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.

```
procedure sorted_kruskal(G : connected weighted graph)
   {same setting as Algorithm 2}
      j ← 1
      F ← the empty tree
      for k ← 1 to n − 1 do {add n − 1 edges in all}
         begin
            while F ∪ {e_j} contains a cycle do
                  j ← j + 1 {pass over any edges that cannot be used}
            F ← F ∪ {e_j} {this edge does not form a cycle}
            j ← j + 1 {go on to consider the next edge}
         end
      return(F)
```

11. **(a)** There are at least two "obvious" ways to do this. One is to follow Prim's or Kruskal's Algorithm, but add the *largest* weight edge at each stage, rather than the smallest weight edge (consistent with the rules of the algorithm). The other solution would be to replace each edge weight $w(e)$ by its negative $-w(e)$ and apply the algorithm as written (we never used the fact that weights were positive).

(b) Following the first of the suggestions given here, we select the edges in the following order: bc, df, ab, be, de. Thus the maximum spanning tree has weight 31.

13. We will show that after each deletion, the resulting graph still *contains* a minimum spanning tree of G. Then since we end up with a tree, it must be a minimum spanning tree. We proceed by induction on the number of deletions, the base case being $T = G$ (no deletions); we already know that G contains a minimum spanning tree. Now suppose that T contains a minimum spanning tree S, and suppose that in the operation of this algorithm, edge e is deleted to form T'. We must show that T' contains a spanning tree. If e is not in S, there is nothing to prove. If e is in S, consider $S - \{e\}$. It is a forest with two components, C_1 and C_2. Since e was in a cycle in T, there must be an edge e' in T joining a vertex in C_1 and a vertex in C_2. Since e' was not deleted from T to form T' (even though e' is in a cycle with e), we know that $w(e') \leq w(e)$. Furthermore, $S' = S - \{e\} \cup \{e'\}$ is a tree. The weight of S' is at most the weight of S. Thus S' is a minimum spanning tree of G, and S' is a subgraph of T', since e' is in T'. In other words, we have found the desired minimum spanning tree contained in T'.

15. It is *not* enough to observe here that if the weights of the edges are unique, then there are no choices involved in the execution of Prim's or Kruskal's algorithm. All we proved in this section was that these two algorithms produce minimum spanning trees; there may well be (and are) other algorithms that give us minimum spanning trees as well, and we have no control over how they might operate. What is asked for here is a proof that *no* algorithm could ever possibly produce a different minimum spanning tree (under the hypothesis that the edge weights are all distinct).

For a valid proof, let T be the minimum spanning tree of G produced by Prim's algorithm, containing edges $e_1, e_2, \ldots, e_{n-1}$. We must show that there is no other minimum spanning tree. Suppose instead that G does have another minimum spanning tree, and let T' be one that agrees with T on $e_1, e_2, \ldots, e_k$ for the largest possible $k < n-1$. Now proceed as in the proof of the correctness of Prim's algorithm. The tree T'' constructed there has strictly smaller weight than T', since $w(e_{k+1}) < w(e')$. This is impossible, since T' was a minimum spanning tree. Thus no such minimum spanning tree $T' \neq T$ exists, so T is the unique minimum spanning tree.

17. Let T_k be the tree constructed after the kth pass through the outer loop of Prim's algorithm. We will show by induction that each T_k is contained in some minimum spanning tree of G; Theorem 1 follows by taking $k = n-1$. The assertion is trivial for $k = 0$. Suppose that it is true for T_k. Let S be a minimum spanning tree containing T_k. We must show that $T_k \cup \{e_{k+1}\}$ is contained in a minimum spanning tree. If e_{k+1} is in S, we are done. If not, look at $S \cup \{e_{k+1}\}$. It contains a cycle. Since T_{k+1} does not contain a cycle, there is some edge in $S - T_{k+1}$ on the cycle. We follow the cycle around, starting at e_{k+1}, until we come to the first such edge; call it e. Thus e was a candidate when e_{k+1} was added to T_k, so $w(e) \geq w(e_{k+1})$. But $S' = S \cup \{e_{k+1}\} - \{e\}$ is a tree, and its weight does not exceed the weight of S. Thus S' is the desired minimum spanning tree containing T_{k+1}.

SECTION 10.3 Flows

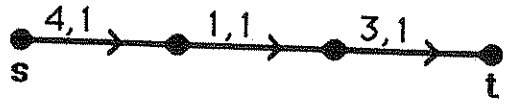
1. To produce a flow we need to choose a nonnegative flow in each edge no greater than the capacity of the edge, so that the net flow into each vertex other than the source and sink is 0. There are many possibilities, such as the zero flow and the flows given in the following table (none of these is a maximum flow).

su	sv	uv	ut	vt
1	1	0	1	1
2	0	2	0	2
3	0	2	1	2
3	1	2	1	3
2	1	2	0	3

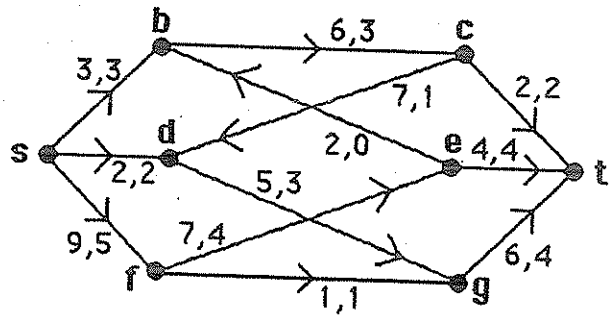
3. We add 1 to the flow in each edge of the path, namely edges sv, vu, and ut. Note that the flow along vu is currently -2, since there is a flow of 2 from u to v. Hence we increase it to -1, which means a flow of 1 from u to v. The new flow is as follows.

su	sv	uv	ut	vt
4	1	1	3	2

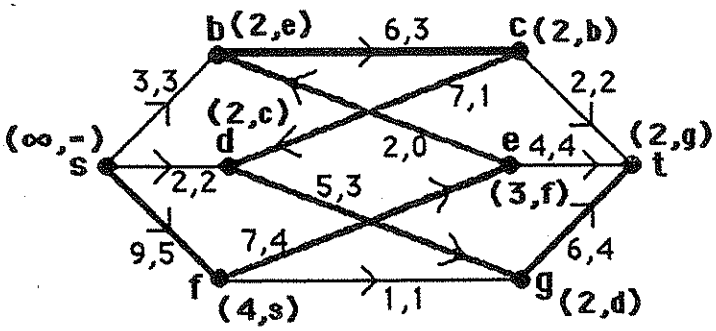
5. As a simple example we have the following network. The "bottleneck" is caused by the edge in the middle; it is saturated, but neither of the other two edges is saturated.



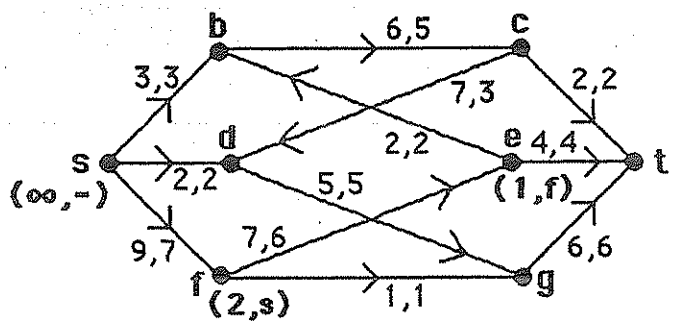
7. We might try to push flow along some obvious paths. We can certainly send 2 units of flow along the top: (s, b, c, t); 2 units of flow along a lower middle road (s, d, g, t), and 1 unit along the bottom (s, f, g, t). Further inspection shows that we can augment this by 4 units along s, f, e, t, and again by 1 unit along s, b, c, d, g, t. In all this produces the flow shown here. We will save any further attempts at improving this for Exercise 9.



9. We begin with the flow we obtained in Exercise 7 and carry out the labeling algorithm. First we label the source $(\infty, -)$ to indicate that any amount of extra flow can arrive at the source. Then we process the source (the only labeled but as yet unprocessed vertex). Only one edge from the source, sf, is not yet saturated. There is an excess capacity in it of $9 - 5 = 4$ units. Therefore we label f with $(4, s)$ to indicate that 4 units of flow can come from the source to vertex f. Now f is the only unprocessed labeled vertex, so we look at it. The edge to vertex e has excess capacity of $7 - 4 = 3$ units, so we label e with $(3, f)$. Similarly there is unused capacity $(2 - 0 = 2$ units worth) in edge eb, so we can now label vertex b with $(2, e)$. Again there is unused capacity $(6 - 3 = 3$ units worth) in edge bc, so we can now label vertex c. Since only 2 units could arrive at vertex b, only 2 units can go on to vertex c, so we label c with $(2, b)$; in other words, the numerical label on c is the minimum of the excess capacity through bc and the numerical label on b. We continue in this way and see that this extra 2 units can make it all the way to the sink. The completed labeling process gives us the following picture.



The heavy lines in this picture show the augmentation path s, f, e, b, c, d, g, t, which can be obtained in reverse order by following the second coordinates of the labels, starting with the sink t. We add 2 to the flow in each edge along this path, obtaining the flow shown below (ignore the labels for a moment).



Now we label this network. The labels on s, f, and e are obtained in the same manner as before. We see that 1 unit of flow can make its way to vertex e. However, at that point we are stuck, since both edges between e and unvisited vertices are now saturated. Therefore the algorithm tells us that we have found a maximum flow. In fact we can also read off the minimum cut from this picture. It is the set of edges leading from labeled vertices to unlabeled ones, namely sb, sd, fg, eb, and et. The sum of the flows in these three saturated edges, $3+2+1+2+4=12$, is the value of this maximum flow. This value is also the total flow out of the source and the total flow into the sink.

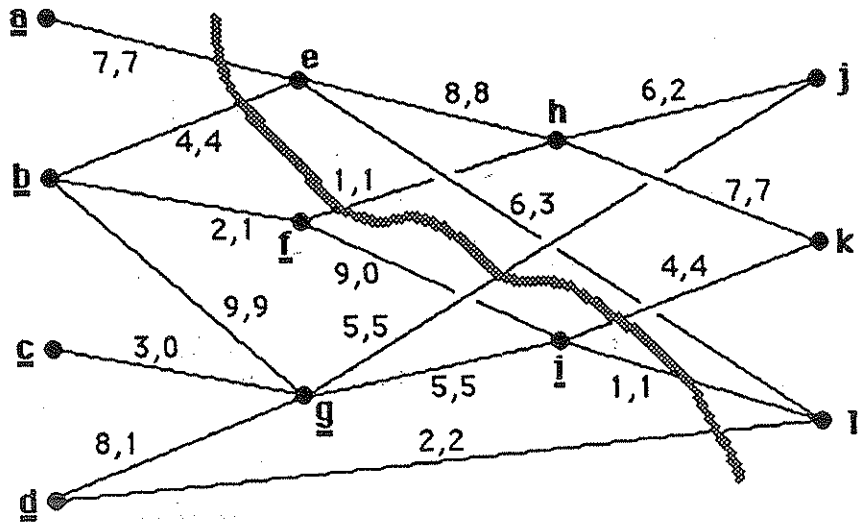
11. We present the operation of the algorithm in the following table, showing the flows at each stage and the augmentations performed. The first row shows the zero flow and the augmentation path found on the first pass. The new flow is shown in the second row, and so on. The final flow (as well as the cut produced by the algorithm) is shown in the last row. Note that there happened to be no "backward" edges involved here, in which the flow decreased from one pass to the next. The flow we obtain is the same flow as that found in the solution to Exercise 9.

sb	*sd*	*sf*	*bc*	*eb*	*dg*	*cd*	*fe*	*fg*	*ct*	*et*	*gt*	
0	0	0	0	0	0	0	0	0	0	0	0	augment along s, b, c, t by 2
2	0	0	2	0	0	0	0	0	2	0	0	augment along s, f, e, t by 4
2	0	4	2	0	0	0	4	0	2	4	0	augment along s, d, g, t by 2
2	2	4	2	0	2	0	4	0	2	4	2	augment along s, b, c, d, g, t by 1
3	2	4	3	0	3	1	4	0	2	4	3	augment along s, f, g, t by 1
3	2	5	3	0	3	1	4	1	2	4	4	augment along s, f, e, b, c, d, g, t by 2
3	2	7	5	2	5	3	6	1	2	4	6	cut $\{sb, sd, fg, eb, et\}$ determined

13. We could apply the Ford–Fulkerson algorithm to find the maximum flow and minimum cut, but let us try to do it by inspection. In other words, we will find the augmentation paths by just looking for them as intelligent human beings looking at a useful picture, rather than by the mechanical labeling process. We might begin by pushing 2 units of flow along the path a, b, c, d, g, to come in through the vertex d (and note that this is as much as we can possibly get through d, since the only edge coming into d has capacity 2), and 4 units of flow along a, f, g (and this, too, is the best we can do, since the capacity of edge fg is only 4). The argument given here shows that we have found the maximum flow, namely $2+4=6$. The corresponding cut is the two saturated edges fg and cd; they are the only two edges from $\{a, b, c, e, f\}$ to $\{d, g\}$.

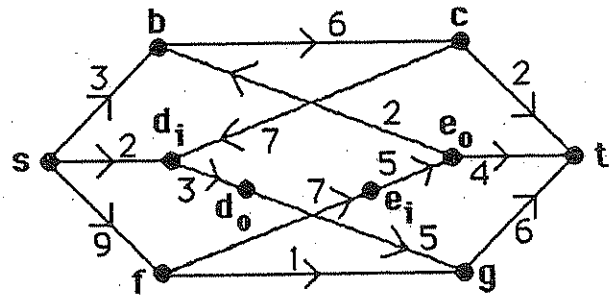
15. Add a new vertex S to be the new source, and put an edge of capacity ∞ from S to each original source. This allows as much of the commodity as necessary to reach each original source unimpeded.

17. Following the ideas in Exercises 15 and 16, we can imagine a supersource S to the left, with edges Sa, Sb, Sc, and Sd, all of capacity ∞, and a supersink T to the right with edges jT, kT, and lT, again all with infinite capacity. Then we can apply the Ford–Fulkerson algorithm, or try to find a maximum flow by inspection. Let us try the latter. We really need not be concerned with the supersource and supersink explicitly—we just think of as much flow as we need emanating from each of the four sources and as much as we desire being absorbed by each of the three sinks. First we push 7 units of flow along a, e, h, k. Four more units can travel along be; at e three of the units follow el to l, and the remaining unit follows the path e, h, j. Now we try sending 9 units of flow along bg. At g, five of the units then travel along gj, and the remaining 4 units follow g, i, k. We can sneak one more unit in along the augmentation path b, f, h, j and one more along d, g, i, l. Finally, we can send 2 units directly from d to l. This gives us a total flow of 24. To summarize, the flows we have obtained are shown in the following picture.



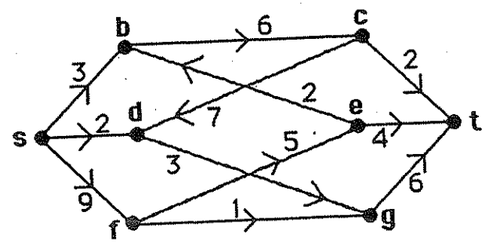
A few more minutes of staring at the network fails to show us any more augmentation paths, so we think that maybe this is a maximum flow. To test this out, we see what one pass of the labeling algorithm will produce. Vertices a, b, c, d, f, g, and i will be labeled, but then the saturated edges ae, be, fh, gj, ik, il, and dl—all the edges from labeled vertices to unlabeled ones—form a cut, and its capacity is, indeed, $7+4+1+5+4+1+2=24$. Our picture above shows the labeled vertices underlined, and the edges in the cut are those that the gray swatch passes through.

19. First we alter the network using the idea in Exercise 18, expanding vertex d into vertices d_i and d_o (the subscripts stand for "in" and "out") with an edge $d_i d_o$ of capacity 3, and similarly for vertex e. The resulting network is shown here. Note that edges into d or e are now directed into d_i or e_i, and edges out of d or e are now directed out of d_o or e_o.



Before we look for a maximum flow, let us simplify things a bit. Vertex d_o has only one edge coming in and one edge going out, so we can do away with it entirely by replacing the path d_i, d_o, g by an edge $d_i g$ of capacity 3 (the smaller of the capacities of the two edges in series that we are eliminating). Similarly, we replace f, e_i, e_o by edge

fe_o, with capacity 5. The network now looks like this. (We have removed the subscripts for convenience.)



Application of the Ford–Fulkerson algorithm (or finding augmentation paths by inspection) gives us the following maximum flow, with value 10. The corresponding cut is $\{dg, fg, ct, et\}$.

sb	sd	sf	bc	dg	fe	fg	cd	ct	eb	et	gt
3	1	6	4	3	5	1	2	2	1	4	4

21. One approach is to replace each undirected edge by a pair of antiparallel edges, each with the capacity of the undirected edge. This allows flow to go in either direction. After each new flow is found, we combine the flows on all pairs of antiparallel edges into a net flow in one of each pair and no flow in the other.

There is a better, more general way to look at this problem, however. Each directed edge currently comes with a maximum capacity, C_{ij}; its minimum capacity is 0, indicating that no flow can go backward in an edge. If we want to allow flow in either direction, then we can set the minimum capacity to be a negative number, D_{ij}. More generally, let D_{ij} be the minimum allowable flow in edge ij. We require simply that $D_{ij} \leq C_{ij}$ for each edge and that the flow F_{ij} satisfy $D_{ij} \leq F_{ij} \leq C_{ij}$. In the case of undirected edges we will take $D_{ij} = -C_{ij}$ and impose an arbitrary direction on each edge. The algorithm now takes on more of a symmetric flavor. The portion of the algorithm in which backward oriented edges are considered is replaced with the following code.

```
else if wv is an edge of G with F_wv > D_wv then
  begin
    Δ' ← min(Δ, F_wv − D_wv)
    { Δ' is necessarily greater than 0 }
    label w : (Δ', v)
    put w into L
  end
```

23. Following the hint, let us look for a minimum cut separating a from z. One good candidate seems to be $\{ab, ac, de\}$, with a value of $3 + 1 + 1 = 5$. If we can find a flow of 5, then we will know that we have found both the maximum flow and the minimum cut. It is not hard to find one, such as the following, by inspection. (We are free to orient the edges in a convenient direction. The edges not mentioned here have a zero flow.)

ad	de	ac	ce	ei	ab	bc	bf	fg	ch	hi	ij	hg	il	lz	jg	gk	kz
1	1	1	1	2	3	2	1	1	2	1	2	1	1	1	2	4	4