state that $\mathcal{D}$ is in after receiving the input string $X = i_1 \ldots i_n$. We claim that

$$\mathcal{N} \text{ can be in a state } s \text{ after receiving } X \text{ if and only if } s \in t_n. \tag{6.19}$$

Before proving (6.19), we'll use it to prove the theorem.

Suppose that $\mathcal{N}$ accepts X. Then it is possible for $\mathcal{N}$ to reach some accepting state $a \in A$, which is in t_n by (6.19). By the definition of B, $t_n \in B$. Thus $\mathcal{D}$ accepts X.

Now suppose that $\mathcal{D}$ accepts X. Since t_n is an accepting state of $\mathcal{D}$, it follows from the definition of B that some $a \in A$ is in t_n. By (6.19), $\mathcal{N}$ can reach a when it receives X. We have shown that (6.19) implies the theorem.

It remains to prove (6.19). We'll use induction on n. Suppose that $n = 1$. Since $t_0 = \{s_0\}$, it follows from (6.18) that $t_1 = f(t_0, i_1)$. Since this is the set of states that can be reached by $\mathcal{N}$ with input i_1, we are done for $n = 1$.

Suppose that $n > 1$. By (6.18),

$$t_n = \bigcup_{s \in t_{n-1}} f(s, i_n).$$

By the induction assumption, t_{n-1} is the set of states s that $\mathcal{N}$ can be in after receiving the input $i_1 \ldots i_{n-1}$. By the definition of f, $f(s, i_n)$ is the set of states that $\mathcal{N}$ can reach from s with input i_n. Thus t_n is the set of states that $\mathcal{N}$ can be in after receiving the input $i_1 \ldots i_n$. $\square$

Exercises

6.6.1. What is the state transition table for the automaton of Example 6.19?

6.6.2. We now wish to check for divisibility by 3.

(a) Give a digraph like that in Example 6.19 for binary numbers.

(b) Give the state transition table for the previous digraph.

(c) Repeat the two previous parts when the input is in base 10 instead of base 2.

*(d) Construct an automaton that accepts decimal input, starting with the unit's digit and checks for divisibility by 3.

*(e) Construct an automaton that accepts binary input, starting with the unit's digit and checks for divisibility by 3.

6.6.3. Design a finite automaton to recognize all strings of zeroes and ones in which no maximal string of ones has even length. (A maximal string of ones is a string of adjacent ones which cannot be extended by including an adjacent element of the string.)

6.6.4. An automaton is given by $(\{b, s, d, z\},\ \{+, -, 0, 1, \ldots, 9\}, f\ , b,\ \{d\})$, where $f(b, +) = s$, $f(b, -) = s$, $f(t, k) = d$ for $t = b, s, d$ and $0 \le k \le 9$, and $f(t, i) = z$ for all other $(t, i) \in S \times I$.

(a) Draw the digraph for the machine.

(b) Describe the strings recognized by the machine.

(c) Describe the strings recognized by the machine if both s and d are acceptance states.

6.6.5. A "floating point number" consists of two parts. The first part consists of an optional sign followed by a nonempty string of digits in which at most one decimal point may be present. The second part is either absent or consists of "E" followed by a signed integer. Draw the digraph of a finite automaton to recognize floating point numbers.

6.6.6. A symbol i_k a string $i_i \ldots i_n$ is "isolated" if (i) either $k = 1$ or $i_k \ne i_{k-1}$, and (ii) either $k = n$ or $i_k \ne i_{k+1}$. For example, 0111010011 contains two isolated zeroes and one isolated one.

(a) Draw a digraph for a finite automaton that accepts just those strings of zeroes and ones that contain at least one isolated one.

(b) Now draw a machine that accepts strings with precisely one isolated one.

6.6.7. In this exercise you are to construct an automaton that behaves like a vending machine. To keep things simple, there are only three items costing 15, 20 and 25 cents and indicated by the inputs A, B and C, respectively. Other allowed inputs are 5, 10 and 25 cents and R (return coins). The machine will accept any amount of money up to 30 cents. Additional coins will be rejected. When input A, B or C is received and sufficient money is present, the selection is delivered and change (if any) returned. If insufficient money is present, no action is taken.

(a) Describe appropriate states and output symbols. Identify the starting state.

(b) Give the state transition table for your automaton.

(c) Draw a digraph for your automaton.

6.6.8. Suppose that $\mathcal{M} = (S, I, f, s_0, A)$ and $\mathcal{M}' = (S', I, f', s_0', A')$ are two automata with the same input symbols and with acceptance states A and A' respectively.

(a) Describe in terms of sets and functions an automaton that accepts only those strings acceptable to both $\mathcal{M}$ and $\mathcal{M}'$.
Hint. The states can be $S \times S'$.

(b) When is there an edge from (s, s') to (t, t') in your new automaton and what input is it associated with?

(c) Use this idea to describe an automaton the recognizes binary numbers which are divisible by 15 in terms of those in Example 6.19 and Exercise 6.6.2.

(d) Design a finite automaton that recognizes those binary numbers which are either divisible by 3 or divisible by 5 or both.

Notes and References

Spanning trees are one of the most important concepts in graph theory. As a result, they are discussed in practically every text. Our point of view is like that taken by Tarjan [7; Ch.6], who treats the subject in greater depth.

An extensive treatment of planarity algorithms can be found in Chapters 6 and 7 of the text by Williamson [8]. Wilson's book [98] is a readable account of the history of the four color problem.

Since flows in networks is an extremely important subject, it can be found in many texts. Papadimitriou and Steiglitz [6] and Tarjan [7] treat the subject extensively. In addition, network flows are related to linear programming, so you will often find network flows discussed in linear programming texts such as the book by Hu [4].

The marriage theorem is a combinatorial result about sets. Sperner's theorem, which we studied in Example 1.22, is another such result. For more on this subject, see the text by Anderson [1]. Related in name but not in results is the *Stable Marriage Problem.* In this case, we have two sets of the same size, say men and women. Each woman ranks all of the men and each man all of the women. The men and women are married to each other. The situation is considered stable if we cannot find a man and a woman who both rank each other higher than their mates. It can be proved that a stable marriage always exists. Gusfield and Irving [3] discuss this problem and its applications and generalizations.

The books on the probabilistic method are more advanced than the discussion here. Perhaps the gentlest book is the one by Molloy and Reed [5].

Automata are discussed in some combinatorics texts that are oriented toward computer science. There are also textbooks, such as Drobot [2] devoted to the subject.

1. Ian Anderson, *Combinatorics of Finite Sets*, Dover (2002).
2. Vladimir Drobot, *Formal Languages and Automata Theory*, W. H. Freeman (1989).
3. Dan Gusfield and Robert W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press (1989).
4. T.C. Hu, *Integer Programming and Network Flows*, Addison-Wesley (1969).
5. Michael Molloy and Bruce Reed, *Graph Coloring and the Probabilistic Method*, Springer-Verlag (2002).
6. Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover (1998).
7. Robert E. Tarjan, *Data Structures and Algorithms*, SIAM (1983).
8. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).
9. Robin Wilson, *Four Colours Suffice: How the Map Problem was Solved*, Penguin Press (2002).