8

MORE GRAPH THEORY

8:1 MINIMUM-DISTANCE TREES

This chapter discusses five real-world problems that can be solved using graphs. Reflecting the current state of knowledge concerning graph algorithms, some of our problems have good solutions, while others have no known good algorithmic solution. In the latter case we present exponential algorithms. In Section 4 we present an **approximation algorithm**, that is, an algorithm that runs in polynomial time but doesn't necessarily give a best possible answer.

Our first application is plowing snow off the streets of a city. We envision two problems. The first consists of clearing roads connecting important city services along shortest routes. The second consists of finding a route that traverses every remaining street at least once but that is overall as short as possible. In a variation we seek a shortest route that visits a designated set of points in a city. As a fourth problem we design a program to position a laser bit to drill thousands of holes in a sheet of material. Finally, we consider storage allocation in computer memory.

To begin the first snowplowing problem, imagine that we are in charge of plowing the snow off the streets of a city all of whose essential services (e.g., police, fire protection, ambulance, and snowplow) are located in one building called City Hall. Within the city there are special facilities (e.g., hospitals and schools) that we would like to be able to reach with the essential services. How should we plow the streets to enable our vehicles to reach the special facilities as quickly as possible?

At first glance it seems reasonable to think of trees. In a tree every pair of distinct vertices is joined by a unique path. Here too it seems that we don't need more than one clear route joining different locations; however, a minimum-weight spanning tree (defined in Section 5.3) is not what we want. (See also Exercise 12.)

In particular, the snowplowing problem has a distinguished location, City Hall. Hence our graph model needs a distinguished vertex called the **root**. The remaining vertices of the graph correspond with principal intersections within the city. For simplicity assume that each of the special facilities is at one of these intersections. Two vertices of the graph will be joined by an edge if there is a direct road connection between the corresponding intersections. The weight attached to an edge will represent the length of that connection. Thus the resulting weighted graph models the streets of the city, and as a plan for emergency snowplowing we want a spanning tree of this graph with the property that the distance from the root to each of the special facility vertices along edges of the spanning tree is minimized. In fact, we find a spanning tree that contains a minimum-distance path from the root to every vertex. Such a subgraph is called a **minimum-distance spanning tree**.

Here are precise formulations of ideas from the preceding paragraph. Recall from Section 5.3 that the weight of a path P, denoted by $w(P)$, is the sum of the weights of all edges in P; we call this the **length** of P. Recall also that each edge of a weighted graph has positive weight.

Definition. In a weighted connected graph G, the **distance between two vertices x and y**, denoted by $\boldsymbol{d(x, y)}$, is the minimum value of $w(P)$, taken over all paths P from x to y. (Informally, the distance from x to y is the length of the shortest path between them.) A **minimum-distance spanning tree** in a weighted connected graph G with root r is a tree T such that for each vertex v of G, the length of the unique path in T from r to v equals $d(r, v)$.

Example 1.1. In Figure 8.1 we show weighted graphs G and H with root r, their minimum-distance spanning tree, and their minimum-weight spanning tree. Note that the two types of trees may differ.

Problem. Given a weighted connected graph G and a root vertex r, find a minimum-distance spanning tree of G.

Question 1.1. For each vertex $v \neq r$ in the graph shown in Figure 8.2, find the shortest path from v to r. Does the union of these paths form a minimum-distance spanning tree? Pick a different vertex for the root and find all shortest paths to this new vertex. Does the union of these paths form the same tree?

There is a good algorithm to solve the minimum-distance problem, due to E. W. Dijkstra. It is not obvious that every connected weighted graph contains a minimum-distance spanning tree; it is conceivable that the union of shortest paths from r to different vertices contains a cycle. However, one consequence of Dijkstra's algorithm and the proof that it works is that minimum-distance spanning trees always exist. The fundamental idea is simple (and in a sense greedy). We shall

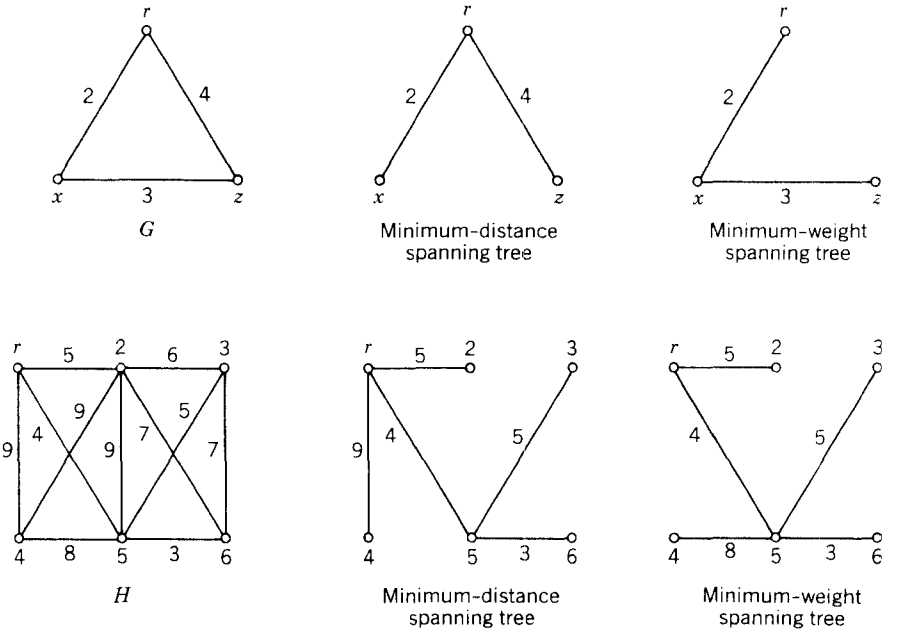


Figure 8.1

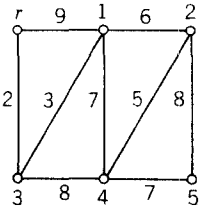


Figure 8.2

describe the algorithm informally and leave details of implementation to the exercises.

Begin with the root r. If we examine the edges incident with r and select one of smallest weight, say $e = (r, x)$, then the shortest path from r to x is the edge e, since we assume that all edge weights are positive. So far the minimum-distance tree consists of e and the vertices r and x. Next we want to extend this tree so that it remains a minimum-distance spanning tree for a subgraph of G. There are two kinds of edges that we might pick to add to the tree. We might select an edge of the form $f = (r, y)$ or $g = (x, z)$. Among the edges incident with r assume that f has minimum weight (other than e). Among the edges incident with x assume that g has minimum weight. A naive choice would be to select whichever of f and g has minimum weight. Unfortunately, this will not work in all cases.

Example 1.2. Suppose that G is as in Figure 8.1. The first edge we choose is (r, x), since it has minimum weight. If we then select (x, z) because its weight is less than the weight of (r, z), then the distance (within the tree) from r to z would be 5. In G the distance from r to z is 4. The minimum-distance spanning tree should consist of the edges (r, x) and (r, z).

What we should do, instead of adding an edge of smallest weight, is to pick a new edge that creates a minimum-distance path to a new vertex. That is, we want to find an edge $e = (y, z)$ such that y is in the minimum-distance spanning tree created so far, z is as close to the root as any vertex not in the tree, and a shortest path from z to the root r uses e and edges already in the tree. This idea is incorporated into step 4 of the following algorithm.

Algorithm DIJKSTRA

STEP 1. Input the weighted graph G and the root vertex r {Assume that G is connected.}
STEP 2. Set $T := \{r\}$
STEP 3. For $j = 1$ to $V - 1$ do
Begin
STEP 4. Find z, a vertex in $G - T$ whose distance from r is minimum; let e be the edge from z to T in some minimum-distance path from z to r
STEP 5. Set $T := T + z + e$
End
STEP 6. Output T and stop.

Step 4 in DIJKSTRA might raise a question. Suppose that z is a closest vertex of $G - T$ to r. How do we know that there is an edge e joining z to a vertex of T? Maybe the shortest path from z to r uses different vertices than those (so far) in T? That this problem will not arise is a consequence of the proof of Theorem 1.1.

Example 1.3. We trace Dijkstra on the graph shown in Figure 8.3. See Table 8.1.

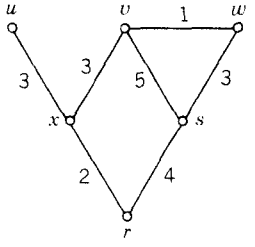


Figure 8.3

Table 8.1

Step No.	j	z	$V(T)$	$E(T)$
2	?	?	$\{r\}$	$\varnothing$
4	1	x		
5	1	x	$\{r, x\}$	$\{(r, x)\}$
4	2	s		
5	2	s	$\{r, x, s\}$	$\{(r, x), (r, s)\}$
4	3	u		
5	3	u	$\{r, x, s, u\}$	$\{(r, x), (r, s), (x, u)\}$
4	4	v		
5	4	v	$\{r, x, s, u, v\}$	$\{(r, x), (r, s), (x, u), (x, v)\}$
4	5	w		
5	5	w	$\{r, x, s, u, v, w\}$	$\{(r, x), (r, s), (x, u), (x, v), (v, w)\}$

Note that when we have a choice, as between u and v in the third application of step 4, we may choose either vertex.

Question 1.2. Given the weighted graph in Figure 8.4 with root r as shown, use DIJKSTRA to find the minimum-distance tree.

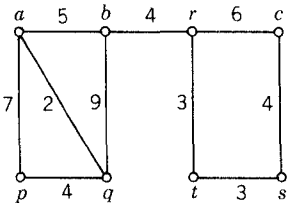


Figure 8.4

Theorem 1.1. DIJKSTRA produces a minimum-distance spanning tree T of a connected weighted graph G with root r.

Proof. We must prove that T is a spanning tree and that for each vertex v of G the distance from v to the root r along the edges of T equals $d(r, v)$, the length of a shortest path in G joining r and v. We prove by induction on $|V(T)|$ that at each stage T is a tree containing a minimum-distance path from each vertex of T to the root r. Thus when DIJKSTRA stops and $|V(T)| = V = |V(G)|$, T is a minimum-distance spanning tree.

Example 1.3 (reexamined). Look at T after the third completion of step 5. T contains four vertices and three edges and is a minimum-distance spanning tree of the subgraph of G that contains the vertices $\{r, x, s, u\}$ and the edges (r, x), (r, s) and (x, u).

Initially, $|V(T)| = 1$ since $V(T) = \{r\}$. In step 5 we add the edge $e = (r, x)$ of least weight and the vertex x to T. Then T is a tree with two vertices and one edge, and this edge provides the shortest path from x to r. Thus the base case is safely accounted for.

Assume that T is a tree containing minimum-distance paths whenever $|V(T)| < k$, and suppose that $|V(T)| = k$. T received a kth vertex, say v, and a $(k-1)$st edge, say e, in step 5. Then $T' = T - v - e$ was stored as T in the previous execution of step 5. Since T' contains $(k-1)$ vertices, by the inductive hypothesis it is a tree that contains minimum-distance paths from each of its vertices to r. At the next occurrence of step 4, the vertex v in $G - T$ was selected as a vertex of minimum distance to r. Since v is not in T, the addition of v and e does not create a cycle and T remains acyclic.

Suppose that $P = \langle v, x, \ldots, r \rangle$ is a minimum-distance path from v to r in G beginning with edge e. Since x is closer to r than v, x is in T'. Otherwise, DIJKSTRA would have selected x before v. Then we add v and $e = (v, x)$ to T', and $T = T' + v + e$ is acyclic and connected, hence a tree. Furthermore, the shortest path from x to r in T' plus e will be a shortest path from v to r in T. (See also Exercise 11.) This proves the inductive step. Thus the tree output by DIJKSTRA is a minimum-distance spanning tree. □

Question 1.3. Where in DIJKSTRA is the connectivity of the graph G essential? Find at least two places in the proof of Theorem 1.1 where we use the fact that edge weights are positive. What are the problems with running DIJKSTRA on the graph shown in Figure 8.5?

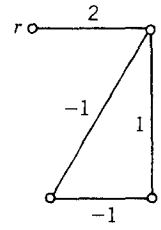


Figure 8.5

We now show that the complexity of DIJKSTRA is $O(V \cdot E)$. We need to perform comparisons and additions to find the minimum-distance paths; however, as in other graph complexity results we count only comparisons. The loop at step 3 occurs $V - 1$ times. In step 4 we need to check at most E edges that join a vertex of T with one in $G - T$ to find the next shortest path and the vertex z. Thus there are no more than $(V-1)E$ comparisons needed in total. In Exercises 19 to 21 a more detailed version of DIJKSTRA is presented, and in that version we can see that only $O(V)$ comparisons are needed within the equivalent of step 4 so that DIJKSTRA has an overall complexity of $O(V^2)$. This was the original complexity

bound obtained by Dijkstra. There has been considerable interest in this minimum-distance spanning tree problem, and variations of this algorithm using more sophisticated data structures have been developed, including one that has complexity $O(E \log(V))$.

With a slight change DIJKSTRA can be applied to unweighted graphs. Recall that distance in an unweighted graph has been defined to be the fewest number of edges in a path joining two vertices. (See Section 5.3.) Thus if we assign a weight of 1 to each edge of an unweighted connected graph and choose a root r, DIJKSTRA will find a minimum-distance spanning tree. In this context the resulting tree is known as a **breadth-first-search** (or **BFS**) **spanning tree**. Notice that when DIJKSTRA is applied to an unweighted graph, first it "visits" and adds in to the tree T all vertices adjacent to the root r. Next it "visits" and adds in all vertices adjacent to vertices adjacent to r, that is, it "visits" all vertices at distance 2 from r, and then successively "visits" all vertices at distance j from r for $j = 3, 4, \ldots$. "Visiting" vertices in a graph in this order is known as breadth-first search.

Example 1.4. Figure 8.6 shows a graph G and two BFS spanning trees of G. Working on the tree G with each edge weight 1, DIJKSTRA first adds edges $(r, 2)$ and $(r, 6)$ to the tree, since vertices 2 and 6 are at distance 1 from the root r. Next the vertices 3 and 5 at distance 2 from r are added to the tree; there is a choice of edges here. Finally, vertex 4 at distance 3 from r is added. Two possible breadth-first-search spanning trees are shown in Figure 8.6.

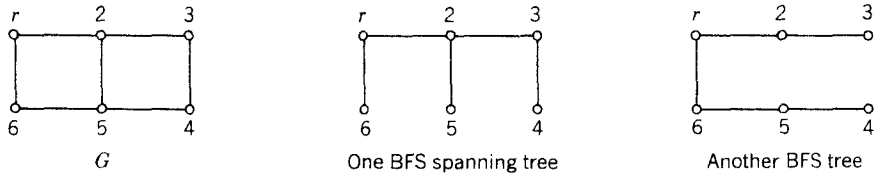


Figure 8.6

Here is a breadth-first-search algorithm, modeled upon DIJKSTRA. The set T contains the vertices and edges of the BFS tree.

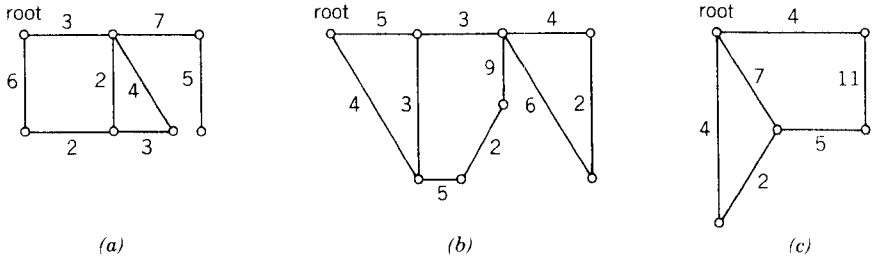
Algorithm BREADTHFIRSTSEARCH (BFS)

STEP 1. Input the unweighted graph G and root r
STEP 2. $T := \{r\}$
STEP 3. For $j = 1$ to $V - 1$ do
  STEP 4. For each vertex v in $G - T$ adjacent to a vertex at distance $(j - 1)$ from r do
    STEP 5. Select w, one neighbor of v in T;
      set $T := T + (v, w) + v$
STEP 6. Output T and stop.

Notice that when applied to a disconnected graph, BFS visits and constructs a spanning tree on precisely the vertices in the same component as the root r. In applications it is common for BFS to perform some calculation when it visits a vertex and to output more than just the spanning tree. Breadth-first search is an important algorithmic technique that will be used again in Section 5.

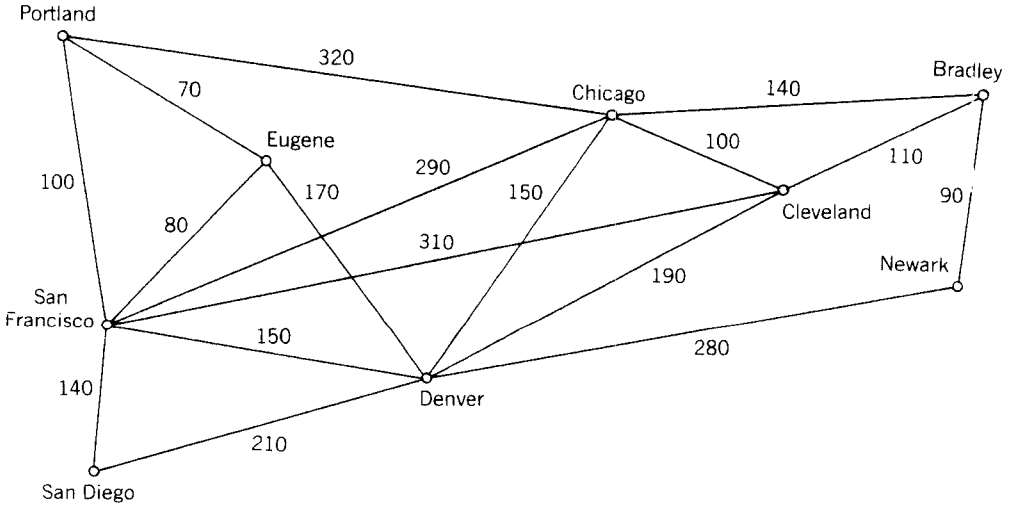
EXERCISES FOR SECTION 1

1. Find a minimum-distance tree for each of the following graphs.



(a) *(b)* *(c)*

2. There do not exist direct flights from Bradley Field to all other airports in the United States. If you wish to fly from Bradley to, for example, Eugene, Oregon, you will have to change planes at least once. The graph indicates some of the possible connecting flights that you might choose. The vertices of the graph are labeled with the names of the corresponding cities. An edge represents a direct flight between the two corresponding cities. The weight on the edge indicates the cost of the flight. Assume that the cost to fly from A



to C changing planes at B equals the cost to fly from A to B plus the cost to fly from B to C. Find a minimum-cost trip from Bradley to Eugene.

3. Rewrite DIJKSTRA so that upon input of a weighted connected graph G and two vertices x and y, it finds a shortest path from x to y. Is your algorithm necessarily more efficient than (the original version of) DIJKSTRA?

4. Find an unweighted connected graph G so that every spanning tree of G is a minimum-distance spanning tree for some choice of the root.

5. Find a weighted graph G so that no matter what vertex is chosen for the root of G, the minimum-distance spanning tree is heavier than the minimum-weight spanning tree (i.e., for every vertex v the sum of the edge weights of a minimum-distance spanning tree with root v is larger than the weight of a minimum-weight spanning tree).

6. Rewrite DIJKSTRA so that it finds a minimum-distance spanning tree if the graph is connected, or else reports that the graph is not connected.

7. Here is a table of costs of some intercity flights; a zero indicates no direct flight. Find the cost of a cheapest trip between every pair of cities.

	C_1	C_2	C_3	C_4	C_5
C_1	—	100	0	150	210
C_2	100	—	0	90	0
C_3	0	0	—	50	280
C_4	150	90	50	—	0
C_5	210	0	280	0	—

8. Run the unweighted version of DIJKSTRA on each of the graphs in Exercise 1 with the edge weights discarded (i.e., run it on the underlying unweighted graphs.)

9. A new commuter airline called Capital Cities offers flights between the capital cities of every pair of states that share a border. So, for example, there is a flight from Pierre, South Dakota, to Bismark, North Dakota, since these two states share a common boundary. Each such flight costs $25. Using Capital Cities, what is the cost of a cheapest trip from Boston to Sacramento? Bismark to Trenton?

10. Suppose that G is a weighted connected graph and that one path $P = \langle x_1, x_2, \ldots, x_j, r\rangle$ to the root is designated as top priority. Rewrite DIJKSTRA so that it finds a spanning tree that includes P. For every vertex v not in P, the algorithm should find as short a path from v to r as possible.

11. Suppose that $P = \langle x, y, \ldots, r\rangle$ is a shortest path from the vertex x to the root r. Explain why the same path, minus x and starting at y, $\langle y, \ldots, r\rangle$, is a

shortest path from y to r. Suppose that P_1 is a shortest path from a vertex x to a vertex z and P_2 is a shortest path from z to r. Is P_1 followed by P_2 a shortest path from x to r?

12. Suppose that a city wants a snowplowing plan to connect City Hall with each of the designated special facilities; however, the plowing budget is greatly overspent. If the sole criterion for choosing plowing routes is that the total plowing cost should be a minimum, then explain why a minimum-weight spanning tree rather than a minimum-distance spanning tree provides the best plan.

13. Prove that at the end of the algorithm BFS precisely those vertices in the same component as the root r are contained in T.

14. Rewrite BFS so that the vertices are assigned a number giving the order in which they become visited. (*Note:* This ordering is not unique but depends on arbitrary choices made within the algorithm.)

15. Verify that BFS performs at most $O(V^2)$ comparisons given a graph with V vertices.

16. Rewrite BFS so that it performs a breadth-first search on each connected component of G.

17. Construct a BFS algorithm that given a graph computes the eccentricity of every vertex. The eccentricity is defined in Exercise 5.4.14.

18. The **radius $r(G)$** and **diameter $d(G)$** of a graph G are respectively the minimum and maximum value of the eccentricity (see the preceding exercise). Construct an algorithm that, given a graph G, outputs the radius and diameter of G. Show that $d(G)/2 \leq r(G) \leq d(G)$ for any graph G. Find graphs to show that there are no better bounds than those given by the inequality above.

19. Here is a detailed version of DIJKSTRA that specifies the equivalent of step 4.

Algorithm DIJKSTRA2

STEP 1. Input the weighted graph G and the root vertex r {Assume that G is connected.}
STEP 2. Set $d(r) = 0$ {$d(x)$ denotes the distance of x from the root in the partial tree.}
STEP 3. Set $V(T) := \{r\}$; $E(T) = \varnothing$ {These will contain, respectively, the vertices and edges of T.}
STEP 4. For $j = 1$ to $V - 1$ do
Begin
STEP 5. For each (t, x) in $E(G)$ with t in $V(T)$ and x in $V(G) - V(T)$ do
STEP 6. Set $c(t, x) := d(t) + w(t, x)$

{At this point $c(t, x)$ indicates the length of the path from r to x using a path in T together with the edge (t, x).}

STEP 7. Set $c_{\min} :=$ minimum value of $c(t, x)$
STEP 8. Set $x_{\min}$ and $t_{\min}$ equal to the vertices that achieve the minimum of step 7
STEP 9. Set $V(T) := V(T) + x_{\min}$
STEP 10. Set $E(T) := E(T) + (t_{\min}, x_{\min})$
STEP 11. Set $d(x_{\min}) := c_{\min}$
End {step 4}
STEP 12. Output $E(T)$ and stop.

Run DIJKSTRA2 on each of the graphs in Exercise 1.

20. Explain why in DIJKSTRA2 when a vertex $x_{\min}$ and the edge $(t_{\min}, x_{\min})$ are added to the tree T, T is a tree containing a minimum-distance path from $x_{\min}$ to r.

21. Count the maximum number of additions and comparisons performed in DIJKSTRA2 and show that each is $O(V^2)$.

22. In the remarks following DIJKSTRA's algorithm we asserted that there is an algorithm for the minimum-distance spanning tree problem that runs in time $O(E \log(V))$. For what graphs is this a better bound than the $O(V^2)$ complexity bound that is obtained in the preceding exercise?

23. Modify DIJKSTRA2 so that the shortest path from each vertex to r is maintained as well as the distance of that path.

24. Using BFS construct an algorithm to check whether a connected graph is bipartite or not. (See Section 5.2 for the definition of bipartite. See also Supplementary Exercise 10 of Chapter 5.)

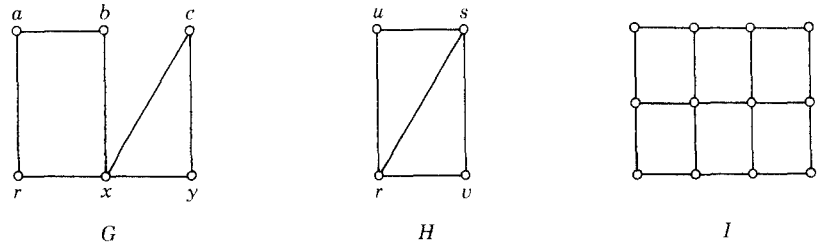
25. Modify DIJKSTRA so that for every pair of distinct vertices v and w in a weighted connected graph, the shortest path between v and w and its length is found. This is known as the **All Pairs Problem**. Show that the complexity of the All Pairs Problem is at most $O(V)$ times the complexity of the minimum-distance spanning tree problem.

8:2 EULERIAN CYCLES

We continue with snowplowing. To repeat the setting, suppose that a weighted graph is drawn to model city streets. Each vertex represents an intersection, and two vertices are joined by an edge if the corresponding intersections are joined by a direct road connection. The weight of an edge is the length of the road. The problem is to plow the streets efficiently (or if, as in the preceding section, certain streets are already clear, to plow the remaining streets efficiently). More precisely,

the problem is to devise a plan to travel along each unplowed street at least once in as short a trip as possible, beginning and ending at City Hall. Of course, it would be most efficient to plow the streets with no repetitions. Is this possible, and if so, how can such a plan be found?

Example 2.1. Consider Figure 8.7. The graph G contains a cycle (for example $\langle r, x, c, y, x, b, a, r \rangle$) that traverses every edge exactly once. Although the graph H contains no such cycle, it does contain the path $\langle r, u, s, v, r, s \rangle$ that traverses every edge exactly once. The graph I is a 3×2 grid graph. In Chapter 3 we saw that it was impossible to traverse each edge of this graph exactly once.



G H I

Figure 8.7

Definition. A path or cycle that includes every edge of a graph exactly once is called **Eulerian**. A graph that contains an Eulerian cycle is called an **Eulerian graph**.

Question 2.1. (a) Find an Eulerian graph with four vertices. (b) Find a graph with eight vertices that is not Eulerian but contains an Eulerian path. (c) Find a connected graph with six vertices that does not contain an Eulerian path.

The problem of characterizing the graphs that contain Eulerian paths led to the first graph theory paper, written in 1736 by Leonhard Euler. Euler was visiting Königsberg, a town with seven bridges and demonstrated that it was impossible to take a walk crossing every bridge exactly once.

Question 2.2. Which of the graphs in Figure 8.8 are Eulerian? Which contain Eulerian paths but not Eulerian cycles?

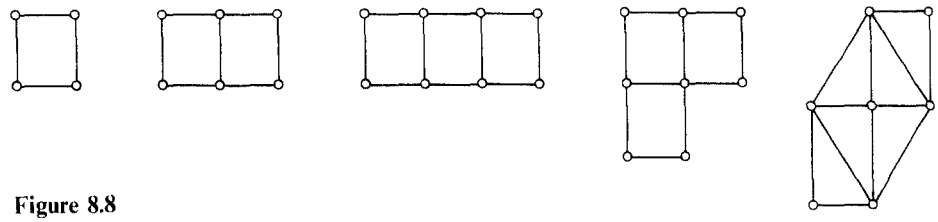


Figure 8.8

Theorem 2.1 (Euler's Theorem). A connected graph is Eulerian if and only if every vertex has even degree.

Euler's theorem tells the snowplow planners that they can plow each street exactly once if and only if an even number of streets comprise every intersection. Note that one can think of an Eulerian cycle as starting and ending at any vertex, in particular City Hall. Our proof of Euler's theorem will lead to an efficient algorithm for finding Eulerian paths and cycles. Then we shall consider ways to modify this algorithm to be useful in diverse settings.

Proof of Euler's theorem. First notice that an Eulerian graph must be connected. Let C be an Eulerian cycle in a graph G. Pick an arbitrary vertex of G, say x. We can assume that C begins at x, leaving on an edge, say e_1, and at some point returns to x on, say e_2. If that is the end of C, then $\deg(x) = 2$, an even number. Otherwise, C leaves x again on, say e_3, and later returns on, say e_4. Each time that C leaves x on an edge e_i, it returns on a different edge e_{i+1}. Since the edges at x can be paired, e_1 with e_2, e_3 with e_4, and so on, there must be an even number of them. Thus $\deg(x)$ is even. Since x was chosen arbitrarily, every vertex of G must have even degree.

Next we prove the converse, that a connected graph with all vertices of even degree contains an Eulerian cycle. We prove this by induction on V. If $V = 1$, then the graph contains no edges and vacuously satisfies the conclusion. A connected graph with $V = 2$ consists of one edge and so does not have vertices of even degree. If $V = 3$, then the only connected graph with all vertices of even degree is the 3-clique, K_3, and this graph contains an Eulerian cycle.

Question 2.3. Find every connected graph with four or five vertices all of whose degrees are even. Show that each such graph is Eulerian.

Assume that every connected graph with fewer than k vertices all of whose degrees are even contains an Eulerian cycle. Let G be a connected graph with k vertices all of even degree. Pick a vertex, say x, and create a path P beginning at x. Extend P, appending incident unused edges at its end, until this is no longer possible. We claim that P is a cycle, ending at x. Since every vertex of G has even degree, when P arrives at a vertex $v \neq x$ there is always an unused edge on which to leave v. Thus P must end at x; hence we rename P as C, since it is a cycle. If C traverses every edge of G, then it is the sought-after Eulerian cycle.

Example 2.2. Consider the graph shown in Figure 8.9. The cycle $C' = \langle r, 2, 5, 6, r \rangle$ can be extended further at r. The cycle $C = \langle r, 2, 5, 6, r, 7, 8, r \rangle$ cannot be extended further at r; however, C is not an Eulerian cycle.

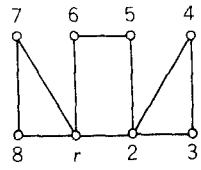


Figure 8.9

If C is not an Eulerian cycle, then we construct G' from G by erasing all edges of C. Since C is a cycle, we erase an even number of edges at each vertex. Since all vertices had even degree originally, their degree remains even in G'. In G' the vertex x has degree 0. Thus each connected component of G' contains fewer than k vertices and is consequently an Eulerian graph.

Question 2.4. Let C be as in Example 2.2. Identify the graph G' obtained by deleting the edges of C.

If H is a component of G', there is a vertex h of H that is also in C, since G was connected. By the inductive hypothesis an Eulerian cycle D can be found on H, beginning and ending at h. Then the original cycle C can be extended by inserting D at the vertex h. This extension can be done for each component of G'. The resulting cycle will traverse every edge of G and so is an Eulerian cycle. □

Example 2.3. In the graph shown in Figure 8.10 let $C = \langle r, 6, 7, r\rangle$ and let H be the component consisting of the four vertices $\{3, 4, 5, 6\}$ and their incident edges. Then with $h = 6$ let $D = \langle 6, 3, 4, 5, 6\rangle$, an Eulerian cycle on H. This can be merged with C to form the larger cycle $C' = \langle r, 6, 3, 4, 5, 6, 7, r\rangle$.

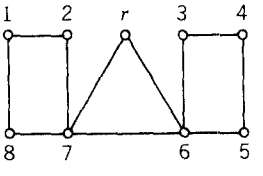


Figure 8.10

Theorem 2.1 also leads to conditions under which a graph contains an Eulerian path but not an Eulerian cycle. One such example was the graph H in Figure 8.7.

Corollary 2.2. A connected graph contains an Eulerian path, but not an Eulerian cycle, if and only if exactly two vertices have odd degree.

Proof. Suppose that x and y are the end vertices of the Eulerian path. Let G' be the graph obtained from G by creating a vertex r adjacent to x and y but to

no other vertex. If G contains an Eulerian path, then this path together with the two edges incident with r form an Eulerian cycle. Since by Theorem 2.1 G' has all vertices of even degree, G has exactly two vertices of odd degree. Conversely, if G is a connected graph with exactly two vertices of odd degree, then G', formed by adding a vertex r adjacent to the two vertices of odd degree, is connected and has every vertex of even degree. By Theorem 2.1 G' has an Eulerian cycle. We can imagine that this cycle begins at x and proceeds to r and then y. In G this cycle becomes an Eulerian path from y to x. □

Question 2.5. The graph in Figure 8.11 contains exactly two vertices of odd degree. Create an Eulerian graph as in the proof of Corollary 2.2. Find an Eulerian cycle on the larger graph and from that an Eulerian path on the original graph.

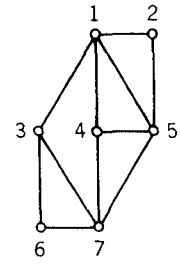


Figure 8.11

The proof of Theorem 2.1 is constructive. That is, it gives us the idea for an efficient algorithm for finding an Eulerian cycle.

Algorithm EULER

STEP 1. Input G, a connected graph with all vertices of even degree; set $C := \langle x \rangle$ {x arbitrary in $V(G)$}
STEP 2. While $|E(G)| > 0$ do
Begin
STEP 3. Pick x in $V(C)$ with $\deg(x, G) > 0$
STEP 4. Create a maximal cycle D beginning at x {D cannot be made longer by appending edges at its end.}
STEP 5. Set $E(G) := E(G) - E(D)$
STEP 6. Set $C :=$ cycle obtained from C by inserting D at x
End
STEP 7. Output C and stop.

Example 2.3 (again). Here is a trace (Table 8.2) of EULER, run on the graph shown in Figure 8.12.

Table 8.2

Step No.	x	C	D
1	r	$\langle r \rangle$	
3	r		
4			$\langle r, 6, 7, r \rangle$
6		$\langle r, 6, 7, r \rangle$	
3	6		
4			$\langle 6, 3, 4, 5, 6 \rangle$
6		$\langle r, 6, 3, 4, 5, 6, 7, r \rangle$	
3	7		
4			$\langle 7, 2, 1, 8, 7 \rangle$
6		$\langle r, 6, 3, 4, 5, 6, 7, 2, 1, 8, 7, r \rangle$	

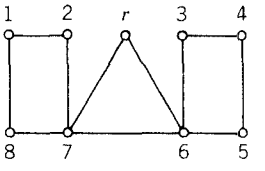


Figure 8.12

Notice that the cycles C and D can be created in any way consistent with the algorithm. For example, the initial cycle C might be $\langle r, 7, 8, 1, 2, 7, 6, r \rangle$ and the first cycle D might be $\langle 6, 5, 4, 3, 6 \rangle$. If this algorithm were implemented on a computer, which particular cycles C and D are created depends on how the graph G is stored. When tracing these cycles by hand, we can choose edges however we wish as long as a cycle is formed.

Question 2.6. Trace the algorithm EULER on the graph in Figure 8.13.

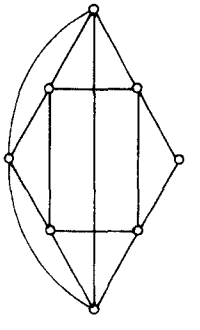


Figure 8.13

We have already proved that EULER works correctly, since it follows the proof of EULER's theorem.

Of course, to use EULER on an arbitrary graph G we would need to verify that G is connected. Since G is connected if and only if it contains a spanning tree by Exercise 5.3.13, connectivity could be checked using the algorithm SPTREE of Exercise 5.3.19 or KRUSKAL or BFS (of Section 8.1.) If G is connected, we could then determine if its vertex degrees are all even. If so, we could run EULER.

Before returning to the snowplowing problem, we consider the complexity of the algorithm EULER. We choose to count comparisons and note that step 4 is the critical step to examine. Suppose that the graph is input as an adjacency matrix. Creating cycle D in step 4 can be accomplished one edge at a time by finding a 1 in the row of the matrix that corresponds with the current vertex. It will take no more than V such comparisons to find the next edge. Thus step 4 may require $O(V \cdot E)$ comparisons. Actually, with appropriate data structures EULER can be made linear in E. (Details appear in Exercise 12.)

The algorithm EULER is designed to run on unweighted graphs. In the original snowplowing problem the related graph was a weighted one. The theory and algorithm so far deal only with a special case, when all vertices of the graph derived from the street system have even degree. In this case we can plow each city street exactly once, and regardless of the street lengths (or edge weights) we have found a minimum-weight cycle without any repetition.

Otherwise, what are the possibilities? Question 2.3 of Chapter 5 states that every graph contains an even number of vertices of odd degree. Thus there cannot be just one vertex of odd degree, but there might be exactly two vertices of odd degree in the graph of the streets. Suppose that City Hall is located at one vertex of odd degree. Then we could use an Eulerian path algorithm (see Exercise 13) to plow each street exactly once, ending at the other vertex of odd degree; call it z. To conclude, the plows would travel home from z to City Hall on a shortest path. This shortest path from z to City Hall could be found using DIJKSTRA with z as the root. In Exercises 8 to 10 you are asked to verify that this scheme does produce the overall most-efficient plowing plan when there are only two vertices of odd degree.

When there are four or more vertices of odd degree, the most-efficient snowplowing plan involves paths between pairs of these vertices of odd degree. (See Exercises 5 to 7). In fact, there is an efficient algorithm to solve this problem in full generality. It consists of pairing up the vertices of odd degree so that the sum of the distances of the minimum-distance paths between pairs is minimized. This is known as the **Minimum Weight Matching Problem** or the Chinese Postman Problem (named after the Chinese mathematician, M-K. Kwan, who first considered this problem.) The solution is complex, beyond the scope of this chapter.

But wait—there's something unsatisfactory about the snowplowing model. Most snowplows don't plow streets just once, but rather twice, once in each direction to clear both sides of the street. Can we devise an algorithm to traverse each

edge of a graph in this more realistic way? The answer is yes, quite easily, given our experience with Eulerian graphs.

We model this new situation with a directed graph, that is, a graph in which each edge is given an orientation or direction from one incident vertex to the other. In directed graphs paths and cycles must traverse each edge in its given orientation.

Definition. A **directed graph** G consists of a finite set $V(G)$ of vertices and a finite set $E(G)$ of edges (also called arcs) such that each edge consists of an ordered pair of distinct vertices of $V(G)$. We think of the edge $e = (x, y)$ as being directed from x to y.

Example 2.5. Figure 8.14 shows some directed graphs.

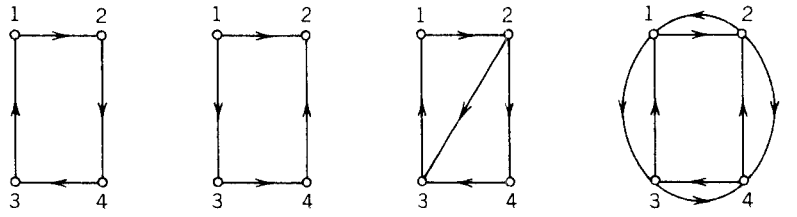


Figure 8.14

Definition. A **path in a directed graph** G from x to y is a sequence of distinct edges, $e_1, e_2, \ldots, e_k$, such that $e_1 = (x, x_1), e_2 = (x_1, x_2), \ldots, e_k = (x_{k-1}, y)$ for some vertices $x_1, x_2, \ldots, x_{k-1}$. *A* **cycle in a directed graph** is a path from a vertex to itself. An **Eulerian path** (respectively, **Eulerian cycle**) is a path (respectively, cycle) that includes every edge exactly once.

Question 2.7. For each graph in Example 2.5 find an Eulerian path or cycle if there is one.

Theorem 2.3. A directed graph whose underlying undirected graph is connected contains an Eulerian cycle if and only if at every vertex v the number of edges directed in to v equals the number of edges directed out of v.

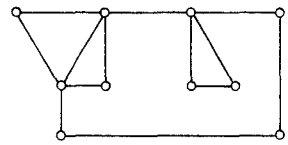
Proof. Follow the same proof as given for Theorem 2.1. □

Theorem 2.3 can be applied to solve the snowplow problem completely. If G is the (undirected) graph derived from the city street plan, let D be the directed graph created by replacing every edge of G by two directed edges, one in each direction. Then an efficient snowplowing plan would be an Eulerian cycle on D.

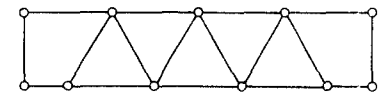
Since each edge in G is replaced by two directed edges in D, a vertex of degree k in G becomes a vertex in D with k edges directed in to it and k edges directed out of it. Hence the conditions of Theorem 2.3 are met and D contains an Eulerian cycle. How do we find an Eulerian cycle? Exercise 18 asks you to modify EULER so that it works on directed graphs.

EXERCISES FOR SECTION 2

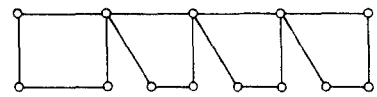
1. Which of the following graphs are Eulerian?



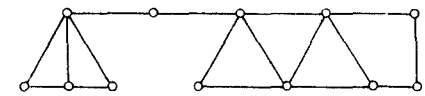
(a)



(b)

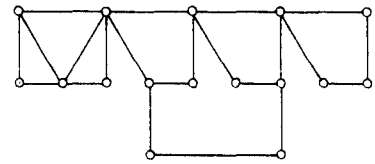


(c)

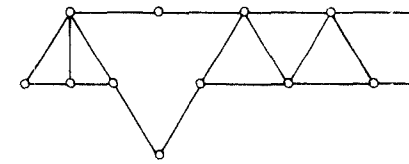


(d)

2. Can an Eulerian graph contain an odd number of edges? Either find an example or prove that there is none.
3. Suppose that $C = \langle x_1, x_2, \ldots, x_k, x_1 \rangle$ is a cycle in an Eulerian graph G with no repeated vertices (except for the start and finish at x_1). Show that there is an Eulerian cycle on G that visits the vertices $x_1, x_2, \ldots, x_k$ in the same order as in C, that is, for each $i = 1, 2, \ldots, k-1$, vertex x_i is reached before vertex x_{i+1}.
4. An Eulerian graph is called **arbitrarily traceable at a vertex** x if every cycle beginning and ending at x can be extended to an Eulerian cycle by continuing the cycle at x. Prove that G is arbitrarily traceable if and only if every cycle of G passes through x.
5. Show that the following graphs can have their edges divided into two paths, each joining two vertices of odd degree.



(a)

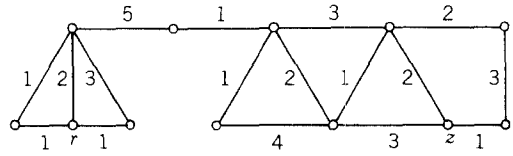


(b)

6. Show that any graph with exactly four vertices of odd degree can have its edges divided into two paths, one path joining a pair of the vertices of odd degree and the other path joining the other pair.

7. Show that a graph with exactly $2k$ vertices of odd degree can have its edges divided into k paths, each path joining a different pair of vertices of odd degree.

8. In the following graph, show that the shortest "snowplowing plan" that traverses each edge at least once consists of an Eulerian path from r to z and then a minimum-distance path from z to r.



9. Suppose that G is a connected (unweighted) graph that contains an Eulerian path but not an Eulerian cycle. Let r and s be the vertices of odd degree. Show that the "trail" that traverses every edge of G at least once, starting and ending at r, and uses the fewest edges, consists of an Eulerian path from r to s plus a shortest path back from s to r. (*Hint:* Let C be a minimum trail that traverses every edge at least once, and let $G^{\#}$ be the graph formed by adding an edge for each repeated use of an edge by C. Thus C is an Eulerian cycle on the graph $G^{\#}$. Study the graph formed from the new edges in $G^{\#}$. Actually, $G^{\#}$ will be a **multigraph**, i.e., some pairs of vertices can be joined by more than one edge.)

10. Answer the same question as in Exercise 9, only assume that the graph G is weighted and path length, as usual, is the sum of the edge weights of edges in the path.

11. Suppose that G is a connected weighted graph with exactly two vertices of odd degree and let v be an arbitrary vertex of G. Describe a shortest trail on G that begins and ends at v and covers each edge at least once. Prove that your trail is the shortest.

12. Here is a closer look at the complexity of EULER with an expanded version of the algorithm. Suppose that for each vertex v, $Nbor(v)$ initially contains a list of all vertices adjacent to v.

Algorithm EULER

STEP 1. Input G, a connected graph with all vertices of even degree and E edges; set $C = \langle x \rangle$ {x arbitrary in $V(G)$}

STEP 2. While $|E(G)| > 0$ do
Begin
STEP 3. Pick x in $V(C)$ with $\deg(x, G) > 0$
STEP 4. Call Trace (D, x) {The procedure Trace finds a maximal cycle D beginning at x}
STEP 5. Set $C :=$ cycle obtained from C by inserting D at x
End
STEP 6. Output C and stop.

Procedure Trace (D, x)

STEP 1. Set $z := x$
STEP 2. While $Nbor(z) \neq \varnothing$ do
Begin
STEP 3. Pick w in $Nbor(z)$
STEP 4. Add w to D
STEP 5. $Nbor(z) := Nbor(z) - \{w\}$; $Nbor(w) := Nbor(w) - \{z\}$
STEP 6. $E(G) := E(G) - (z, w)$
STEP 7. Set $z := w$
End
STEP 8. Return

In this version comparisons are made in steps 2 and 3 in the main program and in step 2 of Trace. Suppose that the algorithm cycles through (the main) step 2 s times, creating the cycles $D_1, D_2, \ldots, D_s$, which together form the Eulerian cycle. Find an upper bound on s in terms of E and then show that the total number of comparisons is $O(E)$.

13. Modify EULER so that upon input of a connected graph with exactly two vertices of odd degree, x and y, it traces an Eulerian path, beginning at x and ending at y.

14. Find conditions under which a directed graph contains an Eulerian path but not an Eulerian cycle. Prove your result.

15. Prove that an undirected graph G is Eulerian if and only if there is a way to direct its edges so that the resulting directed graph contains an Eulerian cycle.

16. Give an example of an undirected Eulerian graph G and then a direction on each of its edges so that the resulting directed graph does not contain an Eulerian cycle.

17. Explain how a directed graph can be stored in a $V \times V$ adjacency matrix.

18. Rewrite the algorithm EULER so that upon input of a directed graph, whose underlying undirected graph is connected, it finds an Eulerian cycle if there is one.

19. A **spanning in-tree** of a directed graph G with root r is defined to be a spanning tree of the underlying undirected graph such that each path P from a vertex y to r within the spanning tree is a directed path from y to r in G. Find examples of directed graphs that do and do not contain a spanning in-tree. Find an example of a graph G with root r that has a spanning in-tree but such that with some other vertex s as root there is no spanning in-tree.

20. Explain why in a spanning in-tree with root r, each vertex, except for r, has exactly one edge of the tree directed out of the vertex.

21. Let G be a directed graph that contains an Eulerian cycle and r an arbitrary vertex. Prove that G has a spanning in-tree with root r.

22. Devise an algorithm that upon input of a directed graph that contains an Eulerian cycle finds a spanning in-tree.

23. Suppose that G is a directed graph that contains an Eulerian cycle. Explain why the following is a valid algorithm for finding an Eulerian cycle in such a graph. First find a spanning in-tree with root r, as in the previous exercise. Then construct a cycle by appending incident, unused edges in any way except that at each vertex $v \neq r$ the unique out-directed edge in the spanning in-tree should be saved for the last exit from v.

24. A directed graph is called **strongly connected** if for every pair of vertices x and y there is a path from x to y and a path from y to x. A directed graph is called **connected** if for every pair of vertices x and y there is a path from one to the other. A directed graph is called **weakly connected** if the underlying undirected graph is connected. Find examples of the following types of directed graphs.
(*a*) The graph is strongly connected.
(*b*) The graph is connected but not strongly connected.
(*c*) The graph is weakly connected but not connected.
(*d*) The graph satisfies none of the connectivity definitions.

25. Explain why a graph that is strongly connected is also connected and why a graph that is connected is also weakly connected.

26. Prove that a directed graph that contains an Eulerian cycle is strongly connected. Is the same true for a directed graph that contains an Eulerian path? If so, explain why; if not, find additional conditions which when met by the graph ensure that it is strongly connected.

8:3 HAMILTONIAN CYCLES

Suppose that a mail carrier wants to pick up mail from every mailbox in town, or an inspector wants to check the traffic signals at every intersection. Can these jobs be accomplished efficiently by visiting each location exactly once? These

problems are modeled by constructing an appropriate graph with a vertex for each location and two vertices joined by an edge if there is a street connection that passes through no intermediate location. The problem is to find a cycle that passes through each vertex of the graph exactly once.

This graph theory problem is known as the **Hamiltonian cycle problem**, named for Sir W. R. Hamilton, the inventor of a related game (see Exercise 3). It is one of graph theory's most demanding unsolved problems. In this section we search for conditions that guarantee the existence of a Hamiltonian cycle and an efficient algorithm to find such a cycle. We also learn the important algorithmic technique of depth-first search. This can be used to solve the Hamiltonian cycle problem for an arbitrary graph, although not efficiently. In the following section we turn to the equally challenging problem of finding a minimum-weight Hamiltonian cycle in a weighted complete graph.

Question 3.1. Which of the graphs in Figure 8.15 contains a cycle that visits each vertex exactly once?

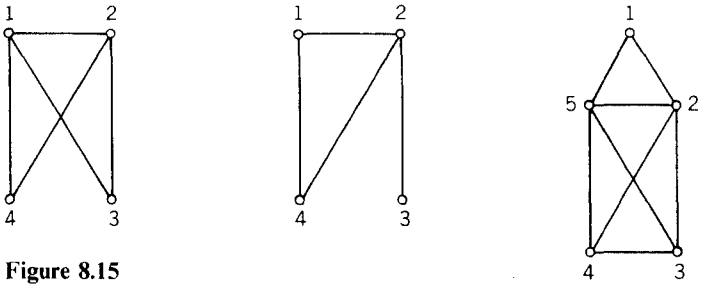


Figure 8.15

Definitions. In a graph G with V vertices, a path (or cycle) that contains exactly $V - 1$ (respectively, V) edges and spans G is called **Hamiltonian**. A graph is called **Hamiltonian** if it contains a Hamiltonian cycle.

Problem. Given a graph G, is it Hamiltonian? If so, find a Hamiltonian cycle.

Notice how similar this problem seems to be to that of Eulerian cycles (of Section 8.2.) In that case we found necessary and sufficient conditions for a connected graph to be Eulerian. In the case of Hamiltonian graphs life is not so simple and no such nice characterization is known (or likely to be discovered).

First we remark that since a Hamiltonian graph must contain a spanning cycle, every vertex must have degree at least 2. Exercises 4 and 5 develop more involved conditions that a graph must satisfy in order to be Hamiltonian.

Question 3.2. Find an example of a connected graph that is not Hamiltonian but does not contain any vertices of degree 1.

The r-clique, with $r \geq 3$, contains a Hamiltonian cycle, namely $\langle 1, 2, \ldots, r, 1 \rangle$. More generally, graphs with all vertices of relatively high degree contain Hamiltonian cycles as seen in the first theorem.

Theorem 3.1. If G has $V \geq 3$ vertices and every vertex has degree at least $V/2$, then G is Hamiltonian.

Proof. The proof is constructive and will lead to an algorithm HAMCYCLE. There are two principal steps. In the first we take a maximal path (i.e., a path that cannot be extended at either end) and find a cycle on the same set of vertices. In the second step we take any vertex not on the cycle and construct a maximal path using it and all of the vertices of the cycle. This new path will be longer than the original one. We continue alternating these two steps until all of the vertices of the graph are in the maximal path whence the next cycle we create will be Hamiltonian.

We first note that any maximal path must contain more than half the vertices of the graph. Suppose that $\langle x = x_1, x_2, \ldots, x_k = y \rangle$ forms a maximal path within the graph G. Since the path is maximal, x cannot be adjacent to any vertex off the path. Thus x has at most $k - 1$ neighbors. Since the degree of x is at least $V/2$, we know that

$$k - 1 \geq \frac{V}{2} \qquad \text{or} \qquad k \geq 1 + \frac{V}{2}.$$

Now we want to find a cycle whose vertex set is the same as that of the maximal path. If x is adjacent to y, then the vertices of the path form a cycle in their natural order. If there exist vertices x_i and x_{i+1} such that x is adjacent to x_{i+1} and y is adjacent to x_i, then $\langle x, x_{i+1}, x_{i+2}, \ldots, y, x_i, x_{i-1}, \ldots, x_2, x \rangle$ is a cycle containing all the vertices of the original path. See Figure 8.16. On the other hand, if there were no such pair x_i and x_{i+1}, then whenever x is adjacent to x_{i+1}, y is not adjacent to x_i. Since the path is maximal, neither x nor y can be adjacent to any vertex off the path. Thus if $\deg(x) = s$, then $\deg(y) \leq k - 1 - s$. Since $\deg(x) \geq V/2$ and $\deg(y) \geq V/2$,

$$V \leq \deg(x) + \deg(y) \leq s + (k - 1 - s) = k - 1 < V,$$

a contradiction.

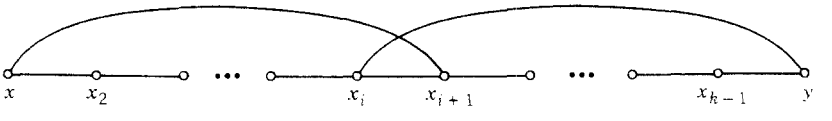


Figure 8.16

Thus we can create a cycle on any vertex set from a maximal path; we relabel vertices so that the resulting cycle is $\langle x_1, \ldots, x_k, x_1 \rangle$.

To justify the second step, let z be any vertex not contained in the cycle $\langle x_1, \ldots, x_k, x_1 \rangle$. Since, as we saw above,

$$k \geq 1 + \frac{V}{2} \qquad \text{and} \qquad \deg(z) \geq \frac{V}{2},$$

z must be adjacent to at least one vertex on the cycle. If z is adjacent to x_i, then

$$\langle z, x_i, x_{i+1}, \ldots, x_k, x_1, \ldots, x_{i-1} \rangle$$

forms a path in G with $k + 1$ vertices. This can be extended to a maximal path. □

Here is an algorithm suggested by the proof of Theorem 3.1.

Algorithm HAMCYCLE

STEP 1. Input G, a graph with V vertices all of degree $\geq V/2$
STEP 2. Set $P := \varnothing$
STEP 3. Repeat
Begin
STEP 4. Pick z in $V(G) - V(P)$; set $P :=$ a maximal path containing $V(P)$ and z
STEP 5. Find C a cycle on $V(P)$
End
until $|V(C)| = V$
STEP 6. Output C and stop.

Example 3.1. Table 8.3 is a trace of HAMCYCLE as applied to the graph in Figure 8.17.

Table 8.3

Step No.	z	P	C
2		$\varnothing$	
4	x	$\langle x, v, t, w, y \rangle$	
5			$\langle x, t, w, y, v, x \rangle$
4	u	$\langle u, t, w, y, v, x \rangle$	
5			$\langle u, w, y, v, x, t, u \rangle$

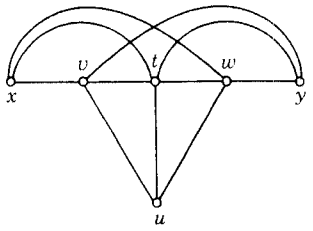


Figure 8.17

Question 3.3. Trace HAMCYCLE on the graph in Figure 8.18.

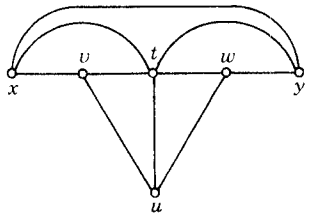


Figure 8.18

How efficient is HAMCYCLE? It is a $O(V^2)$ algorithm; here's why. Constructing the first maximal path in step 4 will require fewer than V comparisons to add each vertex and thus fewer than V^2 comparisons in total. The path has length at least $V/2$ and so the Repeat ... Until loop repeats at most $V/2$ times. In subsequent executions of step 4 the addition of the vertex z will require no more than one comparison for each vertex in P and so fewer than V in total. Constructing a maximal path in step 4 requires fewer than V comparisons for each additional vertex. Thus the total number of comparisons in step 4 is $V^2 + (V/2)(V + V) = 2V^2$. Finding C in step 5 can be accomplished with no more than two comparisons for each edge in P and thus fewer than $2V$ for the step. Thus the total number of comparisons in step 5 is V^2 and HAMCYCLE is $O(V^2)$.

Suppose that we want to find a Hamiltonian cycle in an arbitrary graph, one with all sorts of different degrees. We might try to list all cycles in the graph, but that certainly sounds like the basis of an exponential algorithm. (Recall that algorithms that list all possible subsets, like J-SET and BADMINTREE, are exponential.) Why don't we just try to build as long a path as possible, and then check that it can be completed to a cycle? (The phrase, "Why don't we just ...," is a famous one in algorithms. Often there seems to be a simple way to proceed, but the heart of the matter is then proving that the resulting algorithm always works and is efficient.)

We pursue the idea of hunting for a longest path. This technique, known as **depth-first search** (or **DFS**), is a method for systematically visiting all vertices of a graph by traversing paths that are as long as possible.

Example 3.2. Suppose that we want to visit all vertices in the graph shown in Figure 8.19.

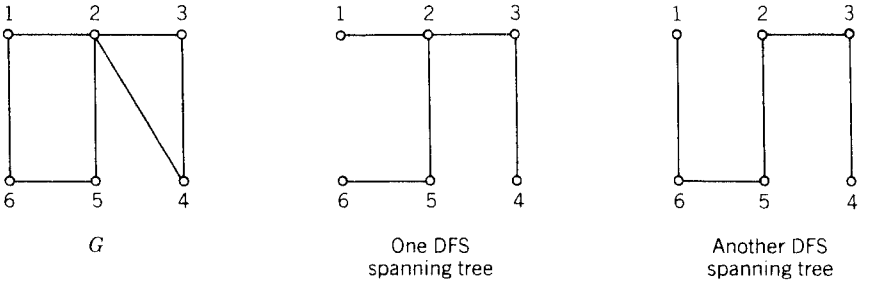


Figure 8.19

Beginning at vertex 1 we might create a path $P = \langle 1,2,3,4 \rangle$. From vertex 4 we can visit no additional new vertices. We backup to vertex 3 from which we also cannot visit new vertices. Then we backup to vertex 2 and visit vertices 5 and then 6. If we keep track of the edges traversed in this process, we find a spanning tree. This tree is known as a **depth-first-search** (or **DFS**) **spanning tree**. As with breadth-first search, a DFS spanning tree is not uniquely determined.

First we present this technique as an algorithm, designed to visit vertices and to construct a spanning tree if possible; in later applications we shall embellish upon this fundamental depth-first-search procedure. As vertices are visited, they and their adjoining edges are placed in T and $E(T)$, the vertices and edges of a DFS tree. We may use the edges in $E(T)$ to backup if need be.

Algorithm DEPTHFIRSTSEARCH (DFS)

STEP 1. Initialize
Input G, a graph with vertices $1,\ldots,V$ and edge set $E(G)$
Set $J := 1$ {J will index the vertex currently visited.)
Set $T := \{1\}$ {T will contain the visited vertices.}
Set $E(T) := \varnothing$ {$E(T)$ will contain the edges of the DFS tree.}
STEP 2. While $|T| < V$ do
STEP 3. If there is a K in $G - T$ such that (J,K) is in $E(G)$, then do
Begin
STEP 4. $T := T + K$
STEP 5. $E(T) := E(T) + (J,K)$
STEP 6. $J := K$
End

Else {no such K}

STEP 7. If $J \neq 1$, then do {backup}

Find (I, J) in $E(T)$ and set $J := I$

Else $\{J = 1\}$

STEP 8. Output T, $E(T)$ and stop.

STEP 9. Output $E(T)$ and stop.

Example 3.3. Table 8.4 is a trace of DFS run on the graph in Figure 8.20.

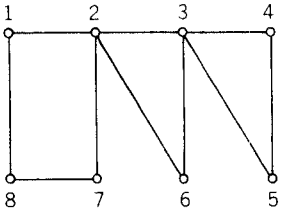


Figure 8.20

Table 8.4

Step No.	J	K	T	$E(T)$
1	1		{1}	∅
3	1	2		
4–6	2	2	{1, 2}	{(1, 2)}
3	2	3		
4–6	3	3	{1, 2, 3}	{(1, 2), (2, 3)}
3	3	4		
4–6	4	4	{1, 2, 3, 4}	{(1, 2), (2, 3), (3, 4)}
3	4	5		
4–6	5	5	{1, 2, 3, 4, 5}	{(1, 2), (2, 3), (3, 4), (4, 5)}
3	5	{no K}		
7	4			
3	4	{no K}		
7	3			
3	3	6		
4–6	6	6	{1, 2, 3, 4, 5, 6}	{(1, 2), (2, 3), (3, 4), (4, 5), (3, 6)}
3	6	{no K}		
7	3			
3	3	{no K}		
7	2			
3	2	7		
4–6	7	7	{1, 2, 3, 4, 5, 6, 7}	{(1, 2), (2, 3), (3, 4), (4, 5), (3, 6), (2, 7)}
3	7	8		
4–6	8	8	{1, 2, 3, 4, 5, 6, 7, 8}	{(1, 2), (2, 3), (3, 4), (4, 5), (3, 6), (2, 7), (7, 8)}
9 Stop.				

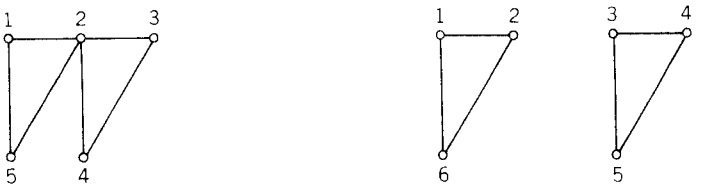


Figure 8.21

Question 3.4. Trace DFS on the graphs in Figure 8.21.

Theorem 3.2. DEPTHFIRSTSEARCH terminates with T containing precisely those vertices that are in the same component as vertex 1. The algorithm performs $O(V^2)$ comparisons.

Proof. Note that if vertex v is not in the same component as vertex 1, then there is no path connecting 1 and v and so v cannot be added to the tree T.

Conversely, suppose that v is a vertex in G that is in the same component as the vertex 1, yet v is not in T after DFS is executed. There is a path from 1 to v. Assume that the edge (u, v) is on that path and that u is in T. (Otherwise, replace v by u in this argument.) Since v is never added to T, $|T| < V$ and the algorithm terminates in step 8. Yet whenever DFS examines the vertex u in step 3, there is a vertex K available, namely $K = v$, and so DFS must add v to T before J is decreased back to 1 in step 8.

To count comparisons in DFS, we first note that the loop at step 2 is executed fewer than V times. Each pass through the loop requires one comparison at steps 2 and 7 and fewer than V comparisons at step 3. Thus DFS requires fewer than $V(V + 2) = O(V^2)$ comparisons in total. □

In contrast we also offer a recursive version of DFS; here the internal backing up (in step 7 of DFS) is managed by the recursive calls.

Procedure R-DFS(J) {The procedure begins a depth-first search at vertex J.}

STEP 1. $T := T + J$
STEP 2. For each edge (J, K) do
  STEP 3. If K is in $G - T$, then do
    Begin
    STEP 4. $E(T) := E(T) + (J, K)$
    STEP 5. Call Procedure DFS(K)
    End
STEP 6. Return.

Then a depth-first search is performed by the following.

Algorithm R-DEPTHFIRSTSEARCH

STEP 1. Input the graph G

STEP 2. $T := \varnothing$; $E(T) := \varnothing$

STEP 3. Call Procedure DFS(1)

STEP 4. Output T, $E(T)$, and stop.

More analysis of this recursive approach is contained in Exercise 21.

Our immediate aim is to use DEPTHFIRSTSEARCH (or R-DEPTHFIRST-SEARCH) to try essentially all possible ways to construct a Hamiltonian cycle. First we create as long a path as possible. If this can be completed to a Hamiltonian cycle, we are done. If not, we backup, throwing away vertices on the path and trying to find another way to extend.

Example 3.4. Here is the idea of a depth-first search for a Hamiltonian cycle on the graph shown in Figure 8.22. We begin creating as long a path as possible: $\langle 1, 2, 3, 4 \rangle$. This cannot be completed to a cycle. We backup to 3, but there is no way to make a new path. When we backup to 2, we can start a new path $\langle 1, 2, 4 \rangle$, which gets completed to the Hamiltonian cycle $\langle 1, 2, 4, 3, 1 \rangle$.

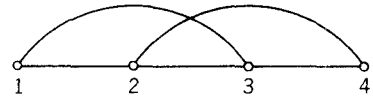


Figure 8.22

In this variation on depth-first search, the variable PATH will contain the vertices of a path that we hope can be completed to a Hamiltonian cycle; entries are initialized as 0. When we backup, we must delete vertices from PATH, resetting entries to 0. The procedure BUILD is used to find the next entry to be added to PATH. As long as vertices are being added to PATH the variable FORWARD equals TRUE, but when backing up FORWARD is FALSE.

Algorithm DFS-HAMCYCLE

STEP 1. Initialize
Input G, a graph with vertices $1, \ldots, V$
Set PATH(1) := 1, PATH(I) := 0 for $I = 2, \ldots, V + 1$
Set $J := 2$ {J indexes the entry of PATH that is currently sought.}
Set FORWARD := TRUE

STEP 2. While $J < V + 1$ do
Begin

STEP 3. If FORWARD = TRUE do
Call Procedure BUILD

```
        Else {FORWARD = FALSE}
                Begin
                STEP 4.  PATH(J) := 0
                STEP 5.  J := J - 1
                STEP 6.  If J ≠ 1, then
                             Call Procedure BUILD
                         Else {J = 1}
                             Output "No Ham cycle" and stop.
                         End {step 3}
        STEP 7.  If J = V + 1, then do
                STEP 8.  If (1, PATH(V)) is in E(G), then
                             PATH(J) := 1
                         Else
                             FORWARD := FALSE
        End {step 2}
STEP 9. Output PATH and stop.
```

Procedure BUILD

```
STEP 1. Find smallest K > PATH(J) such that K ≠ PATH(I) for I < J and
        (PATH(J - 1), K) is in E(G)
STEP 2. If there is no such K, then
        Set FORWARD := FALSE
Else
        Begin
        STEP 3.  PATH(J) := K
        STEP 4.  J := J + 1
        STEP 5.  FORWARD := TRUE
        End {step 2}
End.
```

Note that this algorithm parallels DFS; however, the K chosen in step 1 of the procedure is specified.

Example 3.5. Table 8.5 is a trace of DFS-HAMCYCLE on the graph from Figure 8.22.

Question 3.5. Trace DFS-HAMCYCLE on one graph of Figure 8.15.

In Exercises 13 and 14 you are asked to show that DFS-HAMCYCLE is correct. It would be nice if it were $O(V^2)$. Unfortunately, the seemingly small changes from DEPTHFIRSTSEARCH make the algorithm no longer a polynomial algorithm. (See Exercise 24.)

The Hamiltonian cycle problem of this section is not just an exercise in frustration for the reader—it is equally frustrating for researchers in graph theory and

Table 8.5

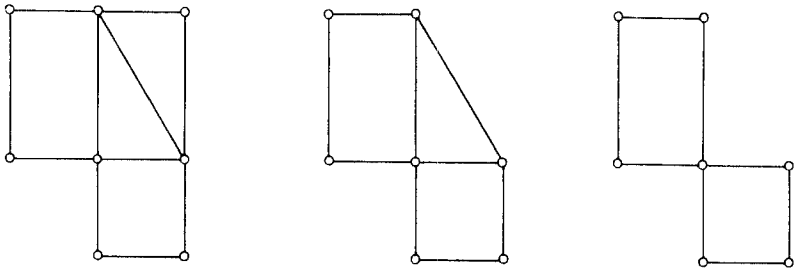
Step No.		*J*	*K*	*FORWARD*	*PATH*
Main	1	2		TRUE	⟨1, 0, 0, 0, 0⟩
BUILD	1	2	2		
	3	2			⟨1, 2, 0, 0, 0⟩
	4	3			
	5			TRUE	
BUILD	1	3	3		
	3–5	4	3	TRUE	⟨1, 2, 3, 0, 0⟩
BUILD	1	4	4		
	3–5	5	4	TRUE	⟨1, 2, 3, 4, 0⟩
Main	7, 8	5		FALSE	
Main	4	5			⟨1, 2, 3, 4, 0⟩
	5	4			
BUILD	1	4	{no *K*}		
	2			FALSE	
Main	4	4			⟨1, 2, 3, 0, 0⟩
	5	3			
BUILD	1	3	4		
	3–5	4	4	TRUE	⟨1, 2, 4, 0, 0⟩
BUILD	1	4	3		
	3–5	5	3	TRUE	⟨1, 2, 4, 3, 0⟩
Main	7, 8	5			⟨1, 2, 4, 3, 1⟩
Main	9	STOP			

graph algorithms! This is one of a collection of problems for which there is no known polynomial algorithm and for which it has not been proved that there cannot be a polynomial algorithm, as yet undiscovered. The question of the complexity of the Hamiltonian cycle problem has been shown to be equivalent to a large collection of similarly unsolved problems; these are known as the **NP-Complete problems**. Two other NP-Complete problems are the Traveling Salesrepresentative Problem, mentioned in Section 5.5, and the Satisfiability Problem of Section 1.10. The NP-Complete problems are equivalent in the sense that if a good algorithm is found for one of these problems, then there are good algorithms for all NP-Complete problems. Conversely, if one of these problems is intractable, then the same is true for all the NP-Complete problems. The NP-Complete problems are intensively studied, and rumors of proofs (and of false proofs!) circulate through the research community. There is an expectation that the issue should be resolved soon. Most algorithmics experts would be surprised if there were good algorithms for the NP-complete problems. (We place no bets on how soon or which way these problems will be resolved . . . .)

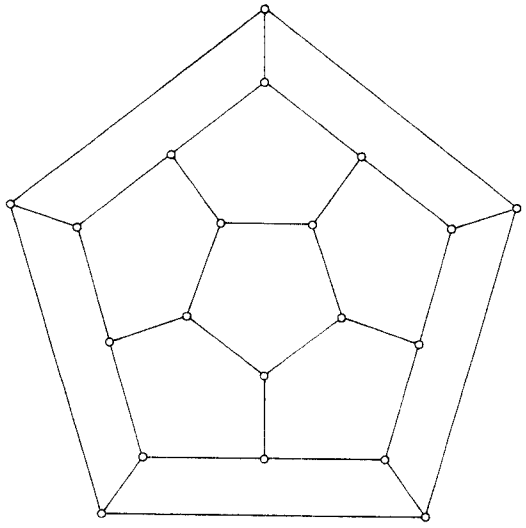
In the next two sections we shall discuss instances of two other NP-Complete problems and shall demonstrate different ways of coping with the unsettled state of affairs.

EXERCISES FOR SECTION 3

1. Which of the following graphs contain a Hamiltonian cycle? Which contain a Hamiltonian path but not a Hamiltonian cycle? Give reasons for your answers.



2. Find examples of the following types of graphs:
   (*a*) Eulerian and Hamiltonian.
   (*b*) Eulerian but not Hamiltonian.
   (*c*) Hamiltonian but not Eulerian.
   (*d*) connected but neither Hamiltonian nor Eulerian.
3. Here is the original graph on which Hamilton based his game; find a Hamiltonian cycle in this graph. One version of the original game consisted of one player selecting a path with five vertices and the second player attempting to



extend the path to a Hamiltonian cycle. Is there a path with five vertices that cannot be extended to a Hamiltonian cycle?

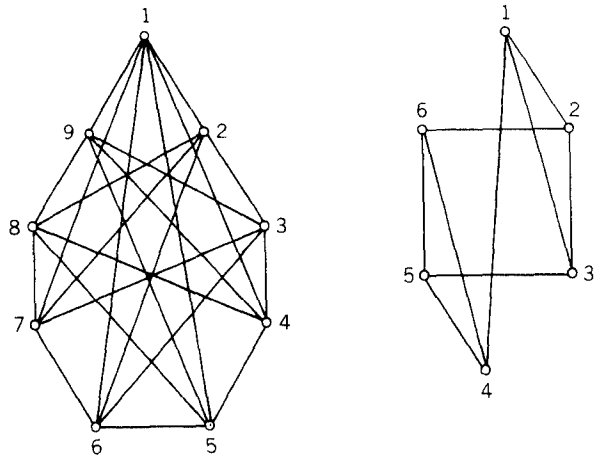
4. Show that if G, a connected graph, contains a vertex v whose removal leaves a disconnected graph, then G is not Hamiltonian. Is the converse true?
5. Let S be a set of vertices of a graph G and let $c(G - S)$ denote the number of connected components in the graph obtained by deleting the vertices of S and all incident edges. Prove that if G is Hamiltonian, then for every nonempty subset S, $c(G - S) \leq |V(S)|$. Is the converse true?
6. Show that the complete bipartite graph $K_{r,s}$ is Hamiltonian if and only if $r = s$.
7. Can a connected bipartite graph with an odd number of vertices be Hamiltonian? Is every connected bipartite graph with an even number of vertices Hamiltonian?
8. Imagine the graph that consists of two copies of $K_{V/2}$ (where V is even) joined by exactly one edge. Use this graph to argue that the hypothesis of Theorem 3.1 (that the degrees are at least $V/2$) cannot be weakened and still have the conclusion hold.
9. Explain why a graph with V vertices and all vertices of degree at least $V/2$ is connected.
10. Suppose that G is a graph with V vertices such that for every pair of nonadjacent vertices, say x and y, the degree of x plus the degree of y is at least V. Show that G is Hamiltonian. Compare this result with Theorem 3.1.
11. Where in the proof of Theorem 3.1 is the assumption that $V \geq 3$ needed?
12. Where does the algorithm HAMCYCLE use the fact that all vertices have degree at least $V/2$? Can this number be replaced by $V/2 - 1$ or by any smaller number, like $2V/5$ or $3V/8$?
13. Prove that DFS-HAMCYCLE is correct when it outputs the edges of a Hamiltonian cycle.
14. Prove that if DFS-HAMCYCLE terminates without finding a Hamiltonian cycle, then G does not contain such a cycle. (*Hint:* Try a proof by contradiction.)
15. Here is an another algorithm to find a Hamiltonian cycle in a graph with all vertices of degree at least $V/2$.

Algorithm HAMCYCLE2

STEP 1. Input G with V vertices, all of degree $\geq V/2$, label each edge with a 0, set label $:= 1$ and $H := G$
STEP 2. While H contains nonadjacent vertices x and y do
Begin
STEP 3. Set $H := H + (x, y)$

STEP 4. Label (x, y) with the value of label
STEP 5. Set label := label + 1
End
STEP 6. Find C a Hamiltonian cycle in H and let k be the largest label on an edge of C
STEP 7. Repeat
Begin
STEP 8. Find the edge (x, y) of label k
STEP 9. Renumber the vertices of C as
$C = \langle x = x_1, x_2, \ldots, x_V = y \rangle$
STEP 10. Find an index i such that x is adjacent to x_i and y is adjacent to x_{i-1}
STEP 11. Set
$C := \langle x = x_1, x_i, x_{i+1}, \ldots, x_V = y, x_{i-1}, x_{i-2}, \ldots, x_1 \rangle$
STEP 12. Delete the edge (x, y)
STEP 13. Set $k :=$ largest label on an edge of C
End
Until $k = 0$
STEP 14. Output C and stop.

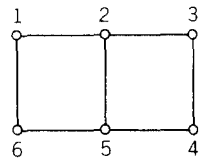
Run HAMCYCLE2 on the two following graphs.

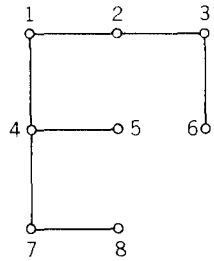


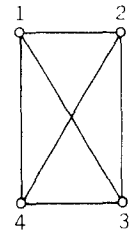
16. Prove that HAMCYCLE2 always produces a Hamiltonian cycle in a graph with all vertices of degree $V/2$ or more.

17. Is HAMCYCLE2 more efficient than HAMCYCLE?

18. Find an example such that the spanning trees created by breadth-first search and depth-first search differ.





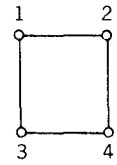


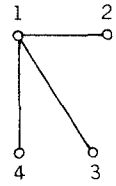
19. Run DEPTHFIRSTSEARCH on the following graphs.

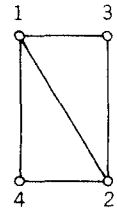
20. (*a*) Modify DEPTHFIRSTSEARCH so that it outputs a spanning tree if the graph is connected and a spanning forest otherwise.
(*b*) Modify DFS further so that it identifies and outputs the vertices in each connected component of the graph.

21. (*a*) Run R-DEPTHFIRSTSEARCH on the graphs of Exercise 19.
(*b*) Prove the analogue of Theorem 3.2 for R-DEPTHFIRSTSEARCH.

22. Run DFS-HAMCYCLE on the following graphs.







23. Construct an algorithm that uses depth-first search to find a Hamiltonian path in a graph or else reports that there is none.

24. Find a family of (possibly disconnected) graphs on V vertices such that DFS-HAMCYCLE creates more than $(V-3)!$ paths, none of which extend to a Hamiltonian cycle.

25. Write a recursive program that searches for a Hamiltonian path in a graph.

26. Write a recursive program that searches for a Hamiltonian cycle in a graph.

27. If G is a graph, we define the **line graph** of G, denoted $L(G)$, to be the graph formed with a vertex in $L(G)$ for every edge of G and two vertices adjacent in $L(G)$ if the corresponding edges of G are incident. Prove that if a graph G is the line graph of an Eulerian graph (i.e., there is an Eulerian graph H such that $L(H) = G$), then G is Hamiltonian. Find examples of such graphs G and H.

28. Find an example of a graph G that is not the line graph of any other graph; that is, there is no graph H such that $G = L(H)$.

29. Show that if G is Eulerian, then so is $L(G)$.

30. Prove that G is the line graph of some graph if and only if the edges of G can be divided into a disjoint collection of maximal complete subgraphs.

31. Three mutually adjacent vertices T in a graph G are said to form an **even triangle** if every vertex of G is adjacent to zero or two vertices of T. Otherwise, three mutually adjacent vertices are said to form an odd triangle. Prove that a graph G is the line graph of another graph if and only if (i) G does not contain two odd "overlapping" triangles of the form $\{a,b,c\}$ and $\{a,b,d\}$ and (ii) G does not contain $K_{1,3}$ as an induced subgraph.

8:4 MINIMUM-WEIGHT HAMILTONIAN CYCLES

A common manufacturing task is drilling holes in a sheet of plastic using a laser. Think of the plastic as being in a fixed location in the horizontal plane. The drill is movable and must be located immediately above the target hole while drill ng. The time it takes to fabricate one unit equals the time it takes to drill the holes together with the time it takes to move the drill. With a given drill and plastic sheet, the drilling time will be a (small) constant. The time it takes to position the drill will depend on the distance the drill must travel. Thus the manufacturer would like to drill the holes in an order that minimizes the total distance the drill travels.

Question 4.1. Suppose that the drill is initially located above the origin in the x-y plane and that after drilling four holes the drill must return to the origin. Suppose the holes are located at $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$. Find all possible drilling sequences and for each find the total distance that the drill travels.

Efficient fabrication leads to the following graph theory problem. Construct a complete graph whose vertices correspond with the locations of the holes. Since this graph is complete, it contains lots of Hamiltonian cycles. We want a shortest one. Assign to the edge (x,y) a weight that represents the distance from the hole labeled x to the hole labeled y. A minimum-weight Hamiltonian cycle in this graph corresponds with a most efficient ordering of the vertices for drilling.

Problem. Given a weighted complete graph, find a Hamiltonian cycle whose total weight is minimum.

The only known algorithms for this problem are exponential (see Exercise 3). A typical manufacturing application might have hundreds or even thousands of holes to drill. Thus a nonpolynomial algorithm for finding a minimum-weight spanning cycle would be impossibly slow. Still in practice the drilling is done with the holes processed in some order.

This section presents a good approximation algorithm for the laser drilling problem. Specifically, given a weighted complete graph whose edge weights represent distances in the plane, the algorithm will produce a Hamiltonian cycle whose

total weight is no more than twice the minimum. Before giving the algorithm formally, we discuss and analyze the major steps in the approximation.

Given a weighted complete graph G, let $H(G)$ denote a minimum-weight Hamiltonian cycle and let $P(G)$ denote a minimum-weight Hamiltonian path. Of course, we don't know a good way to find these subgraphs. However, we can use KRUSKAL (see Section 5.4) to find instead $T(G)$, a minimum-weight spanning tree of G. This will be the first major step in the approximation algorithm.

Example 4.1. Suppose that the vertices of G correspond with the points $(0,0)$, $(1,1)$, $(2,0)$, and $(0,2)$. Figure 8.23 exhibits $T(G)$, $P(G)$, and $H(G)$. Note that $w(T(G)) = 3\sqrt{2}$, $w(P(G)) = 2 + 2\sqrt{2}$, and $w(H(G)) = 4 + 2\sqrt{2}$.

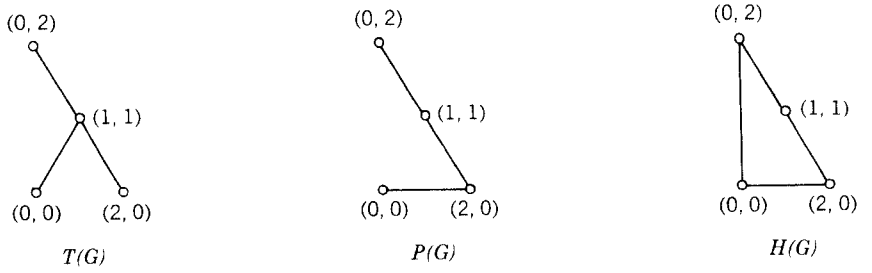


Figure 8.23

Question 4.2. Given the points $(0,0)$, $(1,1)$, $(1,-1)$, $(-1,1)$ and $(-1,-1)$, construct G, $T(G)$, $P(G)$, and $H(G)$ as in the preceding example. Find the weights of each of these graphs.

Proposition 4.1. $w(T(G)) \leq w(P(G)) < w(H(G))$.

Proof. A Hamiltonian path can be obtained from a Hamiltonian cycle by removing an edge (necessarily of positive weight). Thus $w(P(G)) < w(H(G))$. Since $P(G)$ is a path, it is a connected acyclic graph and thus a tree. Since $T(G)$ is a minimum-weight spanning tree of G, its weight can be no more than that of $P(G)$. Thus $w(T(G)) \leq w(P(G))$. □

Note that Proposition 4.1 is true for any weighted graph whose edge weights are positive. In particular, the edge weights don't have to correspond with distances in the plane.

Our second major step will be to find an Eulerian cycle, using material from Section 8.2. (For an alternate explanation, independent of that section, see Exercise 12.) Given any tree T, create a directed graph $D(T)$ by replacing each edge e of T with two directed edges, one in each direction and both of weight $w(e)$. By construction every vertex of $D(T)$ has an equal number of edges directed in and out. By Theorem 2.3 $D(T)$ contains a (directed) Eulerian cycle. We could use either

a modified version of EULER or a depth-first search to create such a cycle. Evidently, $w(D(T)) = 2w(T)$. Let D denote an Eulerian cycle in $D(T)$.

Example 4.1 (continued). Figure 8.24 shows the graph $D(T)$. D might consist of $\langle r, c, b, c, d, c, r \rangle$.

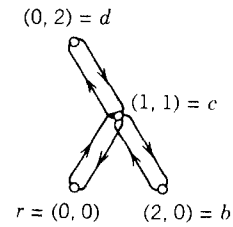


Figure 8.24

Question 4.3. If G is the weighted graph from Question 4.2, find $D(T)$ and some Eulerian cycle D.

The third major step in our approximation algorithm will be to transform D into a Hamiltonian cycle, denoted by C. We need some bookkeeping and a standard look ahead device. Initially, set $C := D$, and let r denote the first vertex of C. Begin traversing C, noting each visited vertex. Continue until you reach a vertex x and an out-directed edge (x, y) such that y is an already visited vertex. Suppose that within D, the edge (x, y) is followed by the edge (y, z). Modify C by replacing (x, y) and (y, z) with (x, z). If z is already visited and (x, z) is followed by (z, w), then replace (x, z) and (z, w) by (x, w), and so on. If the edge (x, y) is not followed by another edge, then $y = r$, and C is the desired Hamiltonian cycle.

Example 4.1 (once again). C initially consists of $\langle r, c, b, c, d, c, r \rangle$. Note that (b, c) visits c for the second time. Thus (b, c) and (c, d) are replaced by (b, d). At this stage $C = \langle r, c, b, d, c, r \rangle$. Now (d, c) visits c for the second time. Thus (d, c) and (c, r) are replaced by (d, r), and $C = \langle r, c, b, d, r \rangle$. Although (d, r) visits r for a second time, it is not followed by another edge. Thus (d, r) is the final edge in the Hamiltonian cycle. Figure 8.25 exhibits the Hamiltonian cycle thus obtained. Note that $w(C) = 2 + 4\sqrt{2}$.

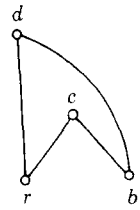


Figure 8.25

Question 4.4. Using the Eulerian cycle from Question 4.3, find a Hamiltonian cycle C. Determine $w(C)$ and using the results of Question 4.2 verify that $w(C) < 2w(H(G))$.

Proposition 4.2. After the edge replacements are complete (as described in the previous paragraph), C is a Hamiltonian cycle and $w(C) < 2w(H(G))$.

Proof. Initially, D is an Eulerian cycle in the directed graph $D(T)$. Since T is a spanning tree of G, D must visit every vertex of G. After all the edge replacements are complete, transforming D to C, no vertex is visited twice other than r. Thus C is a Hamiltonian cycle. By construction

$$w(D) = w(D(T)) = 2w(T(G)) < 2w(H(G)) \qquad \text{by Proposition 4.1.}$$

If $w(C) \leq w(D)$, then the proof is complete. Suppose that (x, y) and (y, z) in D are replaced by (x, z). Since $w(x, z)$ equals the distance from x to z and $w(x, y) + w(y, z)$ equals the distance from x to y plus the distance from y to z, the triangle inequality of plane geometry implies

$$w(x, z) \leq w(x, y) + w(y, z).$$

Thus every time two edges are replaced by one, the weight of the cycle cannot increase. Thus $w(C) \leq w(D)$. □

Note that we have solved the minimum-weight Hamiltonian cycle problem not only in the case where edge weights represent distances in the plane, but also for every weighted complete graph such that $w(x, z) \leq w(x, y) + w(y, z)$ for every triple of vertices x, y, and z. (See also Exercise 7.)

Algorithm APPROXHAM

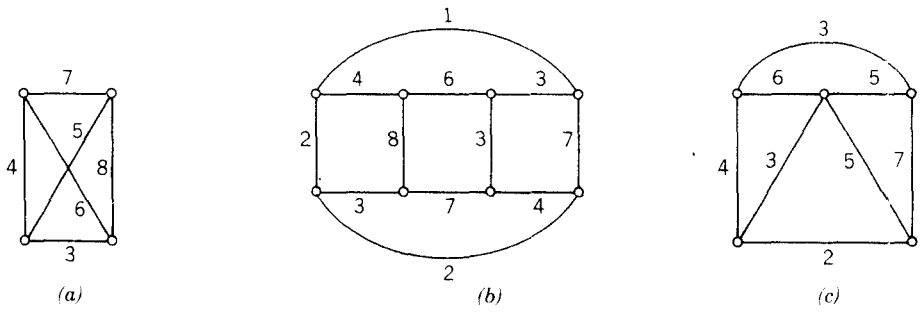
STEP 1. Input G, a weighted complete graph
STEP 2. Use KRUSKAL to obtain T, a minimum-weight spanning tree of G
STEP 3. Create the digraph $D(T)$ by replacing every edge of T by two oppositely directed edges
STEP 4. Use EULER, modified as in Exercise 2.18, to obtain D an Eulerian cycle in $D(T)$ {Suppose that $D = \langle e_1, \ldots, e_{2V-2} \rangle$ and for all j, $e_j = (x_j, y_j)$.}
STEP 5. Set $C := \langle x_1 \rangle$ and mark x_1 visited
STEP 6. For $j = 1$ to $2V - 3$ do
  STEP 7. If y_j is not marked visited, set $C := C$ followed by y_j; mark y_j visited
STEP 8. Set $C := C$ followed by x_1.
STEP 9. Output C and stop.

Notice that APPROXHAM is a polynomial algorithm: Both KRUSKAL and EULER are polynomial algorithms and the comparisons in step 6 are few. (See also Exercise 10.) Thus APPROXHAM is efficient and guarantees a spanning cycle that is at most twice as long as the shortest one. The ideas of this algorithm have been extended so that a spanning cycle within 50 percent of optimal is always produced. The extended algorithm is more complex and uses the so-called Minimum-Weight Matching algorithm alluded to in Section 8.2.

These results may sound weak, but in practice APPROXHAM is considered useful. Considerable experimental work has been done on APPROXHAM, and on average it appears to produce a cycle fairly close to the shortest. However, there are no theoretical results that establish the quality of its average-case behavior. Another useful approximation algorithm is the greedy algorithm: At each point visit the nearest unvisited neighbor. It also seems to work well on average, but in the worst case it is known only to produce a cycle C for which $w(C) \leq (\lceil 1 + \log(V) \rceil / 2) w(H(G))$ if G is a graph with V vertices (see also Exercise 6.)

EXERCISES FOR SECTION 4

1. Suppose that you are required to drill holes at (0,0), (1,1), (3,0), (2,2), (1,2), (3,3), (1,3), and (0,2). Assuming that your drill starts and finishes at the origin, find an optimum drilling schedule.
2. For each of the weighted graphs shown, find a minimum-weight Hamiltonian cycle.



(a) (b) (c)

3. Construct an algorithm that will, given a weighted complete graph, use PERM (see Section 3.4) to generate all possible Hamiltonian cycles and thus find a minimum-weight one. Run your algorithm on the graph shown in part (a) of the preceding problem. Discuss the complexity of your algorithm.
4. Two Hamiltonian cycles on the n-clique K_n are considered the same if one is a cyclic rotation of the other or if one is the reverse of the other. Explain why K_n contains $(n-1)!/2$ different Hamiltonian cycles.

5. With the same understanding of when two Hamiltonian cycles are different as in the previous problem, determine the number of different Hamiltonian cycles on the complete bipartite graph $K_{n,n}$.
6. Find a set of points in the plane for which the greedy algorithm (see the last paragraph of this section and Section 5.5) does not produce a minimum-weight Hamiltonian cycle. Here the weight of an edge is the distance between the corresponding vertices.
7. Find a weighted graph G such that $w(C) > 2w(H)$, where C is the Hamiltonian cycle produced by APPROXHAM and H is a minimum-weight Hamiltonian cycle.
8. Suppose that G is a weighted graph that represents cities serviced by an airline and where the weight of an edge (x, y) represents the cost of flying from x to y. Explain why the triangle inequality might not hold for this graph.
9. Use APPROXHAM to find Hamiltonian cycles in the graphs whose underlying point sets are
   (a) $\{(0,0),(2,0),(4,0),(3,1),(1,3)\}$
   (b) $\{(0,0),(3,0),(2,1),(1,2),(0,3)\}$
   (c) $\{(0,0),(5,1),(6,3),(4,4),(2,7),(1,8),(3,4)\}$
10. Find an integer d such that APPROXHAM uses $O(V^d)$ comparisons when finding a Hamiltonian cycle on a set of V points in the plane.
11. Find an example of a weighted graph that has two different minimum-weight spanning trees such that starting with these two APPROXHAM produces different-weight Hamiltonian cycles.
12. Here is an alternative explanation for the second major step of APPROXHAM. Imagine a tree T drawn with a circle C surrounding it. Then imagine that C is a balloon, which when popped collapses in and surrounds T tightly. Explain how a clockwise traversal of the collapsed cycle creates a cycle D on T that traverses every edge of T twice, once in each direction.
13. Turn the idea of the preceding exercise into a precise algorithm suitable, in particular, for use within APPROXHAM.
14. Here is an algorithm called NEARINSERT that finds a short Hamiltonian cycle in a weighted complete graph G. Let $C_1 = \langle r \rangle$ an arbitrary vertex. In general, if C_j is a j-cycle, let v be some vertex in $V(G) - V(C_j)$ that is as close as possible to some vertex in C_j, say u. Create C_{j+1} by inserting v into C_j immediately following u. Run NEARINSERT on the graphs of Exercise 9.
15. Create an algorithm FARINSERT that parallels the algorithm from the preceding exercise except that the vertex to be inserted is as far as possible from the already created cycle. Run FARINSERT on the graphs of Exercise 9.
16. Suppose that you are given n points in the plane that are contained in a $\sqrt{n}$ by $\sqrt{n}$ square. It has been conjectured that on average the minimum-weight

spanning cycle has length at most $4\sqrt{n}$. Discuss why this is a plausible conjecture and investigate the examples of this section to see if they support this conjecture.

17. Construct an algorithm that finds a Hamiltonian cycle in a weighted complete graph K_n by first greedily discarding as many heavy edges as possible subject to the condition that the degree of each incident vertex exceeds $n/2$, and then uses HAMCYCLE from the previous section to find a Hamiltonian cycle. Run your algorithm on the graphs from Exercise 9.

8:5 GRAPH COLORING AND AN APPLICATION TO STORAGE ALLOCATION

The algorithms of this book have been presented following the format of the Pascal programming language. Pascal is a "high level" language whose statements can be translated by a compiler into machine language statements. These are the instructions that the central processing unit (cpu) executes. To execute the program the computer must store the machine language instructions plus the values of all variables used in the program. While the program occupies a constant block of memory, there is choice in the storage of variables. Efficient use and reuse of memory locations can save significantly on the total amount of memory needed.

Example 5.1. Consider an algorithm that, among other things, calculates the volume of a rectangular box.

Algorithm VOLUME

STEP 1. Input Length, Width, Height
. . .
STEP *i*. Area := Length * Width
. . .
STEP *j*. Volume := Area * Height
. . .

The values of the variables Length, Width, Height, Area, and Volume could be stored in, say, memory locations 0 through 4, respectively. In a more efficient storage allocation scheme the variable Volume could be assigned to either memory location 0 or 1, provided that there is no use for the variables Length or Width after step *j*.

The compiler assigns variables to memory locations. Two different variables can be assigned to the same memory location, provided that both variables are never needed in the program at the same time. Thus the mapping of the variables

to memory locations need not be a one-to-one function, but when appropriate two or more variables can double up in one location.

Definition. Two variables in a program (or algorithm) are said to be **noninterfering** if, regardless of the input values, at no instant during the execution of the program are both variables needed at that instant or needed in memory for the execution of some subsequent step. Otherwise, the two variables are said to be **interfering**.

Example 5.1 (continued). The variables Volume and Length are noninterfering variables, as are Volume and Width. Length and Width are clearly interfering variables. So are Length and Height, since the value of Height must be kept in memory for use in step *j*.

Example 5.2. Suppose that the Post Office offers a choice of rates to magazine publishers. The cost of mailing is a function of either the product of the area of the mailing envelope and its weight, or twice the volume of the envelope.

Algorithm POSTAGE

STEP 1.	Input Length, Width, Weight
STEP 2.	Area := Length * Width
STEP 3.	Cost1 := Area * Weight
...	{Some steps that use only Cost1.}
STEP 6.	Input Height
STEP 7.	Volume := Area * Height
STEP 8.	Weight := 2
STEP 9.	Cost2 := Volume * Weight
...	{Some steps that use only Cost2.}
STEP 12.	Print Cost1, Cost2, Volume

Notice that Height and Weight are noninterfering variables, since Height is only used in steps 6 and 7 whereas Weight is needed in steps 1, 2, 3, 8, and 9. However, Cost1 and Height are interfering variables, since the value of Cost1 must be maintained from step 3 to step 12.

Question 5.1. For the variables Length, Width, Height, Weight, Area, Volume, Cost1, and Cost2 from the preceding example, determine which pairs are noninterfering and which interfering. Find a set of four variables, every pair of which is interfering. Are there other sets of four mutually interfering variables? Are there any sets of five variables with every pair interfering?

Ideally, how should the compiler assign variables to memory locations? We assume that all values of a variable should be assigned to just one location during

the execution of the program. Next the storage allocation scheme used by the compiler should use as few memory locations as possible. In addition, it should determine the allocation of memory quickly. There is a graph naturally associated with this problem, and we shall see that storage allocation can be accomplished by "coloring" the vertices of this graph. Furthermore, once the corresponding graph is created, allocation can be determined quickly if there is a good algorithm to do the related graph coloring.

The corresponding graph is called the **interference graph**. Specifically, this graph has a vertex for each variable in the program, and two vertices are joined by an edge if they are interfering.

Example 5.2 (continued). Figure 8.26 exhibits the interference graph where the vertices are labeled with abbreviations of the names of the corresponding variables.

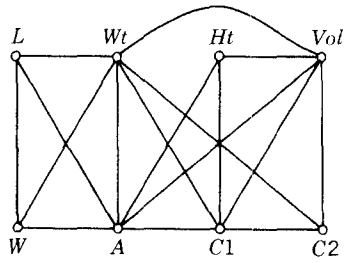


Figure 8.26

Question 5.2. Construct the interference graph for the program from Example 5.1.

Before we proceed with graph coloring, it is appropriate to interject more realism in our storage allocation model. In fact, it is a complex task for a compiler (or for a human) to decide whether or not two variables are interfering. This would involve the potentially infinite task of checking whether or not two variables interfere during the program given any possible input. However, it is possible for a compiler to determine quickly that certain pairs of variables are always noninterfering. These are declared to be noninterfering, and then to be on the safe side all other pairs are declared to be interfering. Thus the "interference graph" derived and used by the compiler may have more edges than the true interference graph. In the simple examples in this section (both algorithms without loops or branching), it is straightforward to determine the real interference graph.

We shift now to graph "colorings." Graph colorings began with the Four-Color Problem, a "puzzle" that asks whether every map can have its countries colored with one of four colors so that no two countries with a border in common receive the same color. This problem intrigued mathematicians for more than a century until it was solved in 1976. (The answer is yes, every map can be 4-colored. The proof due to Kenneth Appel and Wolfgang Haken is long and intricate and includes a computer check of over 1400 cases.)

Definition. For k a positive integer, a graph G is said to be **k-colored** (or **k-colorable**) if each vertex is (or can be) assigned one of k colors so that no two adjacent vertices receive the same color. The **chromatic number of G**, denoted by $\chi(G)$, is the minimum number k such that G can be k-colored. The **clique number of a graph G**, denoted $\mathbf{cl}(G)$, is the largest number r such that G contains an r-clique as a subgraph.

We have seen 2-colorability before in Section 5.2: There a 2-colorable graph was called bipartite.

Example 5.3. The graphs G and H in Figure 8.27 are colored with A, B, C, and D. The graph G can be 3-colored (replace the D with a B) as well as 2-colored (in addition replace C by an A); however, H cannot be 2-colored because it contains three mutually adjacent vertices. The clique numbers of these graphs are 2 and 3, respectively.

G H

Figure 8.27

Storage allocation problems can be translated into problems about colorings of the interference graph. Associate a different color with each memory location. Then the interference graph receives a coloring by assigning a vertex, labeled by a variable, the color of the memory location to which the variable is assigned. Conversely, a graph coloring (with the same colors as associated with memory locations) prescribes a memory allocation for variables. In both these assignments the graph is correctly k-colored if and only if an assignment of k memory locations is made in which pairs of variables that are interfering are assigned to different memory locations.

Here are some central problems in the theory of graph colorings.

Problem A. Construct an algorithm that, given a graph G and an integer k, determines if G can be k-colored, and if so finds a k-coloring.

Problem B. Construct an algorithm that, given a graph, determines its chromatic number and finds a corresponding coloring.

Problem C. Does the chromatic number of a graph equal its clique number?

These problems translate to important ones for storage allocation. Given a program, suppose that the compiler has declared certain pairs of variables to be noninterfering and all others to be interfering. In this context, Problem A asks for an algorithm to determine if k memory locations are sufficient, and if so, to find such an allocation. Problem B asks for an algorithm to determine the minimum number of storage locations needed, and Problem C asks whether the answer to Problem B is the same as the maximum number of mutually interfering variables. (Look back at Question 5.1.)

Question 5.3. The "interference graph" G created by the compiler may have more edges than the true interferences graph H. Explain why $\chi(G) \geq \chi(H)$. If the compiler assigns $k = \chi(G)$ memory locations to a program based on a coloring of G, explain why two "truly" interfering variables will not be assigned the same memory location.

Question 5.4. Determine the chromatic number and the clique number of each graph shown in Figure 8.28.

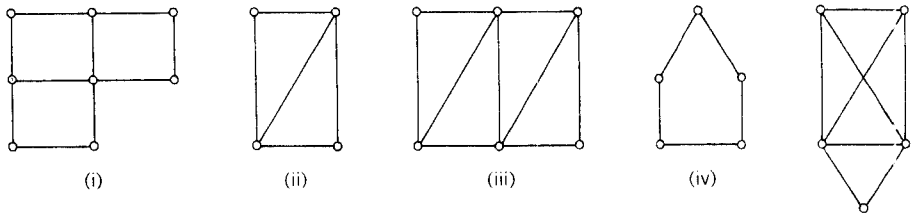


Figure 8.28

Question 5.5. Solve Problem A for $k = 1$.

Problem A for $k = 2$ is not difficult. It is a frequent circumstance that good algorithms follow from insightful theorems. Theorem 5.1, a characterization of 2-colorable graphs, is just such an instance. It should be clear from Example 5.3 and Question 5.4 that a graph containing a cycle with an odd number of vertices cannot be 2-colored. (See Exercise 4.) Thus a 2-colorable graph cannot contain an odd cycle. Surprisingly, this is the only condition needed to ensure that a graph is 2-colorable.

Theorem 5.1. A graph G is 2-colorable if and only if G contains no odd cycle.

Proof. We prove that if G contains no odd cycle, it is 2-colorable. Select any vertex r to be the root and color it red. For each vertex x in G if the distance from r to x is even, color x red. Otherwise, color x blue. If G is not connected, pick a root in each component and repeat. This procedure places 2 colors on the vertices of G, but does each edge join vertices of different colors?

Example 5.4. In the two graphs of Figure 8.29 each vertex, besides the root r, is labeled with its distance from r and the color it receives in the procedure described above.

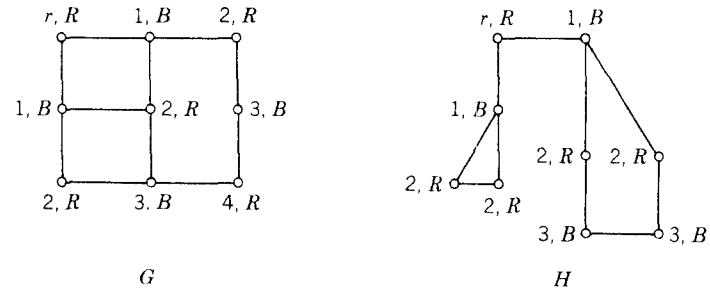


Figure 8.29

To return to the proof, suppose that two vertices, say x and y, with the same color are joined by an edge. If P_x denotes a shortest path from r to x and P_y denotes a shortest path from r to y, then $|P_x|$ and $|P_y|$ are either both even or both odd. In either case their sum is even. The cycle consisting of P_x followed by the edge (x, y) followed by P_y (in reverse order) is an odd cycle unless P_x and P_y have edges in common. In this case these paths are easily seen to contain an odd cycle. This contradiction forces us to conclude that there is no such edge joining two vertices of the same color, that is, every edge joins two vertices that received different colors and so the graph is 2-colored. □

Since the distance from any vertex to the root can be determined by a breadth-first search as described in Section 8.1, this proof suggests an efficient algorithm using BFS to test for 2-colorability (see Exercises 8 and 9).

It would be reasonable to think that there should be an analogue of Theorem 5.1 and the accompanying algorithm for 3-colorable graphs. In fact, there is no known characterization of 3-chromatic graphs or any efficient 3-coloring algorithm. Furthermore, it is unlikely that any will be discovered.

In the remainder of this section we present two algorithms that attempt to k-color a graph, where k is any positive integer. The idea of the first algorithm, known as **Sequential Coloring**, is to dive in and start coloring using as few colors as possible. We consider the vertices from 1 up to V in order and to each vertex v we assign the first available color not already assigned to a neighbor of v.

Algorithm SEQUENTIALCOLOR

STEP 1. Input G with V vertices $x_1, \ldots, x_V$
{Let the potential colors be $1, 2, \ldots, V$.}

STEP 2. For $I = 1$ to V do
  STEP 3. Create $L_I = \langle 1, 2, \ldots, I \rangle$ {L_I is the list of colors that might get assigned to x_I.}
STEP 4. For $I = 1$ to V do
  Begin
  STEP 5. Set $c_I :=$ first color in L_I {c_I is the color assigned to x_I}
  STEP 6. For each J with $I < J$ and (x_I, x_J) in $E(G)$ do
    STEP 7. Set $L_J := L_J - c_I$ {x_J cannot receive the same color as x_I.}
  End
STEP 8. Output each vertex, the color it received, and the total number of colors used; then stop.

Example 5.5. Table 8.6 is a trace of SEQUENTIALCOLOR applied to the graph in Figure 8.30. With the given ordering of the vertices SEQUENTIALCOLOR produces the coloring as shown.

Figure 8.30

Table 8.6

Step No.	I	L_I	c_I	J	L_J
3	1	⟨1⟩			
3	2	⟨1, 2⟩			
3	3	⟨1, 2, 3⟩			
3	4	⟨1, 2, 3, 4⟩			
5	1		1		
7	1			4	⟨2, 3, 4⟩
5	2		1		
7	2			3	⟨2, 3⟩
5	3		2		
7	3			4	⟨3, 4⟩
5	4		3		

Question 5.6. Run SEQUENTIALCOLOR on the labeled graphs shown in Figure 8.31.

SEQUENTIALCOLOR clearly places different colors on adjacent vertices; however, sometimes it uses more than the minimum number of colors. This point is important—we cannot be sure that the results of SEQUENTIALCOLOR are

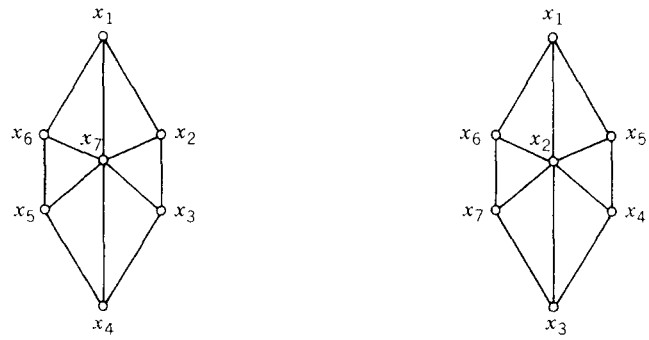


Figure 8.31

the best possible. But does it use just one extra color or just a "few" extra colors (whatever that means)? The answer is no; this algorithm might use far too many colors. Precisely, for every positive integer m there are graphs of chromatic number k on which SequentialColor uses $k + m$ colors. Exercise 29 illustrates this phenomena on 2-chromatic graphs. Thus looking back at the basic coloring problems, this algorithm might or might not solve Problem A (can a graph be k-colored?).

How efficient is SEQUENTIALCOLOR? Here the news is good. The loop in step 2 requires no more than V^2 assignments. The loop at step 4 has two purposes. The coloring (step 5) requires one assignment for each vertex. Updating the lists of possible colors requires one comparison and one assignment for each edge in the graph. Since $E \leq V^2$, SEQUENTIALCOLOR is an $O(V^2)$ algorithm (counting either comparisons or assignments or both.)

However, SEQUENTIALCOLOR can be used to find the chromatic number of a graph and hence to answer Problem B. As you saw in the preceding question, the number of colors used by SEQUENTIALCOLOR depends on the numbering of the vertices not on the graph alone. (The chromatic number depends only on the graph.) If we try every possible numbering of the vertices and with each numbering run SEQUENTIALCOLOR, then the chromatic number of the graph will be the minimum number of colors used in all these runs. This is the content of the next theorem.

Theorem 5.2. Let $\chi(G)$ be the chromatic number of a graph G. Then there is a labeling of the vertices of G with $x_1, x_2, \ldots, x_V$ such that SEQUENTIALCOLOR run on G with this labeling uses $\chi(G)$ colors.

Proof. Suppose that G is colored with colors $1, \ldots, R$. Label the vertices of G so that if $I < J$, then every vertex colored I receives a label before any vertex colored J. One way to accomplish this is to label all the vertices colored 1 (in any order you like), followed by all the vertices colored 2 (in any order you like), continuing until all vertices are labeled. If SEQUENTIALCOLOR is applied to G

with this labeling, then all the vertices originally colored 1 get colored 1. The vertices originally colored 2 might be colored with either 1 or 2. In general, the vertices that were originally colored I will be colored with one of $1, \ldots, I$. Thus no more than R colors are used by SEQUENTIALCOLOR. When $R = \chi(G)$, exactly R colors will be used, by definition of $\chi(G)$. □

The consequence of Theorem 5.2 is that SEQUENTIALCOLOR will find the chromatic number of a graph if all possible orderings of the vertices are tried. There are $V!$ such numberings and so the number of steps of this approach is more than $V!$, making this a slow algorithm. However, we next consider a modification of this approach known as **Backtracking**. This will be more efficient, but how much more we don't divulge yet.

Suppose that we want to know whether a given graph is 3-colorable. The idea of Backtracking is to try systematically all possible 3-colorings of the graph. The algorithm begins by trying to 3-color the graph using SEQUENTIALCOLOR, but if a fourth color is needed, it discontinues this approach. The algorithm backtracks (or backs up) to the last vertex, where there was choice in the coloring, and makes a different choice.

Example 5.6. Figure 8.32 shows a graph that can be 3-colored, but with the given vertex numbering SEQUENTIALCOLOR will 4-color it. A partial 4-coloring is given in Figure 8.32.

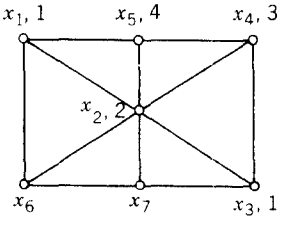


Figure 8.32

Once a fourth color is needed on x_5, we backtrack to x_4. There is no choice for its color if we are to 3-color the graph. However, if we backtrack to x_3, it can be recolored with 3. Then colors 1 and 3 are available for x_4 and x_5, respectively. Furthermore, this partial coloring extends to a 3-coloring of the whole graph by placing colors 3 and 1 on x_6 and x_7, respectively.

In this process of backtracking either a 3-coloring is discovered or all possible 3-colorings fail.

Here are some details of the algorithm BACKTRACKCOLOR. The number of colors that may be used is denoted by N. The possible colors are denoted $1, 2, \ldots, N$. The vertices are denoted by $x_1, x_2, \ldots, x_V$, and c_I contains the color that is used on the vertex x_I. Without loss of generality we may assume that

vertex x_1 gets color 1; initially, $c_I := 0$ for $I = 2, \ldots, V$ to denote that these vertices are not colored. The algorithm parallels SEQUENTIALCOLOR: The vertices are colored in increasing numerical order, but if it ever is the case that the coloring cannot be extended to other vertices, the algorithm moves back one vertex and tries to recolor it with the next larger available color. The algorithm repeats this backtracking until either a vertex is found that can receive another color or until it finds that no vertex can be recolored. In the former case the algorithm begins again, trying to extend the partial coloring; in the latter case essentially all colorings fail, that is, the graph cannot be colored with N colors. The variable FORWARD keeps track of when the coloring is being extended (and FORWARD is TRUE) or when backtracking is occurring (and FORWARD is FALSE.)

Algorithm BACKTRACKCOLOR.

STEP 1. Initialize
Input G with V vertices and a positive integer N {The algorithm will determine if G can be N-colored.}
Set $c_1 := 1$ {x_1 gets color 1.}
Set $c_i := 0$ for $i = 2, \ldots, V$
Set $J := 2$ {J will index the vertex currently being colored.}
Set FORWARD := TRUE
STEP 2. While $J < V + 1$ do
STEP 3. If FORWARD = TRUE then
Call Procedure COLOR
Else {FORWARD = FALSE}
Begin
STEP 4. $c_J := 0$
STEP 5. $J := J - 1$
STEP 6. If $J \neq 1$, then
Call Procedure COLOR
Else {$J = 1$}
Output "There is no N-coloring of G" and stop.
end {Step 3}
STEP 7. Output the coloring and stop.

Procedure COLOR

STEP 1. Find smallest $K > c_J$ such that if $I < J$ and (x_I, x_J) is in $E(G)$, then $c_I \neq K$
STEP 2. If $K \leq N$, then
Begin
STEP 3. Set $c_J := K$
STEP 4. Set $J := J + 1$

STEP 5. Set FORWARD := TRUE
End
Else {$K > N$}
STEP 6. Set FORWARD := FALSE
End.

Note the similarity between BACKTRACKCOLOR and the algorithm DFS-HAMCYCLE of Section 8.3.

Example 5.6 (continued). Table 8.7 is a trace of BACKTRACKCOLOR applied to the graph in Figure 8.32 with $N = 3$.

Table 8.7

Step No.		*J*	*K*	*FORWARD*	*c*[1,2,3,4,5,6,7]
Main	1	2		TRUE	[1,0,0,0,0,0,0]
COLOR	1	2	2		
	3–5	3	2		[1,2,0,0,0,0,0]
COLOR	1	3	1		
	3–5	4		TRUE	[1,2,1,0,0,0,0]
COLOR	1	4	3		
	3–5	5		TRUE	[1,2,1,3,0,0,0]
COLOR	1	5	4		
	6	5		FALSE	
Main	4–5	4			[1,2,1,3,0,0,0]
COLOR	1	4	4		
	6	4		FALSE	
Main	4–5	3			[1,2,1,0,0,0,0]
COLOR	1	3	3		
	3–5	4		TRUE	[1,2,3,0,0,0,0]
COLOR	1	4	1		
	3–5	5		TRUE	[1,2,3,1,0,0,0]
COLOR	1	5	3		
	3–5	6		TRUE	[1,2,3,1,3,0,0]
COLOR	1	6	3		
	3–5	7		TRUE	[1,2,3,1,3,3,0]
COLOR	1	7	1		
	3–5	8		TRUE	[1,2,3,1,3,3,1]
Main	7	Stop.			

Question 5.7. Trace BACKTRACKCOLOR applied to the complete graph K_4 with $N = 3$.

That BACKTRACKCOLOR is not a good algorithm can be seen by imagining an attempt to $(n-1)$-color K_n for any $n > 1$. To begin with, the colors

$1, 2, \ldots, (n-1)$ are assigned to $x_1, \ldots, x_{n-1}$; however, this coloring cannot be completed. Backtracking to x_{n-2}, we find that it can be recolored with $n-1$ and x_{n-1} with $n-2$, but still there is no color left for vertex n. In general, the algorithm will backtrack to x_I with $I > 1$ and try all colors not on vertices $x_1, \ldots, x_{I-1}$. Since these vertices receive $I-1$ distinct colors, there are

$$(n-1) - (I-1) = n - I$$

colors that are tried on x_I for every fixed coloring of the $x_1, \ldots, x_{I-1}$. This means that precisely

$$(n-2)(n-3)\cdots(n-I)\cdots\cdot 2 \cdot 1 = (n-2)!$$

colorings are tried before the algorithm reports that K_n cannot be $(n-1)$-colored. Thus BACKTRACKCOLOR is exponential in the worst case.

Exercises 40 and 41 ask you to verify that BACKTRACKCOLOR is correct. Exercise 43 suggests a recursive version of this coloring scheme.

All known algorithms for deciding whether a graph can be k-colored (for $k > 2$) are exponential. Thus the coloring problem is in the same unresolved state as the Hamiltonian cycle problem, the Traveling Salesrepresentative problem, and the Satisfiability problem. It is also one of the NP-Complete problems (as defined in the end of Section 8.3), which means that there is a polynomial algorithm for one of these four problems if and only if there is a polynomial algorithm for all four of them. However, graph coloring is an important problem with applications to timetable scheduling, routing problems, circuit board testing as well as to storage allocation. Thus coloring algorithms are used in practice. There are ways to seemingly improve the typical running time of these algorithms. For example, vertices of large degree are in some sense the hardest to color, and so numbering the vertices of largest degree with the smallest numbers may lead to a reasonable labeling on which to run sequential coloring. In some applications characteristics of the graphs can be incorporated to increase the efficiency of the algorithm. A variation is considered in Exercise 39.

Curiously, there are no known efficient approximate coloring algorithms, like the approximation algorithm of Section 8.4. Precisely, there is no known algorithm that will color a graph G with $O(\chi(G))$ colors. The best that is known is that there is an algorithm that will color a graph G with $O(\{V/\log(V)\}\chi(G))$ colors, where V is the number of vertices of G. Furthermore, it has been shown that the question of whether a graph is $(c\chi(G))$-colorable for any c less than 2 is also an NP-Complete problem. Thus in the worst case, approximations for graph-coloring algorithms are also very hard.

Look back at Problem C of our fundamental graph-coloring problems: this problem asked whether the chromatic number of a graph equals the size of the largest clique in it. Immediately, we saw that the chromatic number could be greater than the clique number. The clique number of a graph gives a lower bound

on the chromatic number and so if an algorithm, like SEQUENTIALCOLOR, ever achieves a coloring with $\mathrm{cl}(G)$ colors, then the numbers of colors used is the minimum possible. However, determining the clique number of a graph is also one of the NP-Complete problems. A graph G is called **perfect** if $\chi(H) = \mathrm{cl}(H)$ for every H that is an induced subgraph of G. It has recently been shown that there is a polynomial algorithm to determine whether a perfect graph can be k-colored; this algorithm uses the recent Ellipsoid Method of Linear Programming. On the other hand, there is no known polynomial algorithm to determine whether an arbitrary graph is perfect.

Despite all this bad news from the standpoint of efficient algorithms, the storage allocation problem, with which we began this section, is one that must be confronted by compilers and their designers. In the past, coloring algorithms have been used and modified for special needs, and in these cases they were effective in solving storage allocation problems. Recently, progress has been made by Jeanne Ferrante who has used a process known as "renaming" to get a better algorithm that runs in polynomial time and is applicable in all cases. In renaming, a new variable is created and assumes the value of an old variable at some points in the program. This increases the number of variables (and the number of assignment statements), but the new number of variables remains a polynomial in the number of original variables; the necessary program rewriting can be accomplished in one pass through the program. With this change there is a good algorithm for storage allocation that assigns the minimum possible number of memory locations, which is the maximum number of variables with every pair interfering at some point in the program. The latter number is called **Maxlive**. In graph theory terms, some vertices of the interference graph are split so that the total number of vertices is a polynomial of the original V, and then there is a good algorithm that colors the new graph G' in Maxlive colors. Since Maxlive $= \mathrm{cl}(G) \leq \chi(G)$ for every interference graph G, we see that G' has been colored with the minimum number of colors. (See also Exercise 2.)

If, in addition, a compiler can reorder the statements of a program, then it may be possible to reduce the value of Maxlive and hence the amount of memory needed.

Example 5.7. The following reordering reduces Maxlive from 2 to 1.

Program One: Maxlive = 2	Program Two: Maxlive = 1
1. Define A	1. Define A
2. Define B	2. Use A
3. Use A	3. Define B
4. Use B	4. Use B

However, it has been shown that determining the order of a program for which Maxlive is a minimum is an NP-Complete problem! No matter where you look, there are hard unsolved algorithmic problems.

EXERCISES FOR SECTION 5

1. Here is an algorithm that interchanges the values of two variables (from Chapter 2.)

   STEP 1. $x\text{old} := x$
   STEP 2. $x := y$
   STEP 3. $y := x\text{old}$

   Construct the interference graph for this algorithm, find a coloring of the graph with the minimum possible number of colors, and find a storage allocation scheme using this many memory locations.

2. Construct the interference graph G of the following algorithm

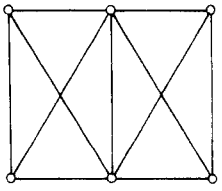
   STEP 1. Input A, B
   STEP 2. $A := A * B$
   STEP 3. $C := A^2$
   STEP 4. $D := C^2$
   STEP 5. $E := D^2$
   STEP 6. $B := 3$
   STEP 7. $E := E * B$.

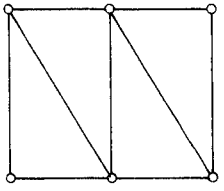
   Determine $\chi(G)$ and $\text{cl}(G)$. Find a way to "rename" at least one variable so that if G' is the resulting interference graph and $\chi(G')$ memory locations are used, then $\chi(G') = \text{cl}(G')$.

3. Here is one possible algorithm for storage allocation. Assign each variable to a different memory location. Thus, if a program has n variables, say $V_1, V_2, \ldots, V_n$, for $i = 1, 2, \ldots, n$, assign V_i to memory M_{i-1}. Explain why this allocation is quick and avoids all conflicts. What is the disadvantage of this algorithm?

4. Let C_j be a cycle with j vertices. Find $\chi(C_j)$ and $\text{cl}(C_j)$ for all j.

5. Find a 4-coloring of the following graphs if possible. For each graph determine whether its chromatic number is four or not.

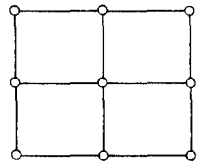


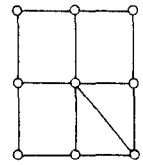


6. Explain why $\chi(G) \geq \text{cl}(G)$ for every graph G. Describe an infinite set of graphs for which $\chi(G) = \text{cl}(G)$ and an infinite set for which $\chi(G) > \text{cl}(G)$.

7. Explain why a graph with V vertices can always be V-colored. Characterize those graphs on V vertices for which $\chi(G) = V$. If G is a graph with V vertices and $\chi(G) = V - 1$, what can you say about $\text{cl}(G)$?

8. Write out the details of a BFS algorithm that tries to 2-color a graph. Run your algorithm on the following graphs.





9. Show that the complexity of the algorithm from the preceding problem is $O(V^2)$.

10. Prove that if G is a graph with maximum degree d, then $\chi(G) \leq d + 1$.

11. Prove that if G is a connected graph with maximum degree d ($d > 2$), then $\chi(G) \leq d$ unless $G = K_{d+1}$.

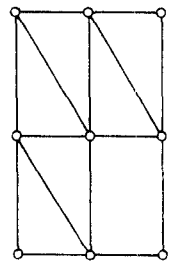
12. For $k = 1, 2, 3$, and 4 find examples of graphs with maximum degree d such that $\chi(G) = d - k$.

13. Suppose that $\alpha(G)$ denotes the largest set of vertices of a graph, no two of which are adjacent. (See also Chapter 5, Supplementary Exercises 20 to 25.) Then prove that if G is any graph with V vertices, then $\chi(G) \geq V/\alpha(G)$.

14. If G is a graph with V vertices and maximum degree d, prove that $\chi(G) \geq V/(V - d)$.

15. If G^c is the complement of the graph G (as defined in Chapter 5, Supplementary Exercise 1), prove that $\chi(G) + \chi(G^c) \leq V + 1$, where V is the number of vertices of G.

16. Color the following map with three colors so that no two regions with a border in common receive the same color. Can fewer colors be used?



Find a map that can be 2-colored and one that can be 4-colored but not 3-colored.

17. (*a*) Draw a map. From this construct a graph as follows: Create a vertex for every region of the map and join two vertices by an edge if the corresponding regions have a border in common. Show that the graph can be drawn so that no two edges cross. The graph is called the **dual** of the original map.

(*b*) Next pick a graph G drawn so that no two edges cross. Find a map such that the dual of the map is the graph G.

18. In general, let M be a map and $G(M)$ the dual graph derived from it (as defined in Exercise 17.) Explain why $\chi(G(M))$ is precisely the minimum number of colors needed to color every region of M so that no two regions with a border in common receive the same color.

19. A graph is called **planar** if it can be drawn in the plane so that no two edges cross. Prove that if G is a connected, planar graph, drawn with F faces (including the outside face) then $V - E + F = 2$. This result is known as **Euler's formula**. (*Hint:* First prove this for trees. Then use induction on the number of edges of the graph.)

20. Prove that neither K_5 nor $K_{3,3}$ is a planar graph. (*Hint:* Use Euler's formula.)

21. Prove that for a planar graph the average degree, $2E/V$, is less than 6. Use this to conclude that every planar graph contains a vertex of degree 5 or less.

22. Prove that every planar graph can be 6-colored.

23. Prove that every planar graph can be 5-colored.

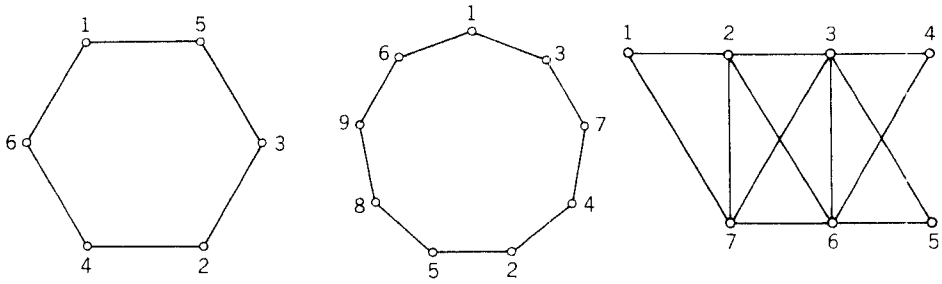
24. There once was a farmer with a large (square) tract of farm land. The farmer had five children and decided to divide the land into five pieces, one for each child, but to facilitate communication she wanted each piece to have a border in common with all other four pieces. Is such a division possible? If so, give an example of such a division (the pieces don't need to be the same size.) If not, find a division with each piece having at least a corner (or vertex) in common with every other piece.

25. Let G be a connected planar graph, drawn in the plane. Prove that G is Eulerian if and only if the resulting map can have its regions 2-colored. (Or in the notation of Exercises 17 and 18, if G is drawn in the plane and M is the resulting map, then G is Eulerian if and only if $G(M)$ is bipartite.)

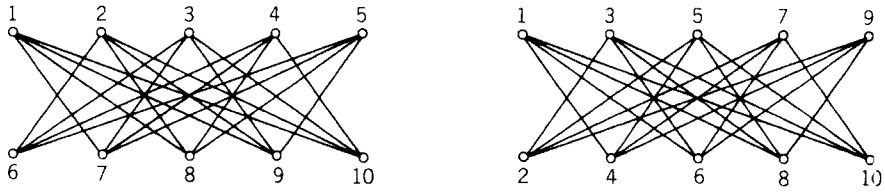
26. Prove that if G is a planar Eulerian graph such that every region of the graph in the plane has exactly three sides, then G can be 3-colored.

27. Design an algorithm that upon input of a graph will find a cycle in the graph with an odd number of vertices or else report that there is none. Then design an algorithm to search for cycles of even length. Compare the complexities of your algorithms.

28. Run SEQUENTIALCOLOR on each of the following labeled graphs. Does the algorithm use the minimum possible number of colors?

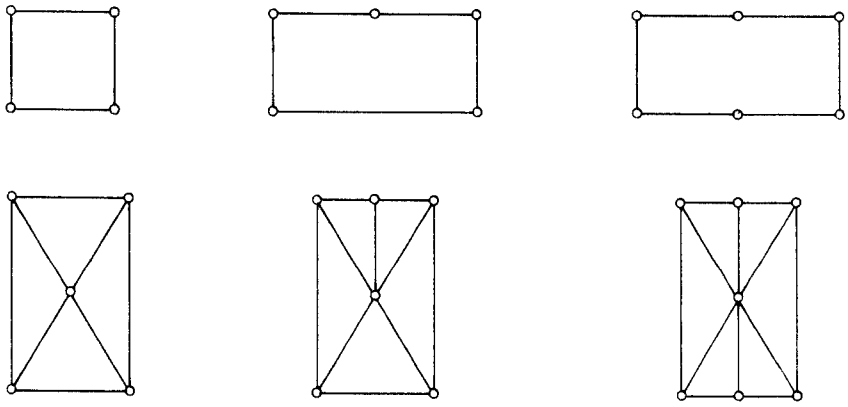


29. Run SEQUENTIALCOLOR on the following graphs.

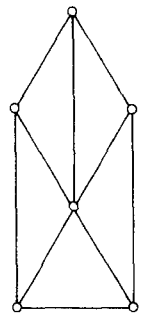


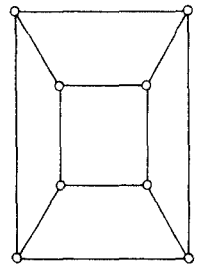
30. Generalize the examples of the preceding problem to show that for every k there are 2-chromatic graphs that when suitably labeled cause SEQUENTIALCOLOR to use $2 + k$ colors on them.

31. Here are some graphs; find the chromatic number of each graph and then find a labeling of them so that SEQUENTIALCOLOR uses that many colors on them.



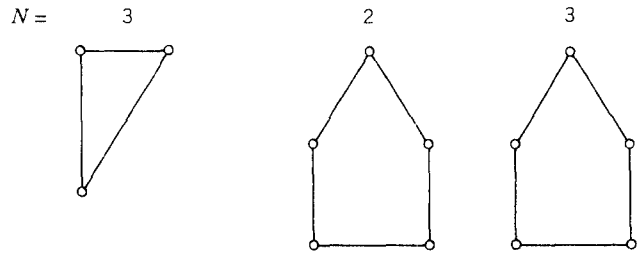
32. Determine the chromatic number and the clique number of the following graphs.





33. Find an example of a 4-colorable graph and a labeling so that SEQUENTIALCOLOR uses more than four colors on it.

34. Run BACKTRACKCOLOR with the indicated values of N on the following graphs.



35. Redesign BACKTRACKCOLOR so that it finds all colorings of a graph with N colors.

36. Design an algorithm to determine cl(G) and determine the complexity of the algorithm.

37. Here is a variation on BACKTRACKCOLOR: First use DEPTHFIRSTSEARCH to visit and label all vertices in the order they are visited. Then run BACKTRACKCOLOR. Find an example of a graph on which the resulting algorithm is more efficient than BACKTRACKCOLOR and one on which they do exactly the same work. In general, on what graphs will this revised algorithm be more efficient and on what graphs equally efficient? Is it ever less efficient?

38. Comment on whether the algorithm SEQUENTIALCOLOR is a greedy algorithm.

39. Design an algorithm that first numbers the vertices of a graph by decreasing degrees (i.e., the vertices of highest degree are numbered first) and then run SEQUENTIALCOLOR. This is known as Largestfirst. Find sets of graphs on which this version is more efficient and sets on which this is no more efficient.

40. Prove that if BACKTRACKCOLOR outputs a coloring, then it has found an N-coloring of G.

41. Prove that if BACKTRACKCOLOR reports "No N-coloring," there is no N-coloring of G. (*Hint:* Proceed by contradiction. Assume that G has an N-coloring and show that BACKTRACKCOLOR must find it.)

42. Find a function $g(V)$ such that BACKTRACKCOLOR requires at most $O(g(V))$ comparisons when run on a graph with V vertices.

43. Write a recursive version of backtrack coloring. (*Hint:* Just as BACKTRACKCOLOR resembles the algorithm DFS-HAMCYCLE, so should the recursive implementations.)