```
process_vertex_late(int v)
{
     bool root;          /* is the vertex the root of the DFS tree? */
     int time_v;         /* earliest reachable time for v */
     int time_parent;    /* earliest reachable time for parent[v] */

     if (parent[v] < 1) {    /* test if v is the root */
             if (tree_out_degree[v] > 1)
                     printf("root articulation vertex: %d \n",v);
             return;
     }

     root = (parent[parent[v]] < 1);      /* is parent[v] the root? */
     if ((reachable_ancestor[v] == parent[v]) && (!root))
             printf("parent articulation vertex: %d \n",parent[v]);

     if (reachable_ancestor[v] == v) {
             printf("bridge articulation vertex: %d \n",parent[v]);

             if (tree_out_degree[v] > 0)  /* test if v is not a leaf */
                     printf("bridge articulation vertex: %d \n",v);
     }

     time_v = entry_time[reachable_ancestor[v]];
     time_parent = entry_time[ reachable_ancestor[parent[v]] ];

     if (time_v < time_parent)
             reachable_ancestor[parent[v]] = reachable_ancestor[v];
}
```

The last lines of this routine govern when we back up a node's highest reachable ancestor to its parent, namely whenever it is higher than the parent's earliest ancestor to date.

We can alternately talk about reliability in terms of edge failures instead of vertex failures. Perhaps our vandal would find it easier to cut a cable instead of blowing up a switching station. A single edge whose deletion disconnects the graph is called a *bridge*; any graph without such an edge is said to be *edge-biconnected*.

Identifying whether a given edge (x, y) is a bridge is easily done in linear time by deleting the edge and testing whether the resulting graph is connected. In fact all bridges can be identified in the same $O(n+m)$ time. Edge (x, y) is a bridge if (1) it is a tree edge, and (2) no back edge connects from y or below to x or above. This can be computed with a minor modification of the `reachable_ancestor` function.

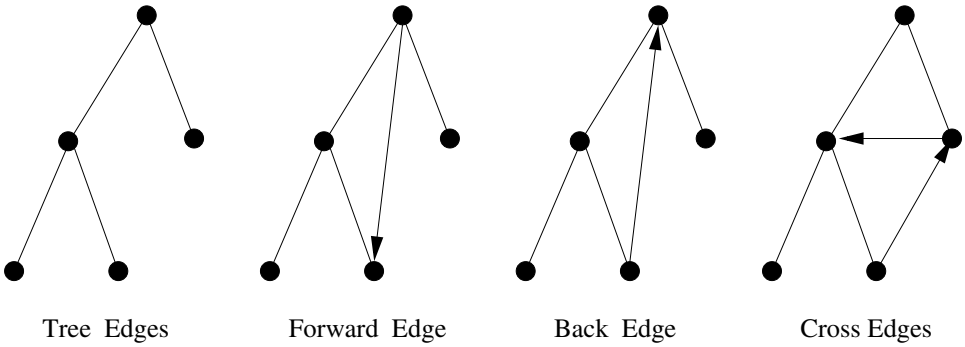


Figure 5.14: Possible edge cases for BFS/DFS traversal

5.10 Depth-First Search on Directed Graphs

Depth-first search on an undirected graph proves useful because it organizes the edges of the graph in a very precise way, as shown in Figure 5.10.

When traversing undirected graphs, every edge is either in the depth-first search tree or a back edge to an ancestor in the tree. Let us review why. Suppose we encountered a "forward edge" (x, y) directed toward a descendant vertex. In this case, we would have discovered (x, y) while exploring y, making it a back edge. Suppose we encounter a "cross edge" (x, y), linking two unrelated vertices. Again, we would have discovered this edge when we explored y, making it a tree edge.

For directed graphs, depth-first search labelings can take on a wider range of possibilities. Indeed, all four of the edge cases in Figure 5.14 can occur in traversing directed graphs. Still, this classification proves useful in organizing algorithms on directed graphs. We typically take a different action on edges from each different case.

The correct labeling of each edge can be readily determined from the state, discovery time, and parent of each vertex, as encoded in the following function:

```
int edge_classification(int x, int y)
{
 if (parent[y] == x) return(TREE);
 if (discovered[y] && !processed[y]) return(BACK);
 if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);
 if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);

 printf("Warning: unclassified edge (%d,%d)\n",x,y);
}
```

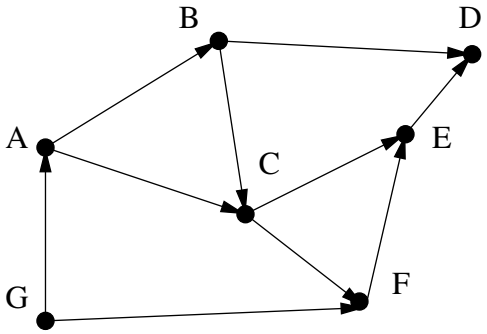


Figure 5.15: A DAG with only one topological sort (G, A, B, C, F, E, D)

As with BFS, this implementation of the depth-first search algorithm includes places to optionally process each vertex and edge—say to copy them, print them, or count them. Both algorithms will traverse all edges in the same connected component as the starting point. Since we need to start with a vertex in each component to traverse a disconnected graph, we must start from any vertex remaining undiscovered after a component search. With the proper initialization, this completes the traversal algorithm:

```
DFS-graph(G)
    for each vertex u ∈ V[G] do
        state[u] = "undiscovered"
            for each vertex u ∈ V[G] do
        if state[u] = "undiscovered" then
            initialize new component, if desired
            DFS(G, u)
```

I encourage the reader to convince themselves of the correctness of these four conditions. What I said earlier about the subtlety of depth-first search goes double for directed graphs.

5.10.1 Topological Sorting

Topological sorting is the most important operation on directed acyclic graphs (DAGs). It orders the vertices on a line such that all directed edges go from left to right. Such an ordering cannot exist if the graph contains a directed cycle, because there is no way you can keep going right on a line and still return back to where you started from!

Each DAG has at least one topological sort. The importance of topological sorting is that it gives us an ordering to process each vertex before any of its successors. Suppose the edges represented precedence constraints, such that edge

(x, y) means job x must be done before job y. Then, any topological sort defines a legal schedule. Indeed, there can be many such orderings for a given DAG.

But the applications go deeper. Suppose we seek the shortest (or longest) path from x to y in a DAG. No vertex appearing after y in the topological order can contribute to any such path, because there will be no way to get back to y. We can appropriately process all the vertices from left to right in topological order, considering the impact of their outgoing edges, and know that we will have looked at everything we need before we need it. Topological sorting proves very useful in essentially any algorithmic problem on directed graphs, as discussed in the catalog in Section 15.2 (page 481).

Topological sorting can be performed efficiently using depth-first searching. A directed graph is a DAG if and only if no back edges are encountered. Labeling the vertices in the reverse order that they are marked *processed* finds a topological sort of a DAG. Why? Consider what happens to each directed edge $\{x, y\}$ as we encounter it exploring vertex x:

- If y is currently *undiscovered*, then we start a DFS of y before we can continue with x. Thus y is marked *completed* before x is, and x appears before y in the topological order, as it must.

- If y is *discovered* but not *completed*, then $\{x, y\}$ is a back edge, which is forbidden in a DAG.

- If y is *processed*, then it will have been so labeled before x. Therefore, x appears before y in the topological order, as it must.

Study the following implementation:

```
process_vertex_late(int v)
{
        push(&sorted,v);
}


process_edge(int x, int y)
{
        int class;              /* edge class */

        class = edge_classification(x,y);

        if (class == BACK)
               printf("Warning: directed cycle found, not a DAG\n");
}
```

```
topsort(graph *g)
{
        int i;                          /* counter */

        init_stack(&sorted);

        for (i=1; i<=g->nvertices; i++)
                if (discovered[i] == FALSE)
                        dfs(g,i);

        print_stack(&sorted);           /* report topological order */
}
```

We push each vertex on a stack as soon as we have evaluated all outgoing edges. The top vertex on the stack always has no incoming edges from any vertex on the stack. Repeatedly popping them off yields a topological ordering.

5.10.2 Strongly Connected Components

We are often concerned with *strongly connected components*—that is, partitioning a graph into chunks such that directed paths exist between all pairs of vertices within a given chunk. A directed graph is *strongly connected* if there is a directed path between any two vertices. Road networks should be strongly connected, or else there will be places you can drive to but not drive home from without violating one-way signs.

It is straightforward to use graph traversal to test whether a graph $G = (V, E)$ is strongly connected in linear time. First, do a traversal from some arbitrary vertex v. Every vertex in the graph had better be reachable from v (and hence discovered on the BFS or DFS starting from v), otherwise G cannot possibly be strongly connected. Now construct a graph $G' = (V, E')$ with the same vertex and edge set as G but with all edges reversed—i.e., directed edge $(x, y) \in E$ iff $(y, x) \in E'$. Thus, any path from v to z in G' corresponds to a path from z to v in G. By doing a DFS from v in G', we find all vertices with paths *to* v in G. The graph is strongly connected iff all vertices in G can (1) reach v and (2) are reachable from v.

Graphs that are not strongly connected can be partitioned into strongly connected components, as shown in Figure 5.16 (left). The set of such components and the weakly-connecting edges that link them together can be determined using DFS. The algorithm is based on the observation that it is easy to find a directed cycle using a depth-first search, since any back edge plus the down path in the DFS tree gives such a cycle. All vertices in this cycle must be in the same strongly connected component. Thus, we can shrink (contract) the vertices on this cycle down to a single vertex representing the component, and then repeat. This process terminates when no directed cycle remains, and each vertex represents a different strongly connected component.

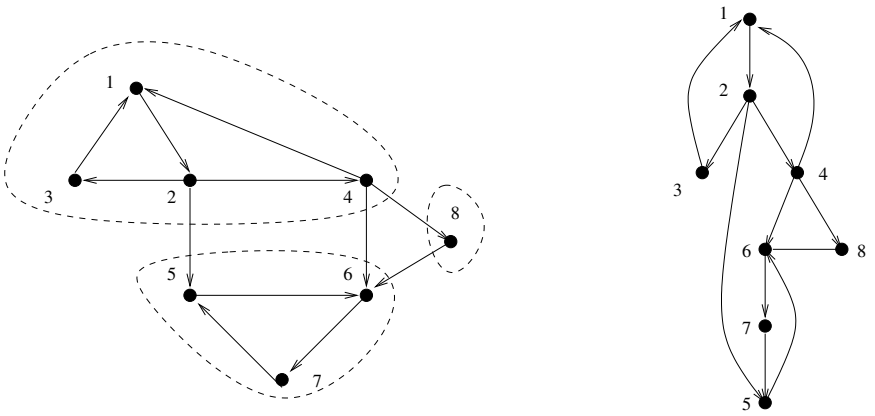


Figure 5.16: The strongly-connected components of a graph, with the associated DFS tree

Our approach to implementing this idea is reminiscent of finding biconnected components in Section 5.9.2 (page 173). We update our notion of the oldest reachable vertex in response to (1) nontree edges and (2) backing up from a vertex. Because we are working on a directed graph, we also must contend with forward edges (from a vertex to a descendant) and cross edges (from a vertex back to an nonancestor but previously discovered vertex). Our algorithm will peal one strong component off the tree at a time, and assign each of its vertices the number of the component it is in:

```
strong_components(graph *g)
{
        int i;                          /* counter */

        for (i=1; i<=(g->nvertices); i++) {
                low[i] = i;
                scc[i] = -1;
        }
        components_found = 0;
        init_stack(&active);
        initialize_search(&g);

        for (i=1; i<=(g->nvertices); i++)
                if (discovered[i] == FALSE) {
                        dfs(g,i);
                }
}
```

Define `low[v]` to be the oldest vertex known to be in the same strongly connected component as v. This vertex is not necessarily an ancestor, but may also be a distant cousin of v because of cross edges. Cross edges that point vertices from *previous* strongly connected components of the graph cannot help us, because there can be no way back from them to v, but otherwise cross edges are fair game. Forward edges have no impact on reachability over the depth-first tree edges, and hence can be disregarded:

```
int low[MAXV+1];        /* oldest vertex surely in component of v */
int scc[MAXV+1];        /* strong component number for each vertex */


process_edge(int x, int y)
{
        int class;           /* edge class */

        class = edge_classification(x,y);

        if (class == BACK) {
                if (entry_time[y] < entry_time[ low[x] ] )
                        low[x] = y;
        }

        if (class == CROSS) {
                if (scc[y] == -1)  /* component not yet assigned */
                        if (entry_time[y] < entry_time[ low[x] ] )
                                low[x] = y;
        }
}
```

A new strongly connected component is found whenever the lowest reachable vertex from v is v. If so, we can clear the stack of this component. Otherwise, we give our parent the benefit of the oldest ancestor we can reach and backtrack:

```
process_vertex_early(int v)
{
        push(&active,v);
}
```

```
process_vertex_late(int v)
{
      if (low[v] == v) {     /* edge (parent[v],v) cuts off scc */
              pop_component(v);
      }

      if (entry_time[low[v]] < entry_time[low[parent[v]]])
              low[parent[v]] = low[v];
}


pop_component(int v)
{
      int t;                /* vertex placeholder */

      components_found = components_found + 1;

      scc[ v ] = components_found;
      while ((t = pop(&active)) != v) {
              scc[ t ] = components_found;
      }
}
```

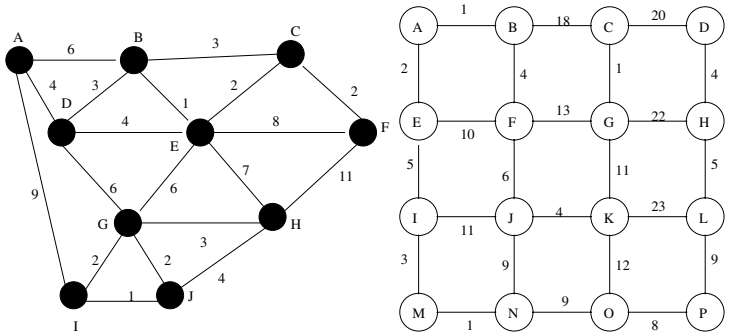
Chapter Notes

Our treatment of graph traversal represents an expanded version of material from Chapter 9 of [SR03]. The *Combinatorica* graph library discussed in the war story is best described in the old [Ski90]. and new editions [PS03] of the associated book. Accessible introductions to the science of social networks include Barabasi [Bar03] and Watts [Wat04].

5.11 Exercises

Simulating Graph Algorithms

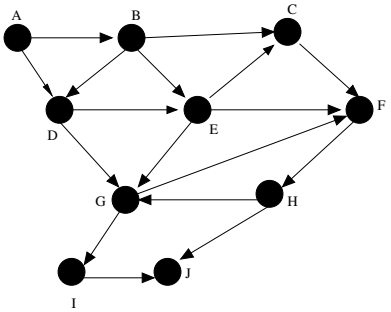
5-1. *[3]* For the following graphs G_1 (left) and G_2 (right):



(a) Report the order of the vertices encountered on a breadth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order (i.e., A before Z).

(b) Report the order of the vertices encountered on a depth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order (i.e., A before Z).

5-2. *[3]* Do a topological sort of the following graph G:



Traversal

5-3. *[3]* Prove by induction that there is a unique path between any pair of vertices in a tree.

5-4. *[3]* Prove that in a breadth-first search on a undirected graph G, every edge is either a tree edge or a cross edge, where x is neither an ancestor nor descendant of y, in cross edge (x, y).

5-5. *[3]* Give a linear algorithm to compute the chromatic number of graphs where each vertex has degree at most 2. Must such graphs be bipartite?

5-6. *[5]* In breadth-first and depth-first search, an undiscovered node is marked *discovered* when it is first encountered, and marked *processed* when it has been completely

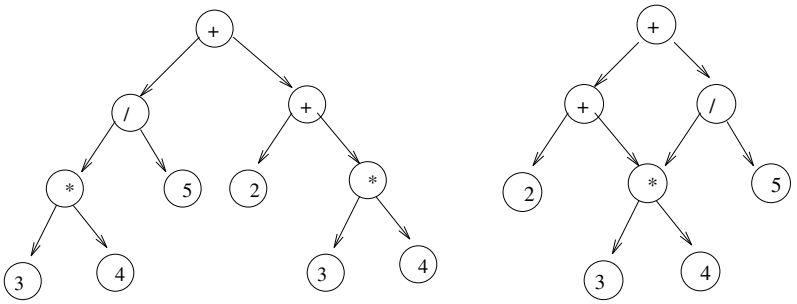


Figure 5.17: Expression $2 + 3 * 4 + (3 * 4)/5$ as a tree and a DAG

searched. At any given moment, several nodes might be simultaneously in the *discovered* state.

(a) Describe a graph on n vertices and a particular starting vertex v such that $\Theta(n)$ nodes are simultaneously in the *discovered* state during a *breadth-first search* starting from v.

(b) Describe a graph on n vertices and a particular starting vertex v such that $\Theta(n)$ nodes are simultaneously in the *discovered* state during a *depth-first search* starting from v.

(c) Describe a graph on n vertices and a particular starting vertex v such that at some point $\Theta(n)$ nodes remain *undiscovered*, while $\Theta(n)$ nodes have been *processed* during a *depth-first search* starting from v. (Note, there may also be *discovered* nodes.)

5-7. *[4]* Given pre-order and in-order traversals of a binary tree, is it possible to reconstruct the tree? If so, sketch an algorithm to do it. If not, give a counterexample. Repeat the problem if you are given the pre-order and post-order traversals.

5-8. *[3]* Present correct and efficient algorithms to convert an undirected graph G between the following graph data structures. You must give the time complexity of each algorithm, assuming n vertices and m edges.

(a) Convert from an adjacency matrix to adjacency lists.

(b) Convert from an adjacency list to an incidence matrix. An incidence matrix M has a row for each vertex and a column for each edge, such that $M[i, j] = 1$ if vertex i is part of edge j, otherwise $M[i, j] = 0$.

(c) Convert from an incidence matrix to adjacency lists.

5-9. *[3]* Suppose an arithmetic expression is given as a tree. Each leaf is an integer and each internal node is one of the standard arithmetical operations $(+, -, *, /)$. For example, the expression $2 + 3 * 4 + (3 * 4)/5$ is represented by the tree in Figure 5.17(a). Give an $O(n)$ algorithm for evaluating such an expression, where there are n nodes in the tree.

5-10. *[5]* Suppose an arithmetic expression is given as a DAG (directed acyclic graph) with common subexpressions removed. Each leaf is an integer and each internal

node is one of the standard arithmetical operations $(+, -, *, /)$. For example, the expression $2 + 3 * 4 + (3 * 4)/5$ is represented by the DAG in Figure 5.17(b). Give an $O(n + m)$ algorithm for evaluating such a DAG, where there are n nodes and m edges in the DAG. Hint: modify an algorithm for the tree case to achieve the desired efficiency.

5-11. *[8]* The war story of Section 5.4 (page 158) describes an algorithm for constructing the dual graph of the triangulation efficiently, although it does not guarantee linear time. Give a worst-case linear algorithm for the problem.

Algorithm Design

5-12. *[5]* The *square* of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ iff there exists $v \in V$ such that $(u, v) \in E$ and $(v, w) \in E$; i.e., there is a path of exactly two edges from u to w.

Give efficient algorithms for both adjacency lists and matrices.

5-13. *[5]* A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices V' such that each edge in E is incident on at least one vertex of V'.

(a) Give an efficient algorithm to find a minimum-size vertex cover if G is a tree.

(b) Let $G = (V, E)$ be a tree such that the weight of each vertex is equal to the degree of that vertex. Give an efficient algorithm to find a minimum-weight vertex cover of G.

(c) Let $G = (V, E)$ be a tree with arbitrary weights associated with the vertices. Give an efficient algorithm to find a minimum-weight vertex cover of G.

5-14. *[3]* A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $V' \in V$ such that every edge in E contains at least one vertex from V'. Delete all the leaves from any depth-first search tree of G. Must the remaining vertices form a vertex cover of G? Give a proof or a counterexample.

5-15. *[5]* A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $V' \in V$ such that every edge in E contains *at least one* vertex from V'. An *independent set* of graph $G = (V, E)$ is a subset of vertices $V' \in V$ such that no edge in E contains both vertices from V'.

An *independent vertex cover* is a subset of vertices that is both an independent set and a vertex cover of G. Give an efficient algorithm for testing whether G contains an independent vertex cover. What classical graph problem does this reduce to?

5-16. *[5]* An *independent set* of an undirected graph $G = (V, E)$ is a set of vertices U such that no edge in E is incident on two vertices of U.

(a) Give an efficient algorithm to find a maximum-size independent set if G is a tree.

(b) Let $G = (V, E)$ be a tree with weights associated with the vertices such that the weight of each vertex is equal to the degree of that vertex. Give an efficient algorithm to find a maximum independent set of G.

(c) Let $G = (V, E)$ be a tree with arbitrary weights associated with the vertices. Give an efficient algorithm to find a maximum independent set of G.

5-17. *[5]* Consider the problem of determining whether a given undirected graph $G = (V, E)$ contains a *triangle* or cycle of length 3.

(a) Give an $O(|V|^3)$ to find a triangle if one exists.

(b) Improve your algorithm to run in time $O(|V| \cdot |E|)$. You may assume $|V| \leq |E|$.

Observe that these bounds gives you time to convert between the adjacency matrix and adjacency list representations of G.

5-18. *[5]* Consider a set of movies $M_1, M_2, \ldots, M_k$. There is a set of customers, each one of which indicates the two movies they would like to see this weekend. Movies are shown on Saturday evening and Sunday evening. Multiple movies may be screened at the same time.

You must decide which movies should be televised on Saturday and which on Sunday, so that every customer gets to see the two movies they desire. Is there a schedule where each movie is shown at most once? Design an efficient algorithm to find such a schedule if one exists.

5-19. *[5]* The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v)$$

(where $\delta(u, v)$ is the number of edges on the path from u to v). Describe an efficient algorithm to compute the diameter of a tree, and show the correctness and analyze the running time of your algorithm.

5-20. *[5]* Given an undirected graph G with n vertices and m edges, and an integer k, give an $O(m + n)$ algorithm that finds the maximum induced subgraph H of G such that each vertex in H has degree $\geq k$, or prove that no such graph exists. An induced subgraph $F = (U, R)$ of a graph $G = (V, E)$ is a subset of U of the vertices V of G, and all edges R of G such that both vertices of each edge are in U.

5-21. *[6]* Let v and w be two vertices in a directed graph $G = (V, E)$. Design a linear-time algorithm to find the *number* of different shortest paths (not necessarily vertex disjoint) between v and w. Note: the edges in G are unweighted.

5-22. *[6]* Design a linear-time algorithm to eliminate each vertex v of degree 2 from a graph by replacing edges (u, v) and (v, w) by an edge (u, w). We also seek to eliminate multiple copies of edges by replacing them with a single edge. Note that removing multiple copies of an edge may create a new vertex of degree 2, which has to be removed, and that removing a vertex of degree 2 may create multiple edges, which also must be removed.

Directed Graphs

5-23. *[5]* Your job is to arrange n ill-behaved children in a straight line, facing front. You are given a list of m statements of the form "i hates j". If i hates j, then you do not want put i somewhere behind j, because then i is capable of throwing something at j.

(a) Give an algorithm that orders the line, (or says that it is not possible) in $O(m + n)$ time.

(b) Suppose instead you want to arrange the children in rows such that if i hates j, then i must be in a lower numbered row than j. Give an efficient algorithm to find the minimum number of rows needed, if it is possible.

5-24. *[3]* Adding a single directed edge to a directed graph can reduce the number of weakly connected components, but by at most how many components? What about the number of strongly connected components?

5-25. *[5]* An *arborescence* of a directed graph G is a rooted tree such that there is a directed path from the root to every other vertex in the graph. Give an efficient and correct algorithm to test whether G contains an arborescence, and its time complexity.

5-26. *[5]* A *mother* vertex in a directed graph $G = (V, E)$ is a vertex v such that all other vertices G can be reached by a directed path from v.

(a) Give an $O(n + m)$ algorithm to test whether a given vertex v is a mother of G, where $n = |V|$ and $m = |E|$.

(b) Give an $O(n+m)$ algorithm to test whether graph G contains a mother vertex.

5-27. *[9]* A *tournament* is a directed graph formed by taking the complete undirected graph and assigning arbitrary directions on the edges—i.e., a graph $G = (V, E)$ such that for all u,$v \in V$, exactly one of (u, v) or (v, u) is in E. Show that every tournament has a Hamiltonian path—that is, a path that visits every vertex exactly once. Give an algorithm to find this path.

Articulation Vertices

5-28. *[5]* An articulation vertex of a graph G is a vertex whose deletion disconnects G. Let G be a graph with n vertices and m edges. Give a simple $O(n + m)$ algorithm for finding a vertex of G that is *not* an articulation vertex—i.e. , whose deletion does not disconnect G.

5-29. *[5]* Following up on the previous problem, give an $O(n + m)$ algorithm that finds a deletion order for the n vertices such that no deletion disconnects the graph. (Hint: think DFS/BFS.)

5-30. *[3]* Suppose G is a connected undirected graph. An edge e whose removal disconnects the graph is called a *bridge*. Must every bridge e be an edge in a depth-first search tree of G? Give a proof or a counterexample.

Interview Problems

5-31. *[3]* Which data structures are used in depth-first and breath-first search?

5-32. *[4]* Write a function to traverse binary search tree and return the ith node in sorted order.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

5-1. "Bicoloring" – Programming Challenges 110901, UVA Judge 10004.

5-2. "Playing with Wheels" – Programming Challenges 110902, UVA Judge 10067.

5-3. "The Tourist Guide" – Programming Challenges 110903, UVA Judge 10099.

5-4. "Edit Step Ladders" – Programming Challenges 110905, UVA Judge 10029.

5-5. "Tower of Cubes" – Programming Challenges 110906, UVA Judge 10051.

6

Weighted Graph Algorithms

The data structures and traversal algorithms of Chapter 5 provide the basic building blocks for any computation on graphs. However, all the algorithms presented there dealt with unweighted graphs—i.e. , graphs where each edge has identical value or weight.

There is an alternate universe of problems for *weighted graphs*. The edges of road networks are naturally bound to numerical values such as construction cost, traversal time, length, or speed limit. Identifying the shortest path in such graphs proves more complicated than breadth-first search in unweighted graphs, but opens the door to a wide range of applications.

The graph data structure from Chapter 5 quietly supported edge-weighted graphs, but here we make this explicit. Our adjacency list structure consists of an array of linked lists, such that the outgoing edges from vertex x appear in the list `edges[x]`:

```
typedef struct {
        edgenode *edges[MAXV+1];   /* adjacency info */
        int degree[MAXV+1];        /* outdegree of each vertex */
        int nvertices;             /* number of vertices in graph */
        int nedges;                /* number of edges in graph */
        int directed;              /* is the graph directed? */
} graph;
```

Each `edgenode` is a record containing three fields, the first describing the second endpoint of the edge (`y`), the second enabling us to annotate the edge with a weight (`weight`), and the third pointing to the next edge in the list (`next`):

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_6,

```
typedef struct {
        int y;                          /* adjacency info */
        int weight;                     /* edge weight, if any */
        struct edgenode *next;          /* next edge in list */
} edgenode;
```

We now describe several sophisticated algorithms using this data structure, including minimum spanning trees, shortest paths, and maximum flows. That these optimization problems can be solved efficiently is quite worthy of our respect. Recall that no such algorithm exists for the first weighted graph problem we encountered, namely the traveling salesman problem.

6.1 Minimum Spanning Trees

A *spanning tree* of a graph $G = (V, E)$ is a subset of edges from E forming a tree connecting all vertices of V. For edge-weighted graphs, we are particularly interested in the *minimum spanning tree*—the spanning tree whose sum of edge weights is as small as possible.

Minimum spanning trees are the answer whenever we need to connect a set of points (representing cities, homes, junctions, or other locations) by the smallest amount of roadway, wire, or pipe. Any tree is the smallest possible connected graph in terms of number of edges, while the minimum spanning tree is the smallest connected graph in terms of edge weight. In geometric problems, the point set $p_1, \ldots, p_n$ defines a complete graph, with edge (v_i, v_j) assigned a weight equal to the distance from p_i to p_j. An example of a geometric minimum spanning tree is illustrated in Figure 6.1. Additional applications of minimum spanning trees are discussed in Section 15.3 (page 484).

A minimum spanning tree minimizes the total length over all possible spanning trees. However, there can be more than one minimum spanning tree in a graph. Indeed, all spanning trees of an unweighted (or equally weighted) graph G are minimum spanning trees, since each contains exactly $n - 1$ equal-weight edges. Such a spanning tree can be found using depth-first or breadth-first search. Finding a minimum spanning tree is more difficult for general weighted graphs, however two different algorithms are presented below. Both demonstrate the optimality of certain greedy heuristics.

6.1.1 Prim's Algorithm

Prim's minimum spanning tree algorithm starts from one vertex and grows the rest of the tree one edge at a time until all vertices are included.

Greedy algorithms make the decision of what to do next by selecting the best local option from all available choices without regard to the global structure. Since we seek the tree of minimum weight, the natural greedy algorithm for minimum

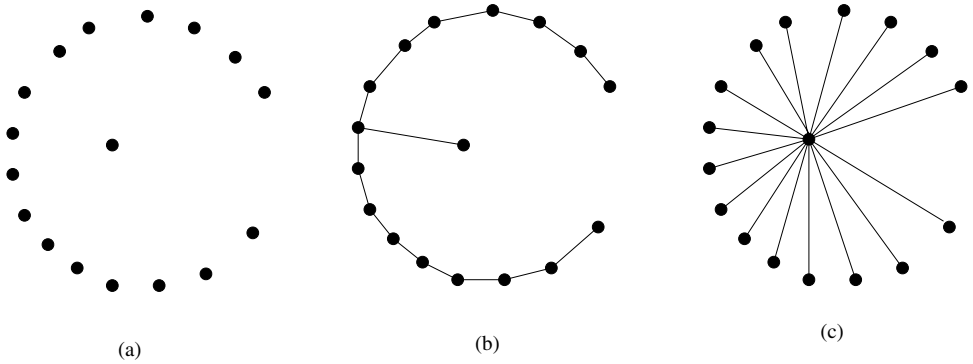


Figure 6.1: (a) Two spanning trees of point set; (b) the minimum spanning tree, and (c) the shortest path from center tree

---

spanning tree repeatedly selects the smallest weight edge that will enlarge the number of vertices in the tree.

Prim-MST(G)
    Select an arbitrary vertex s to start the tree from.
    While (there are still nontree vertices)
        Select the edge of minimum weight between a tree and nontree vertex
        Add the selected edge and vertex to the tree T_{prim}.

Prim's algorithm clearly creates a spanning tree, because no cycle can be introduced by adding edges between tree and nontree vertices. However, why should it be of minimum weight over all spanning trees? We have seen ample evidence of other natural greedy heuristics that do not yield a global optimum. Therefore, we must be particularly careful to demonstrate any such claim.

We use proof by contradiction. Suppose that there existed a graph G for which Prim's algorithm did not return a minimum spanning tree. Since we are building the tree incrementally, this means that there must have been some particular instant where we went wrong. Before we inserted edge (x, y), T_{prim} consisted of a set of edges that was a subtree of some minimum spanning tree T_{min}, but choosing edge (x, y) fatally took us away from a minimum spanning tree (see Figure 6.2(a)).

But how could we have gone wrong? There must be a path p from x to y in T_{min}, as shown in Figure 6.2(b). This path must use an edge (v_1, v_2), where v_1 is in T_{prim}, but v_2 is not. This edge (v_1, v_2) must have weight at least that of (x, y), or Prim's algorithm would have selected it before (x, y) when it had the chance. Inserting (x, y) and deleting (v_1, v_2) from T_{min} leaves a spanning tree no larger than before, meaning that Prim's algorithm did not make a fatal mistake in selecting edge (x, y). Therefore, by contradiction, Prim's algorithm must construct a minimum spanning tree.

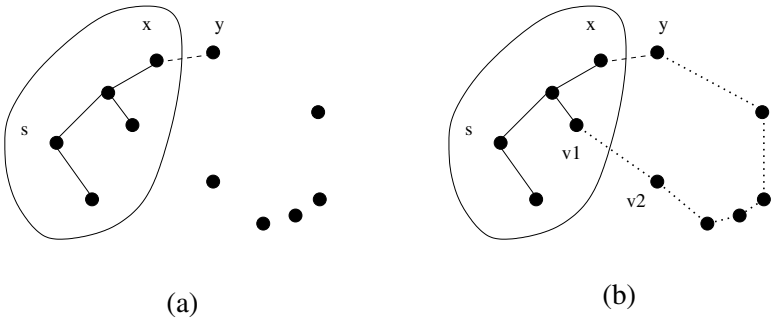


Figure 6.2: Where Prim's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

Implementation

Prim's algorithm grows the minimum spanning tree in stages, starting from a given vertex. At each iteration, we add one new vertex into the spanning tree. A greedy algorithm suffices for correctness: we always add the lowest-weight edge linking a vertex in the tree to a vertex on the outside. The simplest implementation of this idea would assign each vertex a Boolean variable denoting whether it is already in the tree (the array `intree` in the code below), and then searches all edges at each iteration to find the minimum weight edge with exactly one `intree` vertex.

Our implementation is somewhat smarter. It keeps track of the cheapest edge linking every nontree vertex in the tree. The cheapest such edge over all remaining non-tree vertices gets added in each iteration. We must update the costs of getting to the non-tree vertices after each insertion. However, since the most recently inserted vertex is the only change in the tree, all possible edge-weight updates must come from its outgoing edges:

```
prim(graph *g, int start)
{
    int i;                      /* counter */
    edgenode *p;                /* temporary pointer */
    bool intree[MAXV+1];        /* is the vertex in the tree yet? */
    int distance[MAXV+1];       /* cost of adding to tree */
    int v;                      /* current vertex to process */
    int w;                      /* candidate next vertex */
    int weight;                 /* edge weight */
    int dist;                   /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
            intree[i] = FALSE;
```

```
            distance[i] = MAXINT;
            parent[i] = -1;
        }

    distance[start] = 0;
    v = start;

    while (intree[v] == FALSE) {
           intree[v] = TRUE;
           p = g->edges[v];
           while (p != NULL) {
                  w = p->y;
                  weight = p->weight;
                  if ((distance[w] > weight) && (intree[w] == FALSE)) {
                         distance[w] = weight;
                         parent[w] = v;
                    }
                    p = p->next;
            }

            v = 1;
            dist = MAXINT;
            for (i=1; i<=g->nvertices; i++)
                    if ((intree[i] == FALSE) && (dist > distance[i])) {
                            dist = distance[i];
                            v = i;
                    }
    }
}
```

Analysis

Prim's algorithm is correct, but how efficient is it? This depends on which data structures are used to implement it. In the pseudocode, Prim's algorithm makes n iterations sweeping through all m edges on each iteration—yielding an $O(mn)$ algorithm.

But our implementation avoids the need to test all m edges on each pass. It only considers the $\leq n$ cheapest known edges represented in the `parent` array and the $\leq n$ edges out of new tree vertex v to update `parent`. By maintaining a Boolean flag along with each vertex to denote whether it is in the tree or not, we test whether the current edge joins a tree with a non-tree vertex in constant time.

The result is an $O(n^2)$ implementation of Prim's algorithm, and a good illustration of power of data structures to speed up algorithms. In fact, more sophisticated

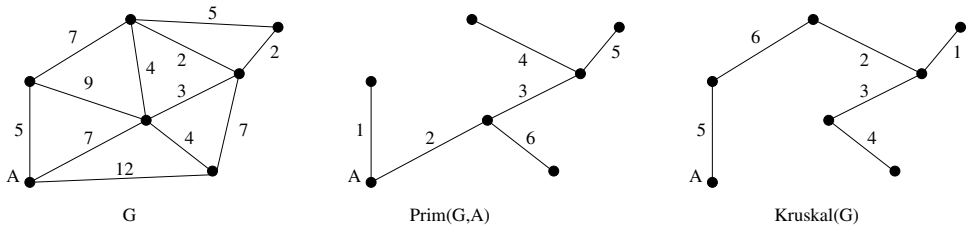


Figure 6.3: A graph G (l) with minimum spanning trees produced by Prim's (m) and Kruskal's (r) algorithms. The numbers on the trees denote the order of insertion; ties are broken arbitrarily

priority-queue data structures lead to an $O(m+n \lg n)$ implementation, by making it faster to find the minimum cost edge to expand the tree at each iteration.

The minimum spanning tree itself or its cost can be reconstructed in two different ways. The simplest method would be to augment this procedure with statements that print the edges as they are found or totals the weight of all selected edges. Alternately, the tree topology is encoded by the `parent` array, so it plus the original graph describe everything about the minimum spanning tree.

6.1.2 Kruskal's Algorithm

Kruskal's algorithm is an alternate approach to finding minimum spanning trees that proves more efficient on sparse graphs. Like Prim's, Kruskal's algorithm is greedy. Unlike Prim's, it does not start with a particular vertex.

Kruskal's algorithm builds up connected components of vertices, culminating in the minimum spanning tree. Initially, each vertex forms its own separate component in the tree-to-be. The algorithm repeatedly considers the lightest remaining edge and tests whether its two endpoints lie within the same connected component. If so, this edge will be discarded, because adding it would create a cycle in the tree-to-be. If the endpoints are in different components, we insert the edge and merge the two components into one. Since each connected component is always a tree, we need never explicitly test for cycles.

```
Kruskal-MST(G)
    Put the edges in a priority queue ordered by weight.
    count = 0
    while (count < n − 1) do
        get next edge (v, w)
        if (component (v) ≠ component(w))
            add to T_kruskal
            merge component(v) and component(w)
```

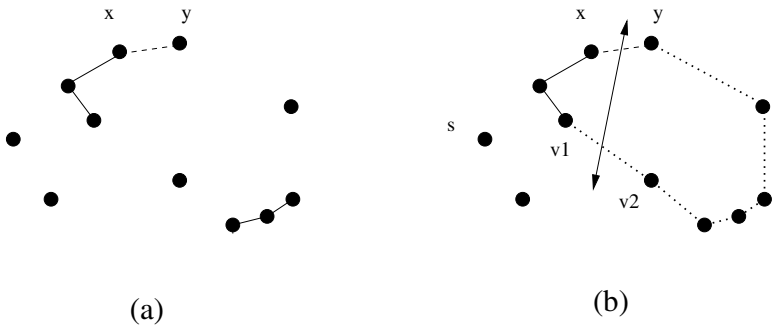


Figure 6.4: Where Kruskal's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

This algorithm adds $n-1$ edges without creating a cycle, so it clearly creates a spanning tree for any connected graph. But why must this be a *minimum* spanning tree? Suppose it wasn't. As with the correctness proof of Prim's algorithm, there must be some graph on which it fails. In particular, there must a single edge (x, y) whose insertion first prevented the tree $T_{kruskal}$ from being a minimum spanning tree T_{min}. Inserting this edge (x, y) into T_{min} will create a cycle with the path from x to y. Since x and y were in different components at the time of inserting (x, y), at least one edge (say (v_1, v_2)) on this path would have been evaluated by Kruskal's algorithm later than (x, y). But this means that $w(v_1, v_2) \geq w(x, y)$, so exchanging the two edges yields a tree of weight at most T_{min}. Therefore, we could not have made a fatal mistake in selecting (x, y), and the correctness follows.

What is the time complexity of Kruskal's algorithm? Sorting the m edges takes $O(m \lg m)$ time. The for loop makes m iterations, each testing the connectivity of two trees plus an edge. In the most simple-minded approach, this can be implemented by breadth-first or depth-first search in a sparse graph with at most n edges and n vertices, thus yielding an $O(mn)$ algorithm.

However, a faster implementation results if we can implement the component test in faster than $O(n)$ time. In fact, a clever data structure called *union-find*, can support such queries in $O(\lg n)$ time. Union-find is discussed in the next section. With this data structure, Kruskal's algorithm runs in $O(m \lg m)$ time, which is faster than Prim's for sparse graphs. Observe again the impact that the right data structure can have when implementing a straightforward algorithm.

Implementation

The implementation of the main routine follows fairly directly from the psuedocode:

```
kruskal(graph *g)
{
    int i;                      /* counter */
    set_union s;                /* set union data structure */
    edge_pair e[MAXV+1];        /* array of edges data structure */
    bool weight_compare();

    set_union_init(&s, g->nvertices);

    to_edge_array(g, e);        /* sort edges by increasing cost */
    qsort(&e,g->nedges,sizeof(edge_pair),weight_compare);

    for (i=0; i<(g->nedges); i++) {
            if (!same_component(s,e[i].x,e[i].y)) {
                    printf("edge (%d,%d) in MST\n",e[i].x,e[i].y);
                    union_sets(&s,e[i].x,e[i].y);
            }
    }
}
```

6.1.3 The Union-Find Data Structure

A *set partition* is a partitioning of the elements of some universal set (say the integers 1 to n) into a collection of disjointed subsets. Thus, each element must be in exactly one subset. Set partitions naturally arise in graph problems such as connected components (each vertex is in exactly one connected component) and vertex coloring (a person may be male or female, but not both or neither). Section 14.6 (page 456) presents algorithms for generating set partitions and related objects.

The connected components in a graph can be represented as a set partition. For Kruskal's algorithm to run efficiently, we need a data structure that efficiently supports the following operations:

- *Same component*(v_1, v_2) – Do vertices v_1 and v_2 occur in the same connected component of the current graph?
- *Merge components*(C_1, C_2) – Merge the given pair of connected components into one component in response to an edge between them.

The two obvious data structures for this task each support only one of these operations efficiently. Explicitly labeling each element with its component number enables the *same component* test to be performed in constant time, but updating the component numbers after a merger would require linear time. Alternately, we can treat the merge components operation as inserting an edge in a graph, but

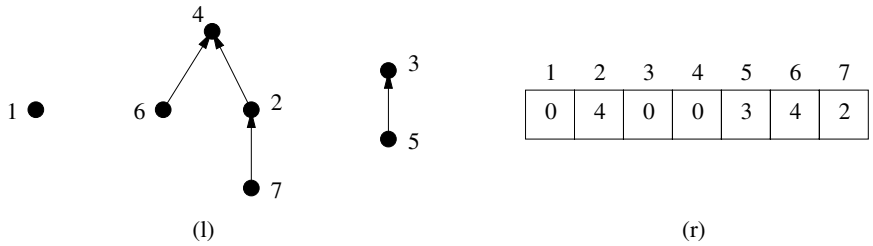


Figure 6.5: Union-find example: structure represented as trees (l) and array (r)

then we must run a full graph traversal to identify the connected components on demand.

The union-find data structure represents each subset as a "backwards" tree, with pointers from a node to its parent. Each node of this tree contains a set element, and the *name* of the set is taken from the key at the root. For reasons that will become clear, we will also maintain the number of elements in the subtree rooted in each vertex v:

```
typedef struct {
      int p[SET_SIZE+1];       /* parent element */
      int size[SET_SIZE+1];    /* number of elements in subtree i */
      int n;                   /* number of elements in set */
} set_union;
```

We implement our desired component operations in terms of two simpler operations, *union* and *find*:

- *Find(i)* – Find the root of tree containing element i, by walking up the parent pointers until there is nowhere to go. Return the label of the root.

- *Union(i,j)* – Link the root of one of the trees (say containing i) to the root of the tree containing the other (say j) so $find(i)$ now equals $find(j)$.

We seek to minimize the time it takes to execute *any* sequence of unions and finds. Tree structures can be very unbalanced, so we must limit the height of our trees. Our most obvious means of control is the decision of which of the two component roots becomes the root of the combined component on each *union*.

To minimize the tree height, it is better to make the smaller tree the subtree of the bigger one. Why? The height of all the nodes in the root subtree stay the same, while the height of the nodes merged into this tree all increase by one. Thus, merging in the smaller tree leaves the height unchanged on the larger set of vertices.

Implementation

The implementation details are as follows:

```
set_union_init(set_union *s, int n)
{
        int i;                          /* counter */

        for (i=1; i<=n; i++) {
                s->p[i] = i;
                s->size[i] = 1;
        }

        s->n = n;
}

int find(set_union *s, int x)
{
        if (s->p[x] == x)
                return(x);
        else
                return( find(s,s->p[x]) );
}

int union_sets(set_union *s, int s1, int s2)
{
        int r1, r2;                     /* roots of sets */

        r1 = find(s,s1);
        r2 = find(s,s2);

        if (r1 == r2) return;           /* already in same set */

        if (s->size[r1] >= s->size[r2]) {
                s->size[r1] = s->size[r1] + s->size[r2];
                s->p[ r2 ] = r1;
        }
        else {
                s->size[r2] = s->size[r1] + s->size[r2];
                s->p[ r1 ] = r2;
        }
}

bool same_component(set_union *s, int s1, int s2)
{
        return ( find(s,s1) == find(s,s2) );
}
```

Analysis

On each union, the tree with fewer nodes becomes the child. But how tall can such a tree get as a function of the number of nodes in it? Consider the smallest possible tree of height h. Single-node trees have height 1. The smallest tree of height-2 has two nodes; from the union of two single-node trees. When do we increase the height? Merging in single-node trees won't do it, since they just become children of the rooted tree of height-2. Only when we merge two height-2 trees together do we get a tree of height-3, now with four nodes.

See the pattern? We must double the number of nodes in the tree to get an extra unit of height. How many doublings can we do before we use up all n nodes? At most, $\lg_2 n$ doublings can be performed. Thus, we can do both unions and finds in $O(\log n)$, good enough for Kruskal's algorithm. In fact, union-find can be done even faster, as discussed in Section 12.5 (page 385).

6.1.4 Variations on Minimum Spanning Trees

This minimum spanning tree algorithm has several interesting properties that help solve several closely related problems:

- *Maximum Spanning Trees* – Suppose an evil telephone company is contracted to connect a bunch of houses together; they will be paid a price proportional to the amount of wire they install. Naturally, they will build the most expensive spanning tree possible. The *maximum spanning tree* of any graph can be found by simply negating the weights of all edges and running Prim's algorithm. The most negative tree in the negated graph is the maximum spanning tree in the original.

  Most graph algorithms do not adapt so easily to negative numbers. Indeed, shortest path algorithms have trouble with negative numbers, and certainly do *not* generate the longest possible path using this technique.

- *Minimum Product Spanning Trees* – Suppose we seek the spanning tree that minimizes the product of edge weights, assuming all edge weights are positive. Since $\lg(a \cdot b) = \lg(a) + \lg(b)$, the minimum spanning tree on a graph whose edge weights are replaced with their logarithms gives the minimum product spanning tree on the original graph.

- *Minimum Bottleneck Spanning Tree* – Sometimes we seek a spanning tree that minimizes the maximum edge weight over all such trees. In fact, every minimum spanning tree has this property. The proof follows directly from the correctness of Kruskal's algorithm.

  Such bottleneck spanning trees have interesting applications when the edge weights are interpreted as costs, capacities, or strengths. A less efficient

but conceptually simpler way to solve such problems might be to delete all "heavy" edges from the graph and ask whether the result is still connected. These kind of tests can be done with simple BFS/DFS.

The minimum spanning tree of a graph is unique if all m edge weights in the graph are distinct. Otherwise the order in which Prim's/Kruskal's algorithm breaks ties determines which minimum spanning tree is returned.

There are two important variants of a minimum spanning tree that are *not* solvable with these techniques.

- *Steiner Tree* – Suppose we want to wire a bunch of houses together, but have the freedom to add extra intermediate vertices to serve as a shared junction. This problem is known as a *minimum Steiner tree*, and is discussed in the catalog in Section 16.10.

- *Low-degree Spanning Tree* – Alternately, what if we want to find the minimum spanning tree where the highest degree node in the tree is small? The lowest max-degree tree possible would be a simple path, and have $n - 2$ nodes of degree 2 with two endpoints of degree 1. A path that visits each vertex once is called a *Hamiltonian path*, and is discussed in the catalog in Section 16.5.

6.2 War Story: Nothing but Nets

I'd been tipped off about a small printed-circuit board testing company nearby in need of some algorithmic consulting. And so I found myself inside a nondescript building in a nondescript industrial park, talking with the president of Integri-Test and one of his lead technical people.

"We're leaders in robotic printed-circuit board testing devices. Our customers have very high reliability requirements for their PC-boards. They must check that each and every board has no wire breaks *before* filling it with components. This means testing that each and every pair of points on the board that are supposed to be connected *are* connected."

"How do you do the testing?" I asked.

"We have a robot with two arms, each with electric probes. The arms simultaneously contact both of the points to test whether two points are properly connected. If they are properly connected, then the probes will complete a circuit. For each net, we hold one arm fixed at one point and move the other to cover the rest of the points."

"Wait!" I cried. "What is a net?"

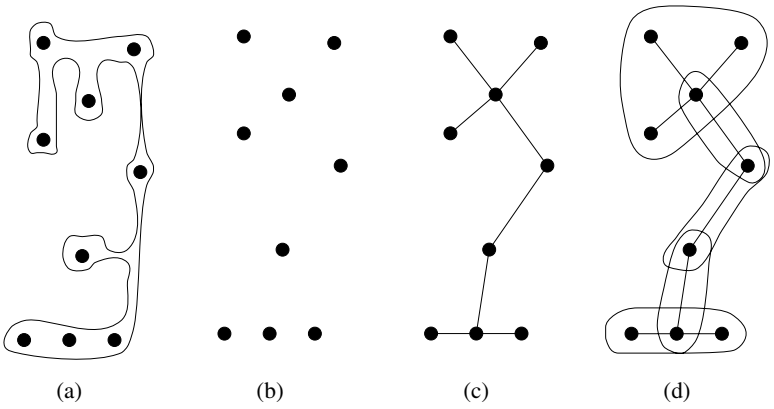


Figure 6.6: An example net showing (a) the metal connection layer, (b) the contact points, (c) their minimum spanning tree, and (d) the points partitioned into clusters

"Circuit boards are certain sets of points that are all connected together with a metal layer. This is what we mean by a net. Sometimes a net consists of two points—i.e. , an isolated wire. Sometimes a net can have 100 to 200 points, like all the connections to power or ground."

"I see. So you have a list of all the connections between pairs of points on the circuit board, and you want to trace out these wires."

He shook his head. "Not quite. The input for our testing program consists only of the net contact points, as shown in Figure 6.6(b). We don't know where the actual wires are, but we don't have to. All we must do is verify that all the points in a net are connected together. We do this by putting the left robot arm on the leftmost point in the net, and then have the right arm move around to all the other points in the net to test if they are all connected to the left point. So they must all be connected to each other."

I thought for a moment about what this meant. "OK. So your right arm has to visit all the other points in the net. How do you choose the order to visit them?"

The technical guy spoke up. "Well, we sort the points from left to right and then go in that order. Is that a good thing to do?"

"Have you ever heard of the traveling salesman problem?" I asked.

He was an electrical engineer, not a computer scientist. "No, what's that?"

"Traveling salesman is the name of the problem that you are trying to solve. Given a set of points to visit, how do you order them to minimize the travel time. Algorithms for the traveling salesman problem have been extensively studied. For small nets, you will be able to find the optimal tour by doing an exhaustive search. For big nets, there are heuristics that will get you very close to the optimal tour." I would have pointed them to Section 16.4 (page 533) if I had had this book handy.

The president scribbled down some notes and then frowned. "Fine. Maybe you can order the points in a net better for us. But that's not our real problem. When you watch our robot in action, the right arm sometimes has to run all the way to the right side of the board on a given net, while the left arm just sits there. It seems we would benefit by breaking nets into smaller pieces to balance things out."

I sat down and thought. The left and right arms each have interlocking TSP problems to solve. The left arm would move between the leftmost points of each net, while the right arm to visits all the other points in each net as ordered by the left TSP tour. By breaking each net into smaller nets we would avoid making the right arm cross all the way across the board. Further, a lot of little nets meant there would be more points in the left TSP, so each left-arm movement was likely to be short, too.

"You are right. We should win if we can break big nets into small nets. We want the nets to be small, both in the number of points and in physical area. But we must be sure that if we validate the connectivity of each small net, we will have confirmed that the big net is connected. One point in common between two little nets is sufficient to show that the bigger net formed by the two little nets is connected, since current can flow between any pair of points."

Now we had to break each net into overlapping pieces, where each piece was small. This is a clustering problem. Minimum spanning trees are often used for clustering, as discussed in Section 15.3 (page 484). In fact, that was the answer! We could find the minimum spanning tree of the net points and break it into little clusters whenever a spanning tree edge got too long. As shown in Figure 6.6(d), each cluster would share exactly one point in common with another cluster, with connectivity ensured because we are covering the edges of a spanning tree. The shape of the clusters will reflect the points in the net. If the points lay along a line across the board, the minimum spanning tree would be a path, and the clusters would be pairs of points. If the points all fell in a tight region, there would be one nice fat cluster for the right arm to scoot around.

So I explained the idea of constructing the minimum spanning tree of a graph. The boss listened, scribbled more notes, and frowned again.

"I like your clustering idea. But minimum spanning trees are defined on graphs. All you've got are points. Where do the weights of the edges come from?"

"Oh, we can think of it as a complete graph, where every pair of points are connected. The weight of the edge is defined as the distance between the two points. Or is it. . . ?"

I went back to thinking. The edge cost should reflect the travel time between between two points. While distance is related to travel time, it wasn't necessarily the same thing.

"Hey. I have a question about your robot. Does it take the same amount of time to move the arm left-right as it does up-down?"

They thought a minute. "Yeah, it does. We use the same type of motor to control horizontal and vertical movements. Since the two motors for each arm are

independent, we can simultaneously move each arm both horizontally and vertically."

"So the time to move both one foot left and one foot up is exactly the same as just moving one foot left? This means that the weight for each edge should *not* be the Euclidean distance between the two points, but the biggest difference between either the $x-$ or y-coordinate. This is something we call the L_∞ metric, but we can capture it by changing the edge weights in the graph. Anything else funny about your robots?" I asked.

"Well, it takes some time for the robot to come up to speed. We should probably also factor in acceleration and deceleration of the arms."

"Darn right. The more accurately you can model the time your arm takes to move between two points, the better our solution will be. But now we have a very clean formulation. Let's code it up and let's see how well it works!"

They were somewhat skeptical whether this approach would do any good, but agreed to think about it. A few weeks later they called me back and reported that the new algorithm reduced the distance traveled by about 30% over their previous approach, at a cost of a little more computational preprocessing. However, since their testing machine cost $200,000 a pop and a PC cost $2,000, this was an excellent tradeoff. It is particularly advantageous since the preprocessing need only be done once when testing multiple instances of a particular board design.

The key idea leading to the successful solution was modeling the job in terms of classical algorithmic graph problems. I smelled TSP the instant they started talking about minimizing robot motion. Once I realized that they were implicitly forming a star-shaped spanning tree to ensure connectivity, it was natural to ask whether the minimum spanning tree would perform any better. This idea led to clustering, and thus partitioning each net into smaller nets. Finally, by carefully designing our distance metric to accurately model the costs of the robot itself, we could incorporate such complicated properties (as acceleration) without changing our fundamental graph model or algorithm design.

Take-Home Lesson: Most applications of graphs can be reduced to standard graph properties where well-known algorithms can be used. These include minimum spanning trees, shortest paths, and other problems presented in the catalog.

6.3 Shortest Paths

A *path* is a sequence of edges connecting two vertices. Since movie director Mel Brooks ("The Producers") is my father's sister's husband's cousin, there is a path in the friendship graph between me and him, shown in Figure 6.7—even though the two of us have never met. But if I were trying to impress how tight I am with Cousin Mel, I would be much better off saying that my Uncle Lenny grew up with him. I have a friendship path of length 2 to Cousin Mel through Uncle Lenny, while

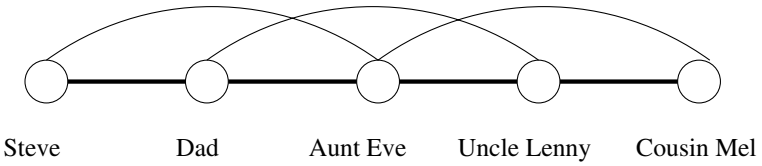


Figure 6.7: Mel Brooks is my father's sister's husband's cousin

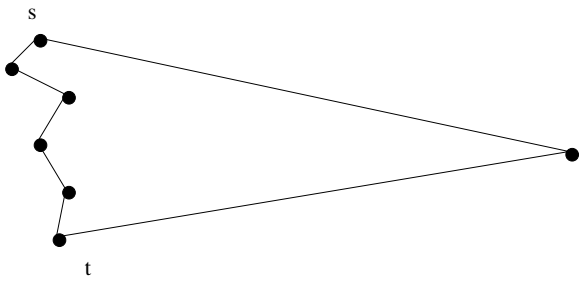


Figure 6.8: The shortest path from s to t may pass through many intermediate vertices

the path is of length 4 by blood and marriage. This multiplicity of paths hints at why finding the *shortest path* between two nodes is important and instructive, even in nontransportation applications.

The shortest path from s to t in an unweighted graph can be constructed using a breadth-first search from s. The minimum-link path is recorded in the breadth-first search tree, and it provides the shortest path when all edges have equal weight.

However, BFS does *not* suffice to find shortest paths in weighted graphs. The shortest weighted path might use a large number of edges, just as the shortest route (timewise) from home to office may involve complicated shortcuts using backroads, as shown in Figure 6.8.

In this section, we will present two distinct algorithms for finding the shortest paths in weighted graphs.

6.3.1 Dijkstra's Algorithm

Dijkstra's algorithm is the method of choice for finding shortest paths in an edge- and/or vertex-weighted graph. Given a particular start vertex s, it finds the shortest path from s to every other vertex in the graph, including your desired destination t.

Suppose the shortest path from s to t in graph G passes through a particular intermediate vertex x. Clearly, this path must contain the shortest path from s to x as its prefix, because if not, we could shorten our s-to-t path by using the shorter

s-to-x prefix. Thus, we must compute the shortest path from s to x before we find the path from s to t.

Dijkstra's algorithm proceeds in a series of rounds, where each round establishes the shortest path from s to *some* new vertex. Specifically, x is the vertex that minimizes $dist(s, v_i) + w(v_i, x)$ over all unfinished $1 \leq i \leq n$, where $w(i, j)$ is the length of the edge from i to j, and $dist(i, j)$ is the length of the shortest path between them.

This suggests a dynamic programming-like strategy. The shortest path from s to itself is trivial unless there are negative weight edges, so $dist(s, s) = 0$. If (s, y) is the lightest edge incident to s, then this implies that $dist(s, y) = w(s, y)$. Once we determine the shortest path to a node x, we check all the outgoing edges of x to see whether there is a better path from s to some unknown vertex through x.

```
ShortestPath-Dijkstra(G, s, t)
    known = {s}
    for i = 1 to n, dist[i] = ∞
    for each edge (s, v), dist[v] = w(s, v)
    last = s
    while (last ≠ t)
        select v_next, the unknown vertex minimizing dist[v]
        for each edge (v_next, x), dist[x] = min[dist[x], dist[v_next] + w(v_next, x)]
        last = v_next
        known = known ∪ {v_next}
```

The basic idea is very similar to Prim's algorithm. In each iteration, we add exactly one vertex to the tree of vertices for which we *know* the shortest path from s. As in Prim's, we keep track of the best path seen to date for all vertices outside the tree, and insert them in order of increasing cost.

The difference between Dijkstra's and Prim's algorithms is how they rate the desirability of each outside vertex. In the minimum spanning tree problem, all we cared about was the weight of the next potential tree edge. In shortest path, we want to include the closest outside vertex (in shortest-path distance) to s. This is a function of both the new edge weight *and* the distance from s to the tree vertex it is adjacent to.

Implementation

The pseudocode actually obscures how similar the two algorithms are. In fact, the change is very minor. Below, we give an implementation of Dijkstra's algorithm based on changing exactly three lines from our Prim's implementation—one of which is simply the name of the function!

```
dijkstra(graph *g, int start)      /* WAS prim(g,start) */
{
     int i;                    /* counter */
     edgenode *p;              /* temporary pointer */
     bool intree[MAXV+1];      /* is the vertex in the tree yet? */
     int distance[MAXV+1];     /* distance vertex is from start */
     int v;                    /* current vertex to process */
     int w;                    /* candidate next vertex */
     int weight;               /* edge weight */
     int dist;                 /* best current distance from start */

     for (i=1; i<=g->nvertices; i++) {
             intree[i] = FALSE;
             distance[i] = MAXINT;
             parent[i] = -1;
     }

     distance[start] = 0;
     v = start;

     while (intree[v] == FALSE) {
            intree[v] = TRUE;
            p = g->edges[v];
            while (p != NULL) {
                    w = p->y;
                    weight = p->weight;
/* CHANGED */              if (distance[w] > (distance[v]+weight)) {
/* CHANGED */                      distance[w] = distance[v]+weight;
/* CHANGED */                      parent[w] = v;
                       }
                       p = p->next;
                }

               v = 1;
               dist = MAXINT;
               for (i=1; i<=g->nvertices; i++)
                     if ((intree[i] == FALSE) && (dist > distance[i])) {
                             dist = distance[i];
                             v = i;
                       }
        }
}
```

This algorithm finds more than just the shortest path from s to t. It finds the shortest path from s to all other vertices. This defines a shortest path spanning tree rooted in s. For undirected graphs, this would be the breadth-first search tree, but in general it provides the shortest path from s to all other vertices.

Analysis

What is the running time of Dijkstra's algorithm? As implemented here, the complexity is $O(n^2)$. This is the same running time as a proper version of Prim's algorithm; except for the extension condition it *is* the same algorithm as Prim's.

The length of the shortest path from `start` to a given vertex t is exactly the value of `distance[t]`. How do we use `dijkstra` to find the actual path? We follow the backward `parent` pointers from t until we hit `start` (or `-1` if no such path exists), exactly as was done in the `find_path()` routine of Section 5.6.2 (page 165).

Dijkstra works correctly only on graphs without negative-cost edges. The reason is that midway through the execution we may encounter an edge with weight so negative that it changes the cheapest way to get from s to some other vertex already in the tree. Indeed, the most cost-effective way to get from your house to your next-door neighbor would be repeatedly through the lobby of any bank offering you enough money to make the detour worthwhile.

Most applications do not feature negative-weight edges, making this discussion academic. Floyd's algorithm, discussed below, works correctly unless there are negative cost cycles, which grossly distort the shortest-path structure. Unless that bank limits its reward to one per customer, you might so benefit by making an infinite number of trips through the lobby that you would *never* decide to actually reach your destination!

Stop and Think: Shortest Path with Node Costs

Problem: Suppose we are given a graph whose weights are on the vertices, instead of the edges. Thus, the cost of a path from x to y is the sum of the weights of all vertices on the path.

Give an efficient algorithm for finding shortest paths on vertex-weighted graphs.

Solution: A natural idea would be to adapt the algorithm we have for edge-weighted graphs (Dijkstra's) to the new vertex-weighted domain. It should be clear that we can do it. We replace any reference to the weight of an edge with the weight of the destination vertex. This can be looked up as needed from an array of vertex weights.

However, my preferred approach would leave Dijkstra's algorithm intact and instead concentrate on constructing an edge-weighted graph on which Dijkstra's

algorithm will give the desired answer. Set the weight of each directed edge (i, j) in the input graph to the cost of vertex j. Dijkstra's algorithm now does the job.

This technique can be extended to a variety of different domains, such as when there are costs on both vertices and edges. ∎

6.3.2 All-Pairs Shortest Path

Suppose you want to find the "center" vertex in a graph—the one that minimizes the longest or average distance to all the other nodes. This might be the best place to start a new business. Or perhaps you need to know a graph's *diameter*—the longest shortest-path distance over all pairs of vertices. This might correspond to the longest possible time it takes a letter or network packet to be delivered. These and other applications require computing the shortest path between all pairs of vertices in a given graph.

We could solve *all-pairs shortest path* by calling Dijkstra's algorithm from each of the n possible starting vertices. But Floyd's all-pairs shortest-path algorithm is a slick way to construct this $n \times n$ distance matrix from the original weight matrix of the graph.

Floyd's algorithm is best employed on an adjacency matrix data structure, which is no extravagance since we must store all n^2 pairwise distances anyway. Our `adjacency_matrix` type allocates space for the largest possible matrix, and keeps track of how many vertices are in the graph:

```
typedef struct {
     int weight[MAXV+1][MAXV+1];  /* adjacency/weight info */
     int nvertices;               /* number of vertices in graph */
} adjacency_matrix;
```

The critical issue in an adjacency matrix implementation is how we denote the edges absent from the graph. A common convention for unweighted graphs denotes graph edges by 1 and non-edges by 0. This gives exactly the wrong interpretation if the numbers denote edge weights, for the non-edges get interpreted as a free ride between vertices. Instead, we should initialize each non-edge to `MAXINT`. This way we can both test whether it is present and automatically ignore it in shortest-path computations, since only real edges will be used, provided `MAXINT` is less than the diameter of your graph.

There are several ways to characterize the shortest path between two nodes in a graph. The Floyd-Warshall algorithm starts by numbering the vertices of the graph from 1 to n. We use these numbers not to label the vertices, but to order them. Define $W[i, j]^k$ to be the length of the shortest path from i to j using only vertices numbered from $1, 2, ..., k$ as possible intermediate vertices.

What does this mean? When $k = 0$, we are allowed no intermediate vertices, so the only allowed paths are the original edges in the graph. Thus the initial

all-pairs shortest-path matrix consists of the initial adjacency matrix. We will perform n iterations, where the kth iteration allows only the first k vertices as possible intermediate steps on the path between each pair of vertices x and y.

At each iteration, we allow a richer set of possible shortest paths by adding a new vertex as a possible intermediary. Allowing the kth vertex as a stop helps only if there is a short path that goes through k, so

$$W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1})$$

The correctness of this is somewhat subtle, and I encourage you to convince yourself of it. But there is nothing subtle about how simple the implementation is:

```
floyd(adjacency_matrix *g)
{
    int i,j;                    /* dimension counters */
    int k;                      /* intermediate vertex counter */
    int through_k;              /* distance through vertex k */

    for (k=1; k<=g->nvertices; k++)
            for (i=1; i<=g->nvertices; i++)
                 for (j=1; j<=g->nvertices; j++) {
                       through_k = g->weight[i][k]+g->weight[k][j];
                       if (through_k < g->weight[i][j])
                               g->weight[i][j] = through_k;
                 }
}
```

The Floyd-Warshall all-pairs shortest path runs in $O(n^3)$ time, which is asymptotically no better than n calls to Dijkstra's algorithm. However, the loops are so tight and the program so short that it runs better in practice. It is notable as one of the rare graph algorithms that work better on adjacency matrices than adjacency lists.

The output of Floyd's algorithm, as it is written, does not enable one to reconstruct the actual shortest path between any given pair of vertices. These paths can be recovered if we retain a parent matrix P of our choice of the last intermediate vertex used for each vertex pair (x, y). Say this value is k. The shortest path from x to y is the concatenation of the shortest path from x to k with the shortest path from k to y, which can be reconstructed recursively given the matrix P. Note, however, that most all-pairs applications need only the resulting distance matrix. These jobs are what Floyd's algorithm was designed for.

6.3.3 Transitive Closure

Floyd's algorithm has another important application, that of computing *transitive closure*. In analyzing a directed graph, we are often interested in which vertices are reachable from a given node.

As an example, consider the *blackmail graph*, where there is a directed edge (i, j) if person i has sensitive-enough private information on person j so that i can get j to do whatever he wants. You wish to hire one of these n people to be your personal representative. Who has the most power in terms of blackmail potential?

A simplistic answer would be the vertex of highest degree, but an even better representative would be the person who has blackmail chains leading to the most other parties. Steve might only be able to blackmail Miguel directly, but if Miguel can blackmail everyone else then Steve is the man you want to hire.

The vertices reachable from any single node can be computed using breadth-first or depth-first searches. But the whole batch can be computed using an all-pairs shortest-path. If the shortest path from i to j remains `MAXINT` after running Floyd's algorithm, you can be sure no directed path exists from i to j. Any vertex pair of weight less than `MAXINT` must be reachable, both in the graph-theoretic and blackmail senses of the word.

Transitive closure is discussed in more detail in the catalog in Section 15.5.

6.4 War Story: Dialing for Documents

I was part of a group visiting Periphonics, which was then an industry leader in building telephone voice-response systems. These are more advanced versions of the *Press 1 for more options, Press 2 if you didn't press 1* telephone systems that blight everyone's lives. We were being given the standard tour when someone from our group asked, "Why don't you guys use voice recognition for data entry. It would be a lot less annoying than typing things out on the keypad."

The tour guide reacted smoothly. "Our customers have the option of incorporating speech recognition into our products, but very few of them do. User-independent, connected-speech recognition is not accurate enough for most applications. Our customers prefer building systems around typing text on the telephone keyboards."

"Prefer typing, my pupik!" came a voice from the rear of our group. "I *hate* typing on a telephone. Whenever I call my brokerage house to get stock quotes some machine tells me to type in the three letter code. To make things worse, I have to hit two buttons to type in one letter, in order to distinguish between the three letters printed on each key of the telephone. I hit the 2 key and it says Press 1 for A, Press 2 for B, Press 3 for C. Pain in the neck if you ask me."

"Maybe you don't have to hit two keys for each letter!" I chimed in. "Maybe the system could figure out the correct letter from context!"