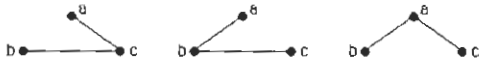
SECTION 10.4 Spanning Trees

2. Since the edge $\{a, b\}$ is part of a simple circuit, we can remove it. Then since the edge $\{b, c\}$ is part of a simple circuit that still remains, we can remove it. At this point there are no more simple circuits, so we have a spanning tree. There are many other possible answers, corresponding to different choices of edges to remove.

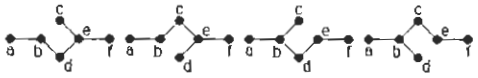
4. We can remove these edges to produce a spanning tree (see comments for Exercise 2): $\{a, i\}$, $\{b, i\}$, $\{b, j\}$, $\{c, d\}$, $\{c, j\}$, $\{d, e\}$, $\{e, j\}$, $\{f, i\}$, $\{f, j\}$, and $\{g, i\}$.

6. There are many, many possible answers. One set of choices is to remove edges $\{a, e\}$, $\{a, h\}$, $\{b, g\}$, $\{c, f\}$, $\{c, j\}$, $\{d, k\}$, $\{e, i\}$, $\{g, l\}$, $\{h, l\}$, and $\{i, k\}$.

8. We can remove any one of the three edges to produce a spanning tree. The trees are therefore the ones shown below.



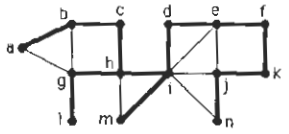
10. We can remove any one of the four edges in the middle square to produce a spanning tree, as shown.



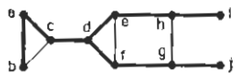
12. This is really the same problem as Exercises 11a, 12a, and 13a in Section 10.1, since a spanning tree of K_n is just a tree with n vertices. The answers are restated here for convenience.

a) 1 **b)** 2 **c)** 3

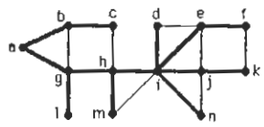
14. The tree is shown in heavy lines. It is produced by starting at a and continuing as far as possible without backtracking, choosing the first unused vertex (in alphabetical order) at each point. When the path reaches vertex l, we need to backtrack. Backtracking to h, we can then form the path all the way to n without further backtracking. Finally we backtrack to vertex i to pick up vertex m.



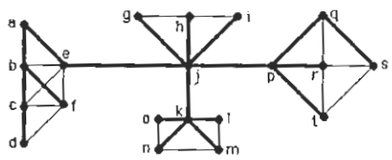
16. If we start at vertex a and use alphabetical order, then the breadth-first search spanning tree is unique. Consider the graph in Exercise 13. We first fan out from vertex a, picking up the edges $\{a, b\}$ and $\{a, c\}$. There are no new vertices from b, so we fan out from c, to get edge $\{c, d\}$. Then we fan out from d to get edges $\{d, e\}$ and $\{d, f\}$. This process continues until we have the entire tree shown in heavy lines below.



The tree for the graph in Exercise 14 is shown in heavy lines. It is produced by the same fanning-out procedure as described above.



The spanning tree for the graph in Exercise 15 is shown in heavy lines.

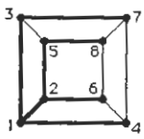


18. a) We start at the vertex in the middle of the wheel and visit all its neighbors—the vertices on the rim. This forms the spanning tree $K_{1,6}$ (see Exercise 19 for the general situation).

b) We start at any vertex and visit all its neighbors. Thus the resulting spanning tree is therefore $K_{1,4}$.

c) See Exercise 21 for the general result. We get a "double star": a $K_{1,3}$ and a $K_{1,2}$ with their centers joined by an edge.

d) By the symmetry of the cube, the result will always be the same (up to isomorphism), regardless of the order we impose on the vertices. We start at a vertex and fan out to its three neighbors. From one of them we fan out to two more, and pick up one more vertex from another neighbor. The final vertex is at a distance 3 from the root. In this figure we have labeled the vertices in the order visited.



20. Since every vertex is connected to every other vertex, the breadth-first search will construct the tree $K_{1,n-1}$, with every vertex adjacent to the starting vertex. The depth-first search will produce a simple path of length $n - 1$ for the same reason.

22. We construct a tree using one of these search methods. We color the first vertex red, and whenever we add a new vertex to the tree, we color it blue if we reach it from a red vertex, and we color it red if we reach it from a blue vertex. When we encounter a vertex that is already in the tree (and therefore will not be added to the tree), we compare its color to that of the vertex we are currently processing. If the colors are the same, then we know immediately that the graph is not bipartite. If we get through the entire process without finding such a clash, then we conclude that the graph is bipartite.

24. If the edge is a cut edge, then it provides the unique simple path between its endpoints. Therefore it must be in every spanning tree for the graph. Conversely, if an edge is not a cut edge, then it can be removed without

disconnecting the graph, and every spanning tree of the resulting graph will be a spanning tree of the original graph not containing this edge. Thus we have shown that an edge of a connected simple graph must be in every spanning tree for this graph if and only if the edge is a cut edge—i.e., its removal disconnects the graph.

26. We can order the vertices of the graph in the order in which they are first encountered in the search processes. Note, however, that we already need an order (at least locally, among the neighbors of a vertex) to make the search processes well-defined. The resulting orders given by depth-first search or breadth-first search are not the same, of course.

28. In each case we will call the colors red, blue, and green. Our backtracking plan is to color the vertices in alphabetical order. We first try the color red for the current vertex, if possible, and then move on to the next vertex. When we have backtracked to this vertex, we then try blue, if possible. Finally we try green. If no coloring of this vertex succeeds, then we erase the color on this vertex and backtrack to the previous vertex. For the graph in Exercise 7, no backtracking is required. We assign red, blue, red, and green to the vertices in alphabetical order. For the graph in Exercise 8, again no backtracking is required. We assign red, blue, blue, green, green, and red to the vertices in alphabetical order. And for the graph in Exercise 9, no backtracking is required either. We assign red, blue, red, blue, and blue to the vertices in alphabetical order.

30. a) The largest number that can possibly be included is 19. Since the sum of 19 and any smaller number in the list is greater than 20, we conclude that no subset with sum 20 contains 19. Then we try 14 and reach the same conclusion. Finally, we try 11, and note that after we have included 8, the list has been exhausted and the sum is not 20. Therefore there is no subset whose sum is 20.

b) Starting with 27 in the set, we soon find that the subset $\{27, 14\}$ has the desired sum of 41.

c) First we try putting 27 into the subset. If we also include 24, then no further additions are possible, so we backtrack and try including 19 with 27. Now it is possible to add 14, giving us the desired sum of 60.

32. a) We begin at the starting position. At each position, we keep track of which moves we have tried, and we try the moves in the order up, down, right, and left. (We also assume that the direction from which we entered this position has been tried, since we do not want our solution to retrace steps.) When we try a move, we then proceed along the chosen route until we are stymied, at which point we backtrack and try the next possible move. Either this will eventually lead us to the exit position, or we will have tried all the possibilities and concluded that there is no solution.

b) We start at position X. Since we cannot go up, we try going down. At the next intersection there is only one choice, so we go left. (All directions are stated in terms of our view of the picture.) This lead us to a dead end. Therefore we backtrack to position X and try going right. This leads us (without choices) to the opening about two thirds of the way from left to right in the second row, where we have the choice of going left or down. We try going down, and then right. No further choices are possible until we reach the opening just above the exit. Here we first try going up, but that leads to a dead end, so we try going down, and that leads us to the exit.

34. There is one tree for each component of the graph.

36. The algorithm is identical to the algorithm for obtaining-spanning trees by deleting edges in simple circuits. While circuits remain, we remove an edge of a simple circuit. This does not disconnect any connected component of the graph, and eventually the process terminates with a forest of spanning trees of the components.

38. We apply breadth-first search, starting from the first vertex. When that search terminates, i.e., when the list is emptied, then we look for the first vertex that has not yet been included in the forest. If no such vertex is found, then we are done. If v is such a vertex, then we begin breadth-first search again from v, constructing the second tree in the forest. We continue in this way until all the vertices have been included.

40. First notice that the order in which vertices are put into (and therefore taken out of) the list L is level-order. In other words, the root of the resulting tree comes first, then the vertices at level 1 (put into the list while processing the root), then the vertices at level 2 (put into the list while processing vertices at level 1), and so on. (A formal proof of this is given in Exercise 43.) Now suppose that uv is an edge not in the tree, and suppose without loss of generality that the algorithm processed u before it processed v. (In other words, u entered the list L before v did.) Since the edge uv is not in the tree, it must be the case that v was already in the list L when u was being processed. In order for this to happen, the parent p of v must have already been processed before u. Note that p's level in the tree is one less than v's level. Therefore u's level is greater than or equal to p's level but less than or equal to v's level, and the proof is complete.

42. Assume that the connected simple graph G does not have a simple path of length at least k. Consider the longest path in the depth-first search tree. Since each edge connects an ancestor and a descendant, we can bound the number of edges by counting the total number of ancestors of each descendant. But if the longest path is shorter than k, then each descendant has at most $k-1$ ancestors. Therefore there can be at most $(k-1)n$ edges.

44. We modify the pseudocode given in Algorithm 1 by initializing a global variable m to be 0 at the beginning of the algorithm, and adding the statements "$m := m + 1$" and "assign m to vertex v" as the first line of procedure *visit*. To see that this numbering corresponds to the numbering of the vertices created by a preorder traversal of the spanning tree, we need to show that each vertex has a smaller number than its children, and that the children have increasing numbers from left to right (assuming that each new child added to the tree comes to the right of its siblings already in the tree). Clearly the children of a vertex get added to the tree only after that vertex is added, so their number must exceed that of their parent. And if a vertex's sibling has a smaller number, then it must have already been visited, and therefore already have been added to the tree.

46. Note that a "lower" level is further down the tree, i.e., further from the root and therefore having a larger value. (So "lower" really means "greater than"!) This is similar to Exercise 40. Again notice that the order in which vertices are put into (and therefore taken out of) the list L is level-order. In other words, the root of the resulting tree comes first, then the vertices at level 1 (put into the list while processing the root), then the vertices at level 2 (put into the list while processing vertices at level 1), and so on. Now suppose that uv is a directed edge not in the tree. First assume that the algorithm processed u before it processed v. (In other words, u entered the list L before v did.) Since the edge uv is not in the tree, it must be the case that v was already in the list L when u was being processed. In order for this to happen, the parent p of v must have already been processed before u. Note that p's level in the tree is one less than v's level. Therefore u's level is greater than or equal to p's level but less than or equal to v's level, so this directed edge goes from a vertex at one level to a vertex either at the same level or one level below. Next suppose that the algorithm processed v before it processed u. Then v's level is at or above u's level, and there is nothing else to prove.

48. Maintain a global variable c, initialized to 0. At the end of procedure *visit*, add the statements "$c := c + 1$" and "assign c to v." We need to show that each vertex has a larger number than its children, and that the children have increasing numbers from left to right (assuming that each new child added to the tree comes to the right of its siblings already in the tree). A vertex v is not numbered until its processing is finished, which means that all of the descendants of v must have finished their processing. Therefore each vertex has a larger number than all of its children. Furthermore, if a vertex's sibling has a smaller number, then it must have already been visited, and therefore already have been added to the tree. (Note that listing the vertices by number gives a postorder traversal of the tree.)

50. Suppose that T_1 contains a edges that are not in T_2, so that the distance between T_1 and T_2 is $2a$. Suppose further that T_2 contains b edges that are not in T_3, so that the distance between T_2 and T_3 is $2b$. Now at

worst the only edges that are in T_1 and not in T_3 are those $a+b$ edges that are in T_1 and not in T_2, or in T_1 and T_2 but not in T_3. Therefore the distance between T_1 and T_3 is at most $2(a+b)$.

52. Following the construction of Exercise 51, we reduce the distance between spanning trees T_1 and T_2 by 2 when we remove edge e_1 from T_1 and add edge e_2 to it. Thus after applying this operation d times, we can convert any tree T_1 into any other spanning tree T_2 (where d is half the distance between T_1 and T_2).

54. By Exercise 16 in Section 9.5 there is an Euler circuit C in the directed graph. We follow C and delete from the directed graph every edge whose terminal vertex has been previously visited in C. We claim that the edges that remain in C form a rooted tree. Certainly there is a directed path from the root to every other vertex, since we only deleted edges that allowed us to reach vertices we could already reach. Furthermore, there can be no simple circuits, since we removed every edge that would have completed a simple circuit.

56. Since this is an "if and only if" statement, we have two things to prove. First, suppose that G contains a circuit $v_1, v_2, \ldots, v_k, v_1$, and without loss of generality, assume that v_1 is the first vertex visited in the depth-first search process. Since there is a directed path from v_1 to v_k, vertex v_k must have been visited before the processing of v_1 is completed. Therefore v_1 is an ancestor of v_k in the tree, and the edge $v_k v_1$ is a back edge. Now we have to prove the converse. Suppose that T contains a back edge uv from a vertex u to its ancestor v. Then the path in T from v to u, followed by this edge, is a circuit in G.

58. Clearly if G contains a back edge uv (i.e., u is a descendant of v), then it contains a circuit: follow the tree from v to u and then return to v along this edge. Conversely, suppose that G contains a circuit (without loss of generality, assume that no vertex is repeated until the circuit closes), and let u be the first vertex of that circuit visited by the depth-first search algorithm. Let v be the vertex on the circuit preceding u. Since all the other vertices of the circuit can be reached from u, the processing of u will not be complete until all of them have been visited. In particular, v must be a descendant of u in the tree, and therefore vu will be a back edge.