Chapter 9: Graph Algorithms

9.1 The following ordering is arrived at by using a queue and assumes that vertices appear on an adjacency list alphabetically. The topological order that results is then

s, G, D, H, A, B, E, I, F, C, t

9.2 Assuming the same adjacency list, the topological order produced when a stack is used is

s, G, H, D, A, E, I, F, B, C, t

Because a topological sort processes vertices in the same manner as a breadth-first search, it tends to produce a more natural ordering.

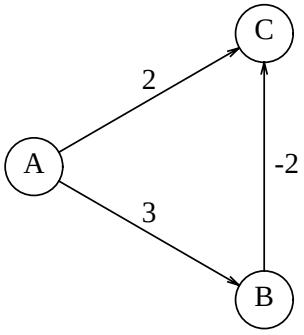
9.4 The idea is the same as in Exercise 5.10.

9.5 (a) (Unweighted paths) A->B, A->C, A->B->G, A->B->E, A->C->D, A->B->E->F.

(b) (Weighted paths) A->C, A->B, A->B->G, A->B->G->E, A->B->G->E->F, A->B->G->E->D.

9.6 We'll assume that Dijkstra's algorithm is implemented with a priority queue of vertices that uses the *DecreaseKey* operation. Dijkstra's algorithm uses $|E|$ *DecreaseKey* operations, which cost $O(\log_d |V|)$ each, and $|V|$ *DeleteMin* operations, which cost $O(d \log_d |V|)$ each. The running time is thus $O(|E| \log_d |V| + |V| d \log_d |V|)$. The cost of the *DecreaseKey* operations balances the *Insert* operations when $d = |E|/|V|$. For a sparse graph, this might give a value of d that is less than 2; we can't allow this, so d is chosen to be max (2, $\lfloor |E|/|V| \rfloor$). This gives a running time of $O(|E| \log_{2+|E|/|V|} |V|)$, which is a slight theoretical improvement. Moret and Shapiro report (indirectly) that d-heaps do not improve the running time in practice.

9.7 (a) The graph shown here is an example. Dijkstra's algorithm gives a path from A to C of cost 2, when the path from A to B to C has cost 1.



(b) We define a pass of the algorithm as follows: Pass 0 consists of marking the start vertex as known and placing its adjacent vertices on the queue. For $j > 0$, pass j consists of marking as known all vertices on the queue at the end of pass $j - 1$. Each pass requires linear time, since during a pass, a vertex is placed on the queue at most once. It is easy to show by induction that if there is a shortest path from s to v containing k edges, then d_v will equal the length of this path by the beginning of pass k. Thus there are at most $|V|$ passes,

giving an $O(|E||V|)$ bound.

9.8 See the comments for Exercise 9.19.

9.10 (a) Use an array *Count* such that for any vertex u, *Count[u]* is the number of distinct paths from s to u known so far. When a vertex v is marked as known, its adjacency list is traversed. Let w be a vertex on the adjacency list.

If $d_v + c_{v,w} = d_w$, then increment *Count[w]* by *Count[v]* because all shortest paths from s to v with last edge (v,w) give a shortest path to w.

If $d_v + c_{v,w} < d_w$, then p_w and d_w get updated. All previously known shortest paths to w are now invalid, but all shortest paths to v now lead to shortest paths for w, so set *Count[w]* to equal *Count[v]*. Note: Zero-cost edges mess up this algorithm.

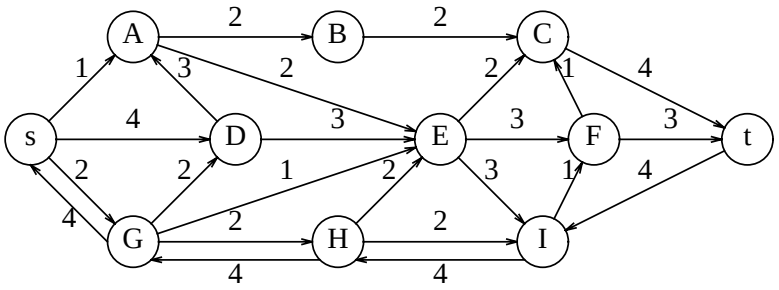
(b) Use an array *NumEdges* such that for any vertex u, *NumEdges[u]* is the shortest number of edges on a path of distance d_u from s to u known so far. Thus *NumEdges* is used as a tiebreaker when selecting the vertex to mark. As before, v is the vertex marked known, and w is adjacent to v.

If $d_v + c_{v,w} = d_w$, then change p_w to v and *NumEdges[w]* to *NumEdges[v]+1* if *NumEdges[v]+1 < NumEdges[w]*.

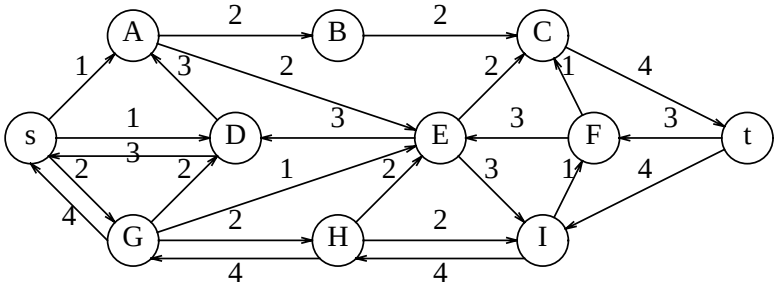
If $d_v + c_{v,w} < d_w$, then update p_w and d_w, and set *NumEdges[w]* to *NumEdges[v]+1*.

9.11 (This solution is not unique).

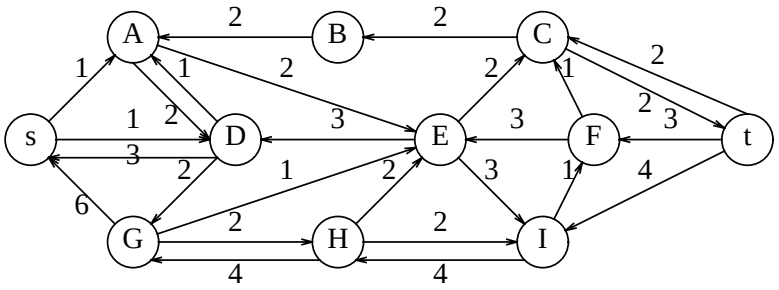
First send four units of flow along the path s, G, H, I, t. This gives the following residual graph:



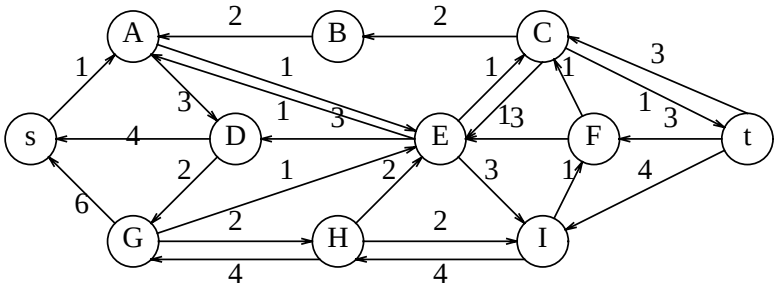
Next, send three units of flow along s, D, E, F, t. The residual graph that results is as follows:



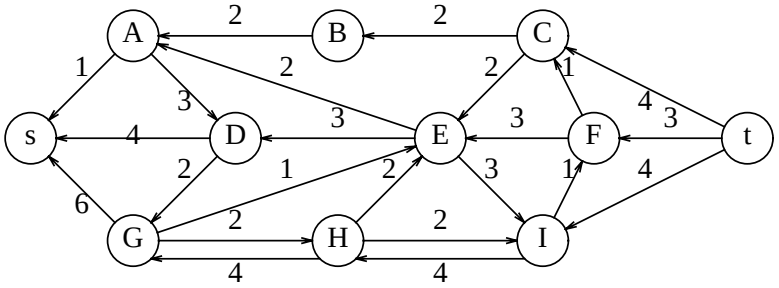
Now two units of flow are sent along the path s, G, D, A, B, C, t, yielding the following residual graph:



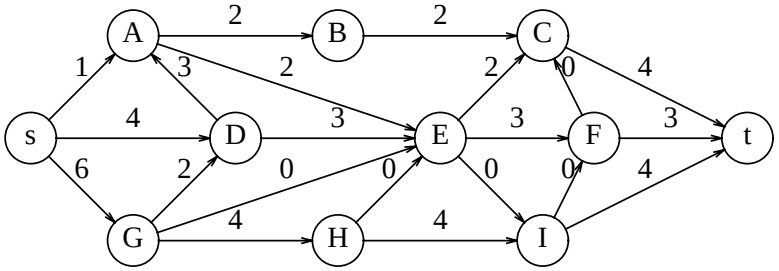
One unit of flow is then sent along s, D, A, E, C, t:



Finally, one unit of flow can go along the path s, A, E, C, t:



The preceding residual graph has no path from s to t. Thus the algorithm terminates. The final flow graph, which carries 11 units, is as follows:



This flow is not unique. For instance, two units of the flow that goes from G to D to A to E could go by G to H to E.

9.12 Let T be the tree with root r, and children r_1, r_2, ..., r_k, which are the roots of T_1, T_2, ..., T_k, which have maximum incoming flow of c_1, c_2, ..., c_k, respectively. By the problem statement, we may take the maximum incoming flow of r to be infinity. The recursive function *FindMaxFlow(T, IncomingCap)* finds the value of the maximum flow in T (finding the actual flow is a matter of bookkeeping); the flow is guaranteed not to exceed *IncomingCap*.

If T is a leaf, then *FindMaxFlow* returns *IncomingCap* since we have assumed a sink of infinite capacity. Otherwise, a standard postorder traversal can be used to compute the maximum flow in linear time.

```
FlowType
FindMaxFlow( Tree T, FlowType IncomingCap )
{
        FlowType ChildFlow, TotalFlow;

        if( IsLeaf( T ) )
                return IncomingCap;
        else
        {
                TotalFlow = 0;
                for( each subtree Ti of T )
                {
                        ChildFlow = FindMaxFlow( Ti, min( IncomingCap, ci ) );
                        TotalFlow += ChildFlow;
                        IncomingCap -= ChildFlow;
                }
                return TotalFlow;
        }
}
```

9.13 (a) Assume that the graph is connected and undirected. If it is not connected, then apply the algorithm to the connected components. Initially, mark all vertices as unknown. Pick any vertex v, color it red, and perform a depth-first search. When a node is first encountered, color it blue if the DFS has just come from a red node, and red otherwise. If at any point, the depth-first search encounters an edge between two identical colors, then the graph is not bipartite; otherwise, it is. A breadth-first search (that is, using a queue) also works. This problem, which is essentially two-coloring a graph, is clearly solvable in linear time. This contrasts with three-coloring, which is NP-complete.

(b) Construct an undirected graph with a vertex for each instructor, a vertex for each course, and an edge between (v,w) if instructor v is qualified to teach course w. Such a graph is bipartite; a matching of M edges means that M courses can be covered simultaneously.

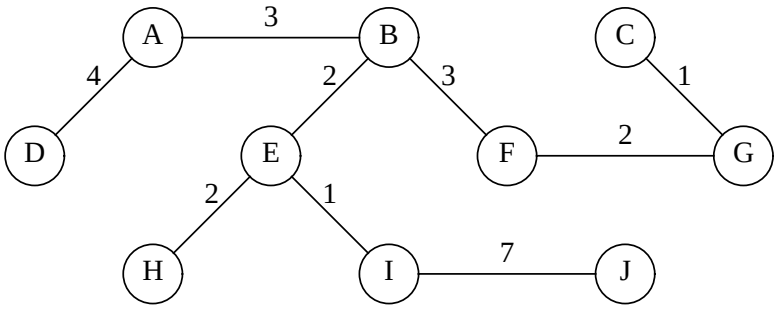
(c) Give each edge in the bipartite graph a weight of 1, and direct the edge from the instructor to the course. Add a vertex s with edges of weight 1 from s to all instructor vertices. Add a vertex t with edges of weight 1 from all course vertices to t. The maximum flow is equal to the maximum matching.

(d) The running time is $O(|E||V|^{1/2})$ because this is the special case of the network flow problem mentioned in the text. All edges have unit cost, and every vertex (except s and t) has either an indegree or outdegree of 1.

9.14 This is a slight modification of Dijkstra's algorithm. Let f_i be the best flow from s to i at any point in the algorithm. Initially, $f_i = 0$ for all vertices, except $s: f_s = \infty$.

At each stage, we select v such that f_v is maximum among all unknown vertices. Then for each w adjacent to v, the cost of the flow to w using v as an intermediate is min $(f_v, c_{v,w})$. If this value is higher than the current value of f_w, then f_w and p_w are updated.

9.15 One possible minimum spanning tree is shown here. This solution is not unique.



9.16 Both work correctly. The proof makes no use of the fact that an edge must be nonnegative.

9.17 The proof of this fact can be found in any good graph theory book. A more general theorem follows:

Theorem: Let $G = (V, E)$ be an undirected, unweighted graph, and let A be the adjacency matrix for G (which contains either 1s or 0s). Let D be the matrix such that $D[v][v]$ is equal to the degree of v; all nondiagonal matrices are 0. Then the number of spanning trees of G is equal to the determinant of $A + D$.

9.19 The obvious solution using elementary methods is to bucket sort the edge weights in linear time. Then the running time of Kruskal's algorithm is dominated by the *Union/Find* operations and is $O(|E|\alpha(|E|, |V|))$. The Van-Emde Boas priority queues (see Chapter 6 references) give an immediate $O(|E|\log\log|V|)$ running time for Dijkstra's algorithm, but this isn't even as good as a Fibonacci heap implementation.

More sophisticated priority queue methods that combine these ideas have been proposed, including M. L. Fredman and D. E. Willard, "Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths," *Proceedings of the Thirty-first Annual IEEE Symposium on the Foundations of Computer Science* (1990), 719-725. The paper presents a linear-time minimum spanning tree algorithm and an $O(|E| + |V| \log|V| / \log\log|V|)$ implementation of Dijkstra's algorithm if the edge costs are suitably small.

9.20 Since the minimum spanning tree algorithm works for negative edge costs, an obvious solution is to replace all the edge costs by their negatives and use the minimum spanning tree algorithm. Alternatively, change the logic so that < is replaced by >, *Min* by *Max*, and vice versa.

9.21 We start the depth-first search at A and visit adjacent vertices alphabetically. The articulation points are C, E, and F. C is an articulation point because $Low[B] \geq Num[C]$; E is an

articulation point because $Low[H] \geq Num[E]$; and F is an articulation point because $Low[G] \geq Num[F]$; the depth-first spanning tree is shown in Fig. 9.1.

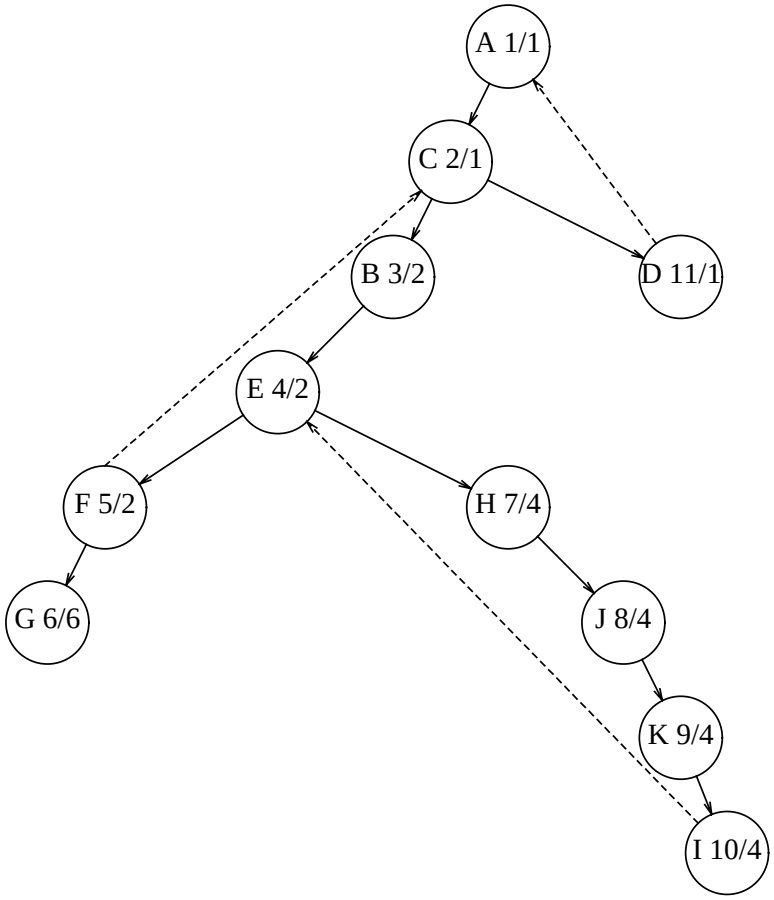


Fig. 9.1.

9.22 The only difficult part is showing that if some nonroot vertex a is an articulation point, then there is no back edge between any proper descendent of a and a proper ancestor of a in the depth-first spanning tree. We prove this by a contradiction.

Let u and v be two vertices such that every path from u to v goes through a. At least one of u and v is a proper descendent of a, since otherwise there is a path from u to v that avoids a. Assume without loss of generality that u is a proper descendent of a. Let c be the child of a that contains u as a descendent. If there is no back edge between a descendent of c and a proper ancestor of a, then the theorem is true immediately, so suppose for the sake of contradiction that there is a back edge (s, t). Then either v is a proper descendent of a or it isn't. In the second case, by taking a path from u to s to t to v, we can avoid a, which is a contradiction. In the first case, clearly v cannot be a descendent of c, so let c' be the child of a that contains v as a descendent. By a similar argument as before, the only possibility is that there is a back edge (s', t') between a descendent of c' and a proper ancestor of a. Then there is a path from u to s to t to t' to s' to v; this path avoids a, which is also a contradiction.

9.23 (a) Do a depth-first search and count the number of back edges.

(b) This is the feedback edge set problem. See reference [1] or [20].

9.24 Let (v,w) be a cross edge. Since at the time w is examined it is already marked, and w is not a descendent of v (else it would be a forward edge), processing for w is already complete when processing for v commences. Thus under the convention that trees (and subtrees) are placed left to right, the cross edge goes from right to left.

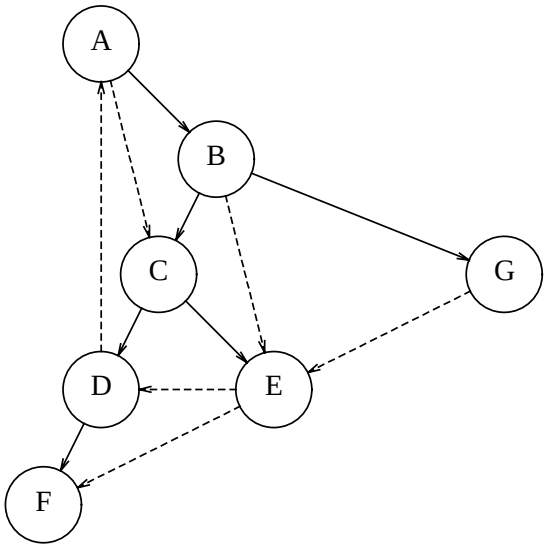
9.25 Suppose the vertices are numbered in preorder and postorder.

If (v,w) is a tree edge, then v must have a smaller preorder number than w. It is easy to see that the converse is true.

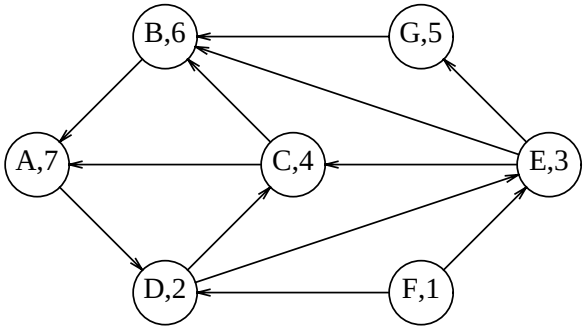
If (v,w) is a cross edge, then v must have both a larger preorder and postorder number than w. The converse is shown as follows: because v has a larger preorder number, w cannot be a descendent of v; because it has a larger postorder number, v cannot be a descendent of w; thus they must be in different trees.

Otherwise, v has a larger preorder number but is not a cross edge. To test if (v,w) is a back edge, keep a stack of vertices that are active in the depth-first search call (that is, a stack of vertices on the path from the current root). By keeping a bit array indicating presence on the stack, we can easily decide if (v,w) is a back edge or a forward edge.

9.26 The first depth-first spanning tree is



G_r, with the order in which to perform the second depth-first search, is shown next. The strongly connected components are {F} and all other vertices.



9.28 This is the algorithm mentioned in the references.

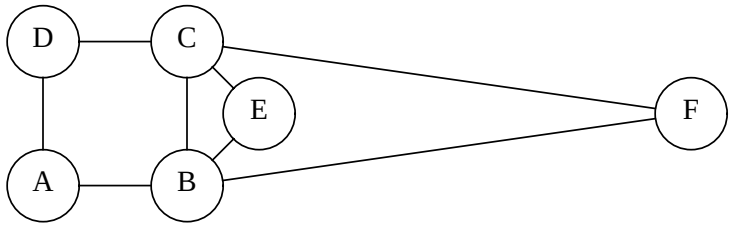
9.29 As an edge (v,w) is implicitly processed, it is placed on a stack. If v is determined to be an articulation point because $Low[w] \geq Num[v]$, then the stack is popped until edge (v,w) is removed: The set of popped edges is a biconnected component. An edge (v,w) is not placed on the stack if the edge (w,v) was already processed as a back edge.

9.30 Let (u,v) be an edge of the breadth-first spanning tree. (u,v) are connected, thus they must be in the same tree. Let the root of the tree be r; if the shortest path from r to u is d_u, then u is at level d_u; likewise, v is at level d_v. If (u,v) were a back edge, then $d_u > d_v$, and v is visited before u. But if there were an edge between u and v, and v is visited first, then there would be a tree edge (v,u), and not a back edge (u,v). Likewise, if (u,v) were a forward edge, then there is some w, distinct from u and v, on the path from u to v; this contradicts the fact that $d_v = d_w + 1$. Thus only tree edges and cross edges are possible.

9.31 Perform a depth-first search. The return from each recursive call implies the edge traversal in the opposite direction. The time is clearly linear.

9.33 If there is an Euler circuit, then it consists of entering and exiting nodes; the number of entrances clearly must equal the number of exits. If the graph is not strongly connected, there cannot be a cycle connecting all the vertices. To prove the converse, an algorithm similar in spirit to the undirected version can be used.

9.34 Neither of the proposed algorithms works. For example, as shown, a depth-first search of a biconnected graph that follows A, B, C, D is forced back to A, where it is stranded.



9.35 These are classic graph theory results. Consult any graph theory for a solution to this exercise.

9.36 All the algorithms work without modification for multigraphs.

9.37 Obviously, G must be connected. If each edge of G can be converted to a directed edge and produce a strongly connected graph G', then G is *convertible.*

Then, if the removal of a single edge disconnects G, G is not convertible since this would also disconnect G'. This is easy to test by checking to see if there are any single-edge biconnected components.

Otherwise, perform a depth-first search on G and direct each tree edge away from the root and each back edge toward the root. The resulting graph is strongly connected because, for any vertex v, we can get to a higher level than v by taking some (possibly 0) tree edges and a back edge. We can apply this until we eventually get to the root, and then follow tree edges down to any other vertex.

9.38 (b) Define a graph where each stick is represented by a vertex. If stick S_i is above S_j and thus must be removed first, then place an edge from S_i to S_j. A legal pick-up ordering is given by a topological sort; if the graph has a cycle, then the sticks cannot be picked up.

9.39 Given an instance of clique, form the graph G' that is the *complement graph* of G: (v,w) is an edge in G' if and only if it is not an edge in G. Then G' has a vertex cover of at most $|V| - K$ if G has a clique of size at least K. (The vertices that form the vertex cover are exactly those not in the clique.) The details of the proof are left to the reader.

9.40 A proof can be found in Garey and Johnson [20].

9.41 Clearly, the *baseball card collector problem (BCCP)* is in NP, because it is easy to check if K packets contain all the cards. To show it is NP-complete, we reduce vertex cover to it. Let $G = (V, E)$ and K be an instance of vertex cover. For each vertex v, place all edges adjacent to v in packet P_v. The K packets will contain all edges (baseball cards) iff G can be covered by K vertices.