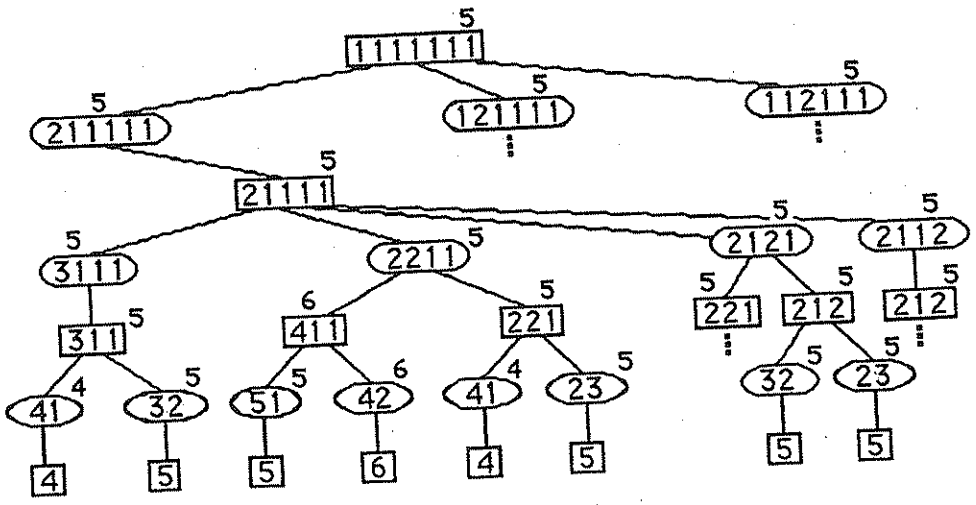
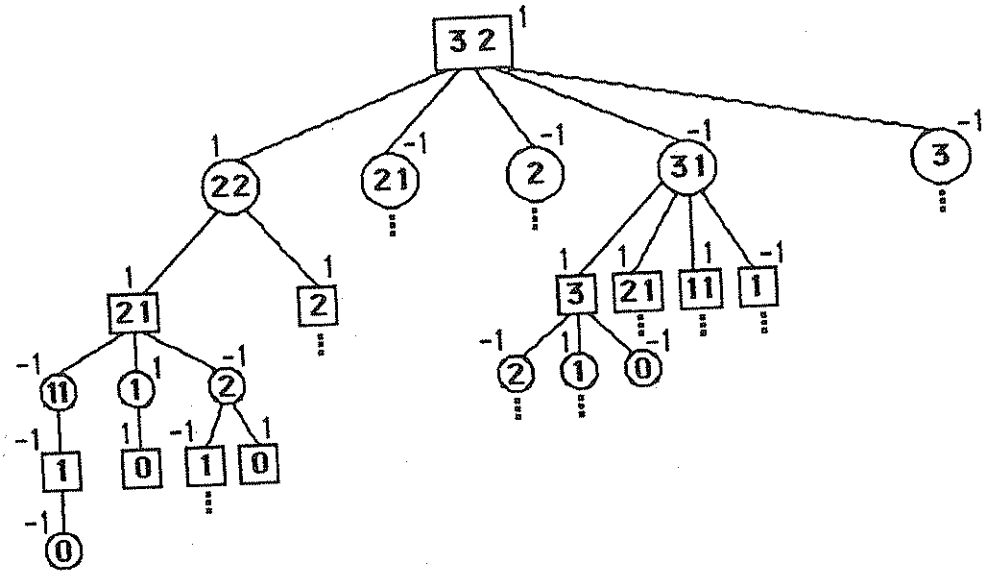


(b) The values have been filled in from the bottom. The value of each leaf is the number that is left (by definition of the payoff in this game). Then from the bottom up the value of each internal square vertex is the maximum of the values of its children, and the value of each internal circle vertex is the minimum of the values of its children. The value of the entire game is 5. Note that almost all the vertices have the value 5, so it is hard for the game to end with any other outcome, even if the players do not play their best strategies.

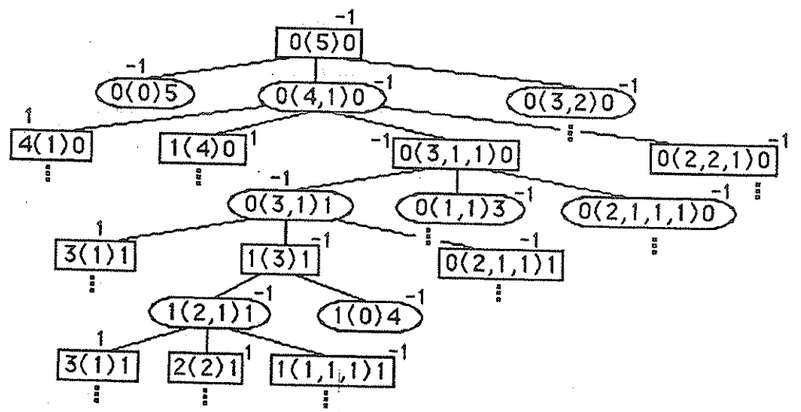
15. (a) This game and the analysis are very similar to nim. Really all that changes is that the value of a leaf is exactly opposite to what it was before. The relevant portion of the game tree is shown here.



(b) As usual we analyze the tree by finding the values from the bottom up. The circle 0's have the value -1 (since Red wins by reaching the position with no stones), and the square 0's have the value 1 (since Blue wins in this case). For internal vertices the minimax rule applies.

(c) Since the root has the value 1, the first player wins. He must move to the position 22 (taking one stone from the pile with three stones) if he wants to guarantee a win.

17. **(a)** A portion of the game tree is shown here; the complete game tree would be too large to fit on the page. From the initial position, which we denote by 0(5)0 to indicate that there is a pile of five stones remaining, and neither player has any stones in his own pot yet, the first player can either give the five stones to the other player, yielding position 0(0)5, or split the pile into two piles, yielding position $0(4,1)0$ or $0(3,2)0$.



(b) A few final positions (leaves) are shown in this tree, such as 1(0)4 in the next to last row. The position is a win for the first player if he has more stones in his pot, and it is a win for the second player if she has more stones in her pot. The position 1(0)4 is a win for the second player, for example, so it has value -1. The values of the vertices in the tree shown here were computed by extending the relevant branches of the tree, or by common sense (for instance, 3(1)1 is a win for the first player, since he already has a majority of the stones).

(c) Since the value of the root is -1, the second player wins this game. The strategy that she must follow is indicated by the paths of -1's. For example, if the first player moves to $0(4,1)0$ on his first move, then the second player can move to $0(3,1,1)0$ or $0(2,2,1)0$ as her response.

19. Suppose that there are two piles, of sizes m and n. If $m = n$ then Red wins as follows. If Blue ever merges the piles, then Red takes the resulting pile and wins. Otherwise Red copies Blue's take in the other pile (for instance, if Blue takes 17 stones from one pile, then Red takes 17 stones from the other). Since Red will never be at a loss for a move, she wins. On the other hand, if $m \neq n$, then Blue wins by taking $|m-n|$ stones from the larger pile to create the position of two piles of equal size, and then he is in the position that Red was in—the two piles are of equal size and he is the second player; thus Blue wins this game. To summarize: Red wins if and only if the piles are of equal size.

21. **(a)** Maintain a global list of positions already analyzed, and their values. As a first alternative when P is not a leaf, check the list and return the appropriate value if the position is on the list. Otherwise proceed with the recursive algorithm, but make sure to update the list before returning the value. Note that part of the information kept for each position already analyzed is the player whose turn it is to move; if the game's rules are symmetric, then a position in which Blue is about to move will have the negative value of the same position in which Red is about to move, and vice versa.

(b) Add one more parameter to the procedure, *depth_to_go*, which will keep track of how deeply to continue to recurse. The initial call to *game_value* uses a predetermined value of *depth_to_go*. Each recursive call to *game_value* passes on *depth_to_go* -1 (or perhaps some other value depending on the situation). As a first alternative when P is not a leaf, if *depth_to_go* $= 0$, then return the value determined by the heuristic evaluation function.

(c) Add one more parameter to the procedure, *goal*. The idea is that once the procedure realizes that its value is going to be worse than *goal*, it just returns *goal*. The initial call to *game_value* uses *goal* $= \infty$. Modify the first innermost **begin**...**end** block of the procedure to read as follows.

```
begin
    Q ← (P followed by move m)
    v' ← game_value(Q, Red, v)
    if v' > v then
        begin
            v ← v'
            if v ≥ goal then return(goal)
        end
end
```

Modify the second innermost **begin**...**end** block in a similar way (with each inequality reversed).

(d) The first outermost **begin**...**end** block is changed to read as follows (and the second in a dual way).

```
begin
    for each legal move m for Blue do
        if game_value(P followed by move m, Red) = 1
            then return(1)
    return(−1)
end
```

23. **(a)** The initial position is $P = \mathbf{N}$ (the set of natural numbers). A move consists of replacing P by a nonempty proper subset of P if P is infinite, or any proper subset of P if P is finite. It would be foolish for any player to move to a finite set, since then the opponent would immediately move to the empty set and win. Thus each player will always move to an infinite set, and the game will continue forever.

(b) Again the initial position is $P = \mathbf{N}$. This time a move consists of replacing P by a proper subset of P. The first player can win immediately by replacing $\mathbf{N}$ by $\emptyset$.

(c) As above, the initial position is $P = \mathbf{N}$. A move consists of replacing P by a finite proper subset of P. The game cannot go on forever, since as soon as the first move is made, say to a set Q, then only $|Q|$ more moves can be made before the empty set appears. On the other hand, consider the game that consists of the first player's moving to any subset of cardinality $l-1$, with each subsequent move being to remove one element from the remaining set. This game lasts l moves.

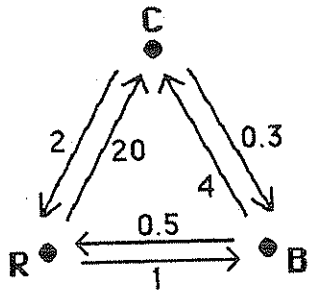
CHAPTER 10
GRAPHS AND DIGRAPHS WITH ADDITIONAL STRUCTURE

SECTION 10.1 Shortest Paths and Longest Paths

1. **(a)** By inspection we can write down the paths. Edge af is a path of length 1. There are two paths of length 2: a, b, f and a, e, f. There is one path of length 3: a, b, c, f.

   (b) The weight of path a, f is just the weight of the edge, namely 5. The weight of a, b, f is the sum of the weights of the edges ab and bf, namely $3 + 1 = 4$. Similarly the weight of a, e, f is $2 + 6 = 8$, and the weight of a, b, c, f is $3 + 3 + 3 = 9$.

3. The following picture shows the weighted digraph model of this situation. For convenience we have changed units, so that each weight represents 1000 pieces of mail. Thus, for example, the edge from B to C is labeled 4, since there are 4000 pieces of mail from banks to consumers; and the edge from C to B is labeled 0.3 since there are 300 pieces of mail in the opposite direction.



5. We represent the operation of the algorithm by a sequence of snapshots, showing the labels of all the vertices. The number following each vertex is the current value of *distance* from the source (vertex b in this exercise) to that vertex (using only vertices that are *final*). The superscript is the vertex from which the currently known path comes. An asterisk indicates that the vertex has been marked final. Once the sink (in this case vertex e) has been marked final, the algorithm is finished. Initially, the source has *distance* 0, and all the other vertices have *distance* ∞.

   $a:\infty \quad b:0 \quad c:\infty \quad d:\infty \quad e:\infty \quad f:\infty \quad g:\infty$

Since vertex b is the nonfinal vertex with smallest label, it becomes final and the *distance*'s to the other vertices are updated. In this case, the label on vertex a is 3, since there is an edge of weight 3 from b to a, and the superscript on a is b; and similarly for vertices c and f.

$$a:3^b \quad b^*:0 \quad c:3^b \quad d:\infty \quad e:\infty \quad f:1^b \quad g:\infty$$

Next vertex f becomes final. The label on vertex g becomes 5^f, since there is now a known path from the source to g passing through vertex f, of weight $1+4=5$, and similarly for vertex e. The labels on vertices a and c do not change, however, since the path through vertex f is no better than the previously known paths to these vertices. Thus our snapshot looks like this.

$$a:3^b \quad b^*:0 \quad c:3^b \quad d:\infty \quad e:7^f \quad f^*:1^b \quad g:5^f$$

We continue in this way, obtaining the following snapshots.

$$\begin{array}{lllllll} a^*:3^b & b^*:0 & c:3^b & d:\infty & e:5^a & f^*:1^b & g:5^f \\ a^*:3^b & b^*:0 & c^*:3^b & d:5^c & e:5^a & f^*:1^b & g:5^f \\ a^*:3^b & b^*:0 & c^*:3^b & d^*:5^c & e:5^a & f^*:1^b & g:5^f \\ a^*:3^b & b^*:0 & c^*:3^b & d^*:5^c & e^*:5^a & f^*:1^b & g:5^f \end{array}$$

At this point vertex e has been labeled, so we know that the minimum weight path has weight 5, and comes from a. Since the superscript for vertex a is b, we know that the path arrived at a from b. Thus the entire path is b, a, e.

7. We give a proof by mathematical induction on l. If $l=1$, then the (i,j)th entry of $A=A^l$, namely, a_{ij}, is the weight of the minimum weight walk of length 1 from vertex i to vertex j: Either there is a walk along the edge from i to j, or else there is no edge and $a_{ij}=\infty$. Assume the inductive hypothesis, that the statement is true for l. Then a minimum weight walk of length at most $l+1$ from i to j is either a minimum weight walk of length at most l, or consists of a minimum weight walk of length l to some vertex k, followed by edge kj. Let $a_{ij}^{\diamond l}$ denote the (i,j)th entry of $A^{\diamond l}$. The weight of a minimum weight walk of length at most l from i to j is thus $a_{ij}^{\diamond l}$, which can be written as $a_{ij}^{\diamond l}+0=a_{ij}^{\diamond l}+a_{jj}$; and the weight of a minimum weight walk of length l from i to k followed by the edge kj is $a_{ik}^{\diamond l}+a_{kj}$. Thus the (i,j)th entry of $A^{\diamond(l+1)}=A^{\diamond l}\diamond A$, namely $\min_{1\le k\le n}(a_{ik}^{\diamond l}+a_{kj})$, is the weight of a minimum weight walk of length at most $l+1$ from i to j.

9. Consider the digraph consisting of edge ab with weight 10, edge bc with weight -6, and edge ca with weight -7. There is no minimum weight walk from a to b, since we can make the weight of a walk as small as desired simply by traveling around the triangle sufficiently often before stopping at b.

11. We represent the execution of the algorithm in the same way as in the solution to Exercise 5. At each stage we find the vertex with the smallest label, make it final, and then update the labels on all the vertices adjacent to the newly finalized vertex, if a path using that vertex is less costly.

$a:0$	$b:\infty$	$c:\infty$	$d:\infty$	$e:\infty$	$f:\infty$	$g:\infty$	$h:\infty$	$i:\infty$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b:3^a$	$c:1^a$	$d:5^a$	$e:\infty$	$f:\infty$	$g:\infty$	$h:\infty$	$i:\infty$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b:3^a$	$c^*:1^a$	$d:5^a$	$e:3^c$	$f:3^c$	$g:\infty$	$h:5^c$	$i:\infty$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d:5^a$	$e:3^c$	$f:3^c$	$g:\infty$	$h:5^c$	$i:\infty$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d:4^e$	$e^*:3^c$	$f:3^c$	$g:\infty$	$h:5^c$	$i:5^e$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d:4^e$	$e^*:3^c$	$f^*:3^c$	$g:4^f$	$h:5^c$	$i:5^e$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g:4^f$	$h:5^c$	$i:5^e$	$j:\infty$	$k:\infty$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h:5^c$	$i:5^e$	$j:7^g$	$k:9^g$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i:5^e$	$j:7^g$	$k:9^g$	$l:\infty$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i^*:5^e$	$j:7^g$	$k:9^g$	$l:9^i$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i^*:5^e$	$j^*:7^g$	$k:9^g$	$l:9^i$	$z:\infty$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i^*:5^e$	$j^*:7^g$	$k^*:9^g$	$l:9^i$	$z:13^k$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i^*:5^e$	$j^*:7^g$	$k^*:9^g$	$l^*:9^i$	$z:11^l$
$a^*:0$	$b^*:3^a$	$c^*:1^a$	$d^*:4^e$	$e^*:3^c$	$f^*:3^c$	$g^*:4^f$	$h^*:5^c$	$i^*:5^e$	$j^*:7^g$	$k^*:9^g$	$l^*:9^i$	$z^*:11^l$

At this point we know that the minimum weight path has weight 11. Furthermore, by tracing the superscripts backward from vertex z, we see that a minimum weight path is a, c, e, i, l, z.

13. We assume that the vertices of the digraph are $0, 1, \ldots, n$, where 0 is the source and n is the sink. The weight of edge ij is $w(i, j)$. We compute the completion time for each vertex, $T(v)$, and we keep track of the pointers needed to reconstruct the critical path ($previous(v)$ is a predecessor of v that causes the completion time of v to be as large as it is).

```
procedure critical_path(G : weighted acyclic digraph)
    for i ← 1 to n do
        T(i) ← −1 {to indicate no time yet assigned}
    T(0) ← 0 {completion time of the source is 0}
    for i ← 1 to n do {compute T for (i.e., label) another vertex}
        begin
            v ← n
            v' ← 0
            while v' ≠ v do {find vertex all of whose predecessors are labeled}
                begin
                    v' ← v
                    for j ← 1 to n − 1 do
                        if jv is an edge and T(j) = −1 then v ← j
                end {v is such a vertex}
            for j ← 1 to n − 1 do {label v}
                if jv is an edge and T(j) + w(j, v) > T(v) then
                    begin
                        T(v) ← T(j) + w(j, v)
                        previous(v) ← j
                    end
        end
    return(T, previous)
```

15. We claim that a maximum weight Hamilton cycle is b, c, e, d, a, b, whose weight is $5 + 5+4+2+3 = 19$. (This cycle was obtained by inspection, by trying to use edges with as large a weight as possible.) The proof that this cycle has maximum weight is similar to the argument given in Example 6. The best that we could possibly hope for would be to use the two edges of largest weight adjacent to each vertex. This would give a total weight of $[(3+3)+(5+4)+(5+5)+(4+2)+(5+4)]/2 = 20$. Now in fact this is impossible to achieve, since if we used bc and ce (the only edges of weight 5 incident to c), then we could not use be (otherwise the cycle would close prematurely); this means that we could not achieve the desired maximum weight as we pass through vertex b. Therefore $20 - 1 = 19$ is the best possible.

17. **(a)** There are $n!$ Hamilton cycles. (We are assuming that we do not recognize two cycles with the same edge sets as identical. Thus a cycle is determined by a permutation of the vertices of the graph.) Therefore it would take

$$20! \cdot 10^{-5} \text{ seconds} \approx 2.4 \times 10^{13} \text{ seconds} \approx 770{,}000 \text{ years}$$

to carry out this analysis.

(b) We are asked to find the largest n such that $n! \cdot 10^{-5} \leq 3600$. This is equivalent to $n! \leq 3.6 \times 10^8$. Since $12! \approx 4.8 \times 10^8$ and $11! \approx 4.0 \times 10^7$, the answer is 11.

19. **(a)** If there is no path from s to t, then vertex t can never be marked final. Thus the **while** loop will never terminate, and the procedure will never halt. (What will happen is that the vertex v chosen in that loop will continue to be the last vertex chosen.) Similarly, it will not halt if there is no directed path from s to t, in the case of a digraph.

(b) Suppose that we insert before the statement $final(v) \leftarrow$ **true** the following statement.

$$\textbf{if } m = \infty \textbf{ then return}(distance, previous)$$

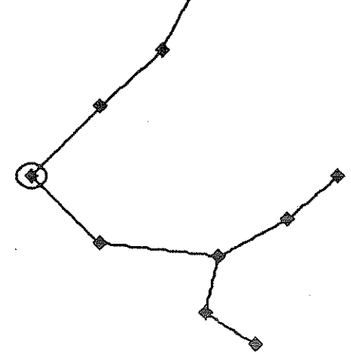
Then in case the algorithm was unable to find a new vertex to mark *final* it will immediately halt.

21. We can apply Dijkstra's algorithm $n-1$ times, once with each vertex (except one) as the source (we change the stopping condition, as in Exercise 12, so that all distances from the source are computed). This will determine the distances between all pairs of vertices in $O(n^3)$ steps ($O(n)$ iterations of an $O(n^2)$ algorithm). The smallest among these n^2 values is the diameter, and we can find it in $O(n^2)$ steps.

23. Work backward from the sink, labeling vertices. The label $L(v)$ is intended to be the latest time at which the subproject represented by v may be started in order to finish the entire project by time $T(t)$. We start by setting $L(t) = T(t)$, which really says nothing, except that we get to the end of the project at the final completion time. Successively find a vertex u all of whose successors have been labeled (this is possible because the digraph is acyclic), and set $L(u) = \min(L(v) - w(u,v))$, where the minimum is taken over all successors v of u. We do this to maintain the possibility of arriving at vertex v at the right time.

SECTION 10.2 Minimum Spanning Trees

1. **(a)** We carefully plot the 10 points given, together with the origin, obtaining the following figure (ignore the lines temporarily).



(b) According to Prim's algorithm, we can build the tree by successively adding in edges of smallest weight (which here is physical distance) between vertices already in the tree and vertices not yet included. Let us start at the central computer (the circled point in the figure). The nearest vertices are those at $(11, 11)$ and $(11, -11)$, so they are added to the tree. Next the vertex at $(21, 20)$ is closest (it is near the vertex at $(11, 11)$), so we add it. We continue in this way, just using our eyesight to determine the unused vertex closest to the tree at each stage. The final tree is as shown above.

3. In each case we break all ties in favor of alphabetical order (the edges are assumed to be listed in the alphabetical order of their endpoints). At each stage in the execution of the algorithm, we add an edge of smallest weight whose addition will not create a cycle (we make this determination visually).

(a) We add the edges in the following order: bf, ae, cd, de, ab, fg. The total weight of the tree is $1+2+2+2+3+4=14$.

(b) We add the edges in the following order: bd, ad, cd, ae. The tree has weight $1+2+2+3=8$.

(c) We add the edges in the following order: ac, de, fg, hi, bc, ce, cf, eh, ij, jl, lz, jk, to obtain a minimum spanning tree of weight $1+1+1+1+2+2+2+2+2+2+2+3=21$.

(d) We add the edges in the following order: ad, bc, ae, be; the tree has weight $1+1+2+2=6$.

5. **(a)** When applying Prim's algorithm, we can start with vertex v and edge e. The minimum spanning tree we find will perforce contain e.

   (b) Consider K_3, in which each edge has weight 1. Let e be one of the edges. Then the conditions of this problem are satisfied. However, the tree consisting of the other two edges of K_3 is a minimum spanning tree, and it does not contain e.

7. **(a)** We express the algorithm in pseudocode, where the argument G is a complete weighted graph with vertices $1, 2, \ldots, n$. We construct *path* as a list of vertices by chosing at each stage an edge of smallest weight that will take us to a vertex we have not visited before. Clearly after $n-1$ iterations this will produce a Hamilton cycle, but there is no reason to believe that it will produce a minimum weight Hamilton cycle.

```
procedure greedy_hamilton(G : complete weighted graph)
    path ← (1)
    visited(1) ← true
    for j ← 2 to n do
        visited(j) ← false
    v ← 1
    for i ← 1 to n − 1 do
        begin
            m ← ∞
            for j ← 1 to n do {find edge of smallest weight
                                to an unvisited vertex}
                if (¬visited(j)) ∧ (w(v, j) < m) then
                    begin
                        w ← j
                        m ← w(v, j)
                    end
            path ← path followed by w
            visited(w) ← true
            v ← w
        end
    path ← path followed by 1 {close the cycle}
    return(path)
```

   (b) Following this algorithm, we start at vertex a and choose the least weight edge incident to the vertex at which we are currently located that will take us to a vertex that we have not yet visited. In doing so we obtain the cycle a, d, b, e, c, a. This cycle has total weight $2+1+4+5+3=15$, whereas we saw in Example 6 of Section 10.1 that a minimum weight Hamilton cycle has weight 13.

9. We assume that the edges, denoted $e_1, e_2, \ldots, e_m$, have been sorted so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. We will have the algorithm go through the list of edges only once, including any edge that does not complete a cycle in the spanning forest being generated as we go along. In pseudocode the algorithm goes as follows.