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,
© Springer-Verlag London Limited 2008

```
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

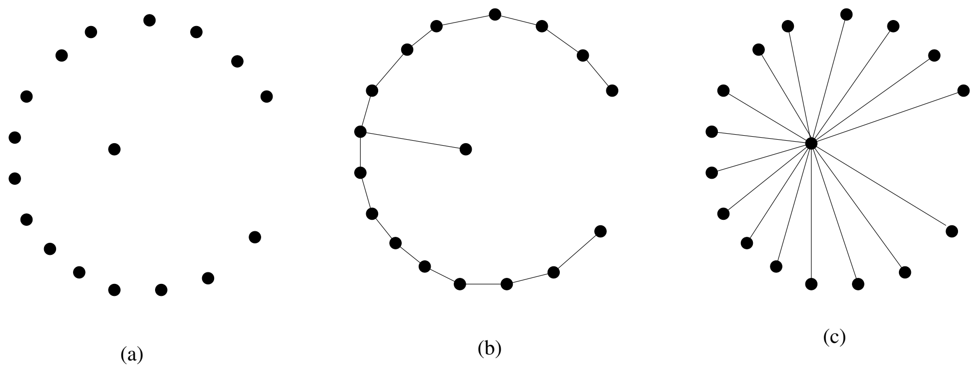

Figure 6.1: (a) Two spanning trees of point set; (b) the minimum spanning tree, and (c) the shortest path from center tree

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.

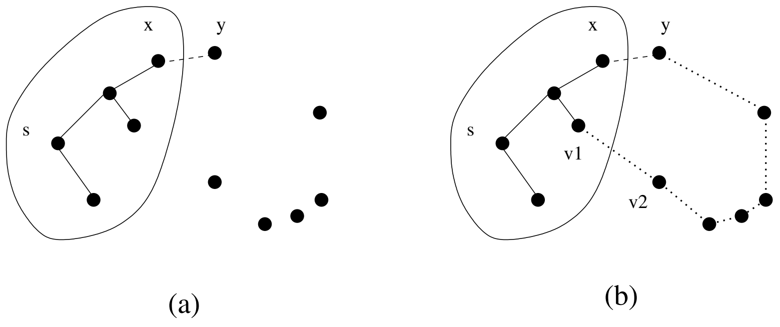

Figure 6.2: Where Prim's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

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

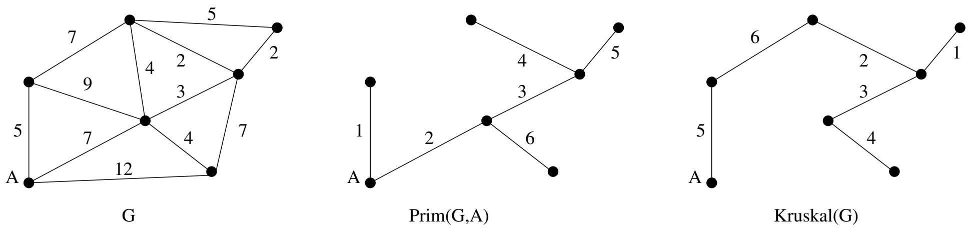

Figure 6.3: A graph G (l) with minimum spanning trees produced by Prim's (m) and Kruskal's (r) algorithms. The numbers on the trees denote the order of insertion; ties are broken arbitrarily

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 $(v) \neq$ component$(w)$)
            add to $T_{kruskal}$
            merge component$(v)$ and component$(w)$
```

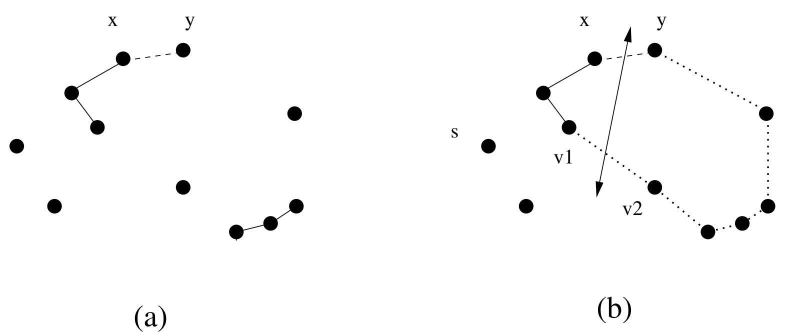

Figure 6.4: Where Kruskal's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

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

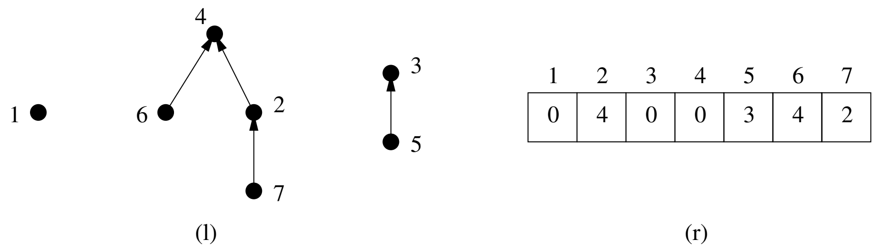

Figure 6.5: Union-find example: structure represented as trees (l) and array (r)

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?"

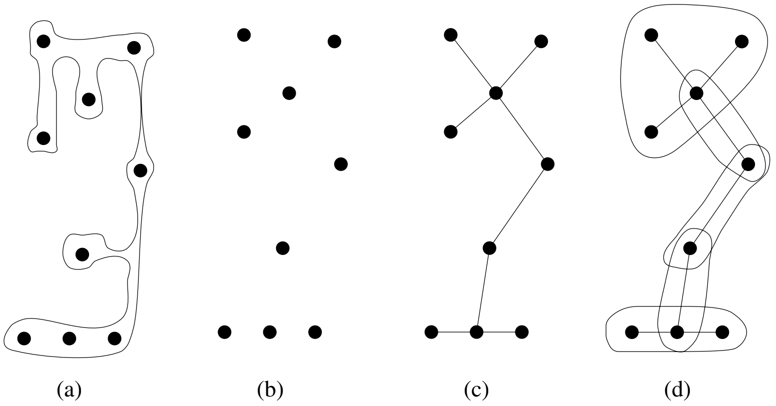

Figure 6.6: An example net showing (a) the metal connection layer, (b) the contact points, (c) their minimum spanning tree, and (d) the points partitioned into clusters

"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

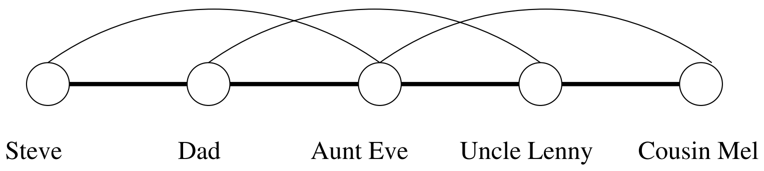

Figure 6.7: Mel Brooks is my father's sister's husband's cousin

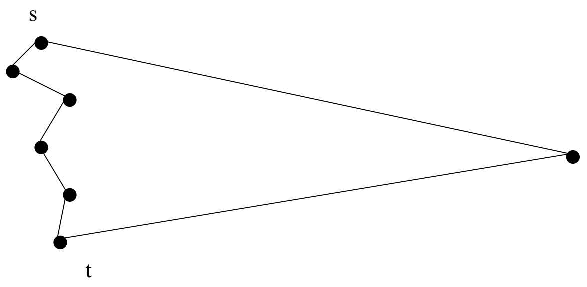

Figure 6.8: The shortest path from s to t may pass through many intermediate vertices

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 $n$, $dist[i] = \infty$
    for each edge $(s, v)$, $dist[v] = w(s, v)$
    $last = s$
    while $(last \neq t)$
        select $v_{next}$, the unknown vertex minimizing $dist[v]$
        for each edge $(v_{next}, x)$, $dist[x] = \min[dist[x], dist[v_{next}] + w(v_{next}, x)]$
        $last = v_{next}$
        $known = known \cup \{v_{next}\}$
```

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!"

"There isn't a whole lot of context when you type in three letters of stock market code."

"Sure, but there would be plenty of context if we typed in English sentences. I'll bet that we could reconstruct English text correctly if they were typed in a telephone at one keystroke per letter."

The guy from Periphonics gave me a disinterested look, then continued the tour. But when I got back to the office, I decided to give it a try.

Not all letters are equally likely to be typed on a telephone. In fact, not all letters *can* be typed, since Q and Z are not labeled on a standard American telephone. Therefore, we adopted the convention that Q, Z, and "space" all sat on the * key. We could take advantage of the uneven distribution of letter frequencies to help us decode the text. For example, if you hit the 3 key while typing English, you more likely meant to type an E than either a D or F. By taking into account the frequencies of a window of three characters (trigrams), we could predict the typed text. This is what happened when I tried it on the Gettysburg Address:

```
enurraore ane reten yeasr ain our ectherr arotght eosti on ugis aootinent a oey oation
aoncdivee in licesty ane eedicatee un uhe rrorosition uiat all oen are arectee e ual

ony ye are enichde in a irect aitil yar uestini yhethes uiat oatioo or aoy oation ro aoncdivee
ane ro eedicatee aan loni eneure ye are oet on a irect aattlediele oe uiat yar ye iate aone
un eedicate a rostion oe uiat eiele ar a einal restini rlace eor uiore yin iere iate uhdis lives
uiat uhe oation ogght live it is aluniethes eittini ane rrores uiat ye rioule en ugir

att in a laries reore ye aan oou eedicate ye aan oou aoorearate ye aan oou ialloy ugis
iroune the arate oen litini ane eeae yin rustgilee iere iate aoorearatee it ear aante our
roor rowes un ade or eeuraat the yople yill little oote oor loni renences yiat ye ray iere
att it aan oetes eosiet yiat uhfy eie iere it is eor ur uhe litini rathes un ae eedicatee iere
un uhe undiniside yopl yhici uhfy yin entght iere iate uiur ear ro onaky aetancde it is
rathes eor ur un ae iere eedicatee un uhe irect uarl rencinini adeore ur uiat eron uhere
ioooree eeae ye uale inarearee eeuotion uo tiat aaure eor yhici uhfy iere iate uhe lart eull
oearure oe eeuotioo tiat ye iere iggily rerolue uiat uhere eeae riall oou iate eide io
```

The trigram statistics did a decent job of translating it into Greek, but a terrible job of transcribing English. One reason was clear. This algorithm knew nothing about English words. If we coupled it with a dictionary, we might be onto something. But two words in the dictionary are often represented by the exact same string of phone codes. For an extreme example, the code string "22737" collides with eleven distinct English words, including *cases, cares, cards, capes, caper*, and *bases*. For our next attempt, we reported the unambiguous characters of any words that collided in the dictionary, and used trigrams to fill in the rest of the characters. We were rewarded with:

```
eourscore and seven yearr ain our eatherr brought forth on this continent azoey nation
conceivee in liberty and dedicatee uo uhe proposition that all men are createe equal

ony ye are engagee in azipeat civil yar uestioi whether that nation or aoy nation ro
conceivee and ro dedicatee aan long endure ye are oet on azipeat battlefield oe that yar
ye iate aone uo dedicate a rostion oe that field ar a final perthni place for those yin here
iate their lives that uhe nation oight live it is altogether fittinizane proper that ye should
en this

aut in a larges sense ye aan oou dedicate ye aan oou consecrate ye aan oou hallow this
ground the arate men litioi and deae yin strugglee here iate consecratee it ear above
our roor power uo ade or detract the world will little oote oor long remember what ye
```

ray here aut it aan meter forget what uhfy die here it is for ur uhe litioi rather uo ae dedicatee here uo uhe toeioisgee york which uhfy yin fought here iate thus ear ro mocky advancee it is rather for ur uo ae here dedicatee uo uhe great task renagogoi adfore ur that from there honoree deae ye uale increasee devotion uo that aause for which uhfy here iate uhe last eull measure oe devotion that ye here highky resolve that there deae shall oou iate fide io vain that this nation under ioe shall iate azoey birth oe freedom and that ioternmenu oe uhe people ay uhe people for uhe people shall oou perish from uhe earth

If you were a student of American history, maybe you could recognize it, but you certainly couldn't read it. Somehow, we had to distinguish between the different dictionary words that got hashed to the same code. We could factor in the relative popularity of each word, but this still made too many mistakes.

At this point, I started working with Harald Rau on the project, who proved to be a great collaborator. First, he was a bright and persistent graduate student. Second, as a native German speaker, he believed every lie I told him about English grammar.

Harald built up a phone code reconstruction program along the lines of Figure 6.9. It worked on the input one sentence at a time, identifying dictionary words that matched each code string. The key problem was how to incorporate grammatical constraints.

"We can get good word-use frequencies and grammatical information from a big text database called the Brown Corpus. It contains thousands of typical English sentences, each parsed according to parts of speech. But how do we factor it all in?" Harald asked.

"Let's think about it as a graph problem," I suggested.

"*Graph problem?* What graph problem? Where is there even a graph?"

"Think of a sentence as a series of phone tokens, each representing a word in the sentence. Each phone token has a list of words from the dictionary that match it. How can we choose which one is right? Each possible sentence interpretation can be thought of as a path in a graph. The vertices of this graph are the complete set of possible word choices. There will be an edge from each possible choice for the ith word to each possible choice for the $(i + 1)$st word. The cheapest path across this graph defines the best interpretation of the sentence."

"But all the paths look the same. They have the same number of edges. Wait. Now I see! We have to add weight to the edges to make the paths different."

"Exactly! The cost of an edge will reflect how likely it is that we will travel through the given pair of words. Perhaps we can count how often that pair of words occurred together in previous texts. Or we can weigh them by the part of speech of each word. Maybe nouns don't like to be next to nouns as much as they like being next to verbs."

"It will be hard to keep track of word-pair statistics, since there are so many of them. But we certainly know the frequency of each word. How can we factor that into things?"

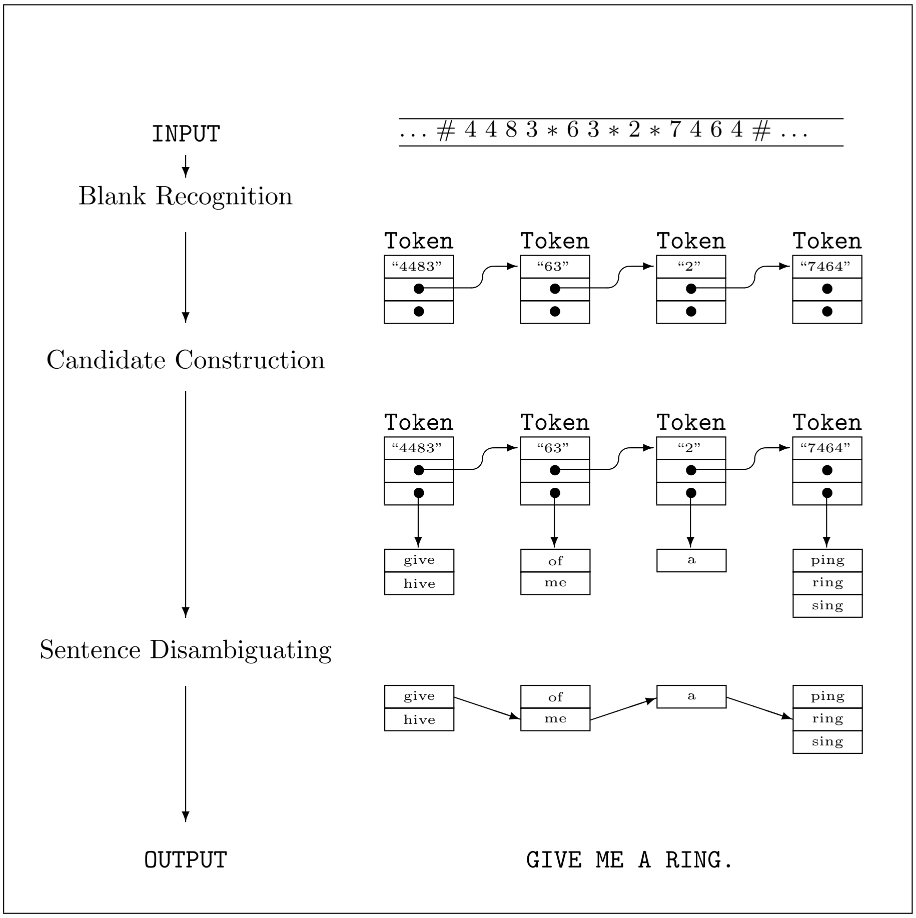

Figure 6.9: The phases of the telephone code reconstruction process

"We can pay a cost for walking through a particular vertex that depends upon the frequency of the word. Our best sentence will be given by the shortest path across the graph."

"But how do we figure out the relative weights of these factors?"

"First try what seems natural to you and then we can experiment with it."

Harald incorporated this shortest-path algorithm. With proper grammatical and statistical constraints, the system performed great. Look at the Gettysburg Address now, with all the reconstruction errors highlighted:

FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL. NOW WE ARE

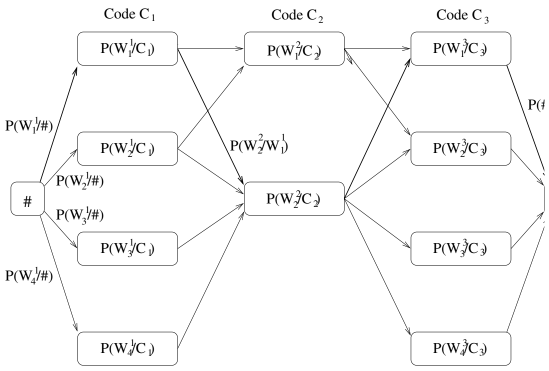

Figure 6.10: The minimum-cost path defines the best interpretation for a sentence

Text	characters	characters correct	non-blanks correct	words correct	time per character
Clinton Speeches	1,073,593	99.04%	98.86%	97.67%	0.97ms
Herland	278,670	98.24%	97.89%	97.02%	0.97ms
Moby Dick	1,123,581	96.85%	96.25%	94.75%	1.14ms
Bible	3,961,684	96.20%	95.39%	95.39%	1.33ms
Shakespeare	4,558,202	95.20%	94.21%	92.86%	0.99ms

Figure 6.11: Telephone-code reconstruction applied to several text samples

ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT **WAS**. WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL **SERVING** PLACE FOR THOSE WHO HERE **HAVE** THEIR LIVES THAT THE NATION MIGHT LIVE. IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO THIS. BUT IN A LARGER SENSE WE CAN NOT DEDICATE WE CAN NOT CONSECRATE WE CAN NOT HALLOW THIS GROUND. THE BRAVE MEN LIVING AND DEAD WHO STRUGGLED HERE HAVE CONSECRATED IT FAR ABOVE OUR POOR POWER TO ADD OR DETRACT. THE WORLD WILL LITTLE NOTE NOR LONG REMEMBER WHAT WE SAY HERE BUT IT CAN NEVER FORGET WHAT THEY DID HERE. IT IS FOR US THE LIVING RATHER TO BE DEDICATED HERE TO THE UNFINISHED WORK WHICH THEY WHO FOUGHT HERE HAVE THUS FAR SO NOBLY ADVANCED. IT IS RATHER FOR US TO BE HERE DEDICATED TO THE GREAT TASK REMAINING BEFORE US THAT FROM THESE HONORED DEAD WE TAKE INCREASED DEVOTION TO THAT CAUSE FOR WHICH THEY HERE **HAVE** THE LAST FULL MEASURE OF DEVOTION THAT WE HERE HIGHLY RESOLVE THAT THESE DEAD SHALL NOT HAVE DIED IN VAIN THAT THIS NATION UNDER GOD SHALL HAVE A NEW BIRTH OF FREEDOM AND THAT GOVERNMENT OF THE PEOPLE BY THE PEOPLE FOR THE PEOPLE SHALL NOT PERISH FROM THE EARTH.

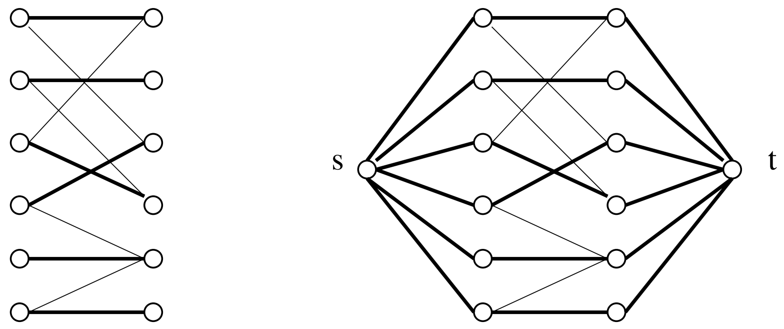

Figure 6.12: Bipartite graph with a maximum matching highlighted (on left). The corresponding network flow instance highlighting the maximum $s-t$ flow (on right).

While we still made a few mistakes, the results are clearly good enough for many applications. Periphonics certainly thought so, for they licensed our program to incorporate into their products. Figure 6.11 shows that we were able to reconstruct correctly over 99% of the characters in a megabyte of President Clinton's speeches, so if Bill had phoned them in, we would certainly be able to understand what he was saying. The reconstruction time is fast enough, indeed faster than you can type it in on the phone keypad.

The constraints for many pattern recognition problems can be naturally formulated as shortest path problems in graphs. In fact, there is a particularly convenient dynamic programming solution for these problems (the Viterbi algorithm) that is widely used in speech and handwriting recognition systems. Despite the fancy name, the Viterbi algorithm is basically solving a shortest path problem on a DAG. Hunting for a graph formulation to solve any given problem is often a good idea.

6.5 Network Flows and Bipartite Matching

Edge-weighted graphs can be interpreted as a network of pipes, where the weight of edge (i, j) determines the *capacity* of the pipe. Capacities can be thought of as a function of the cross-sectional area of the pipe. A wide pipe might be able to carry 10 units of flow in a given time, where as a narrower pipe might only carry 5 units. The *network flow problem* asks for the maximum amount of flow which can be sent from vertices s to t in a given weighted graph G while respecting the maximum capacities of each pipe.

6.5.1 Bipartite Matching

While the network flow problem is of independent interest, its primary importance is in to solving other important graph problems. A classic example is bipartite matching. A *matching* in a graph $G = (V, E)$ is a subset of edges $E' \subset E$ such that no two edges of E' share a vertex. A matching pairs off certain vertices such that every vertex is in, at most, one such pair, as shown in Figure 6.12.

Graph G is *bipartite* or *two-colorable* if the vertices can be divided into two sets, L and R, such that all edges in G have one vertex in L and one vertex in R. Many naturally defined graphs are bipartite. For example, certain vertices may represent jobs to be done and the remaining vertices represent people who can potentially do them. The existence of edge (j, p) means that job j can be done by person p. Or let certain vertices represent boys and certain vertices represent girls, with edges representing compatible pairs. Matchings in these graphs have natural interpretations as job assignments or as marriages, and are the focus of Section 15.6 (page 498).

The largest bipartite matching can be readily found using network flow. Create a *source* node s that is connected to every vertex in L by an edge of weight 1. Create a *sink* node t and connect it to every vertex in R by an edge of weight 1. Finally, assign each edge in the bipartite graph G a weight of 1. Now, the maximum possible flow from s to t defines the largest matching in G. Certainly we can find a flow as large as the matching by using only the matching edges and their source-to-sink connections. Further, there can be no greater possible flow. How can we ever hope to get more than one flow unit through any vertex?

6.5.2 Computing Network Flows

Traditional network flow algorithms are based on the idea of *augmenting paths*, and repeatedly finding a path of positive capacity from s to t and adding it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds to the flow, we must eventually find the global maximum.

The key structure is the *residual flow graph*, denoted as $R(G, f)$, where G is the input graph and f is the current flow through G. This directed, edge-weighted $R(G, f)$ contains the same vertices as G. For each edge (i, j) in G with capacity $c(i, j)$ and flow $f(i, j)$, $R(G, f)$ may contain two edges:

(i) an edge (i, j) with weight $c(i, j) - f(i, j)$, if $c(i, j) - f(i, j) > 0$ and

(ii) an edge (j, i) with weight $f(i, j)$, if $f(i, j) > 0$.

The presence of edge (i, j) in the residual graph indicates that positive flow can be pushed from i to j. The weight of the edge gives the exact amount that can be pushed. A path in the residual flow graph from s to t implies that more flow can be pushed from s to t and the minimum edge weight on this path defines the amount of extra flow that can be pushed.

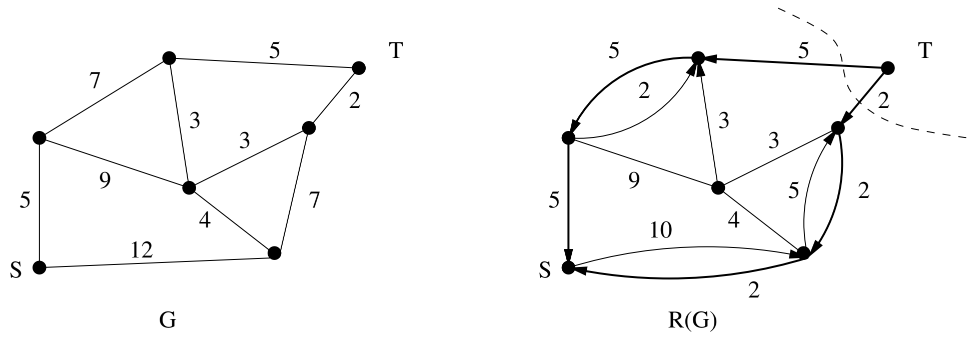

Figure 6.13: Maximum $s-t$ flow in a graph G (on left) showing the associated residual graph $R(G)$ and minimum $s-t$ cut (dotted line near t)

Figure 6.13 illustrates this idea. The maximum $s-t$ flow in graph G is 7. Such a flow is revealed by the two directed t to s paths in the residual graph $R(G)$ of capacities 2 + 5, respectively. These flows completely saturate the capacity of the two edges incident to vertex t, so no augmenting path remains. Thus the flow is optimal. A set of edges whose deletion separates s from t (like the two edges incident to t) is called an s-t cut. Clearly, no s to t flow can exceed the weight of the minimum such cut. In fact, a flow equal to the minimum cut is always possible.

Take-Home Lesson: The maximum flow from s to t always equals the weight of the minimum s-t cut. Thus, flow algorithms can be used to solve general edge and vertex connectivity problems in graphs.

Implementation

We cannot do full justice to the theory of network flows here. However, it is instructive to see how augmenting paths can be identified and the optimal flow computed.

For each edge in the residual flow graph, we must keep track of both the amount of flow currently going through the edge, as well as its remaining *residual* capacity. Thus, we must modify our `edge` structure to accommodate the extra fields:

```
typedef struct {
        int v;                     /* neighboring vertex */
        int capacity;              /* capacity of edge */
        int flow;                  /* flow through edge */
        int residual;              /* residual capacity of edge */
        struct edgenode *next;     /* next edge in list */
} edgenode;
```

We use a breadth-first search to look for any path from source to sink that increases the total flow, and use it to augment the total. We terminate with the optimal flow when no such *augmenting* path exists.

```
netflow(flow_graph *g, int source, int sink)
{
      int volume;          /* weight of the augmenting path */

      add_residual_edges(g);

      initialize_search(g);
      bfs(g,source);

      volume = path_volume(g, source, sink, parent);

      while (volume > 0) {
              augment_path(g,source,sink,parent,volume);
              initialize_search(g);
              bfs(g,source);
              volume = path_volume(g, source, sink, parent);
      }
}
```

Any augmenting path from source to sink increases the flow, so we can use `bfs` to find such a path in the appropriate graph. We only consider network edges that have remaining capacity or, in other words, positive residual flow. The predicate below helps `bfs` distinguish between saturated and unsaturated edges:

```
bool valid_edge(edgenode *e)
{
        if (e->residual > 0) return (TRUE);
        else return(FALSE);
}
```

Augmenting a path transfers the maximum possible volume from the residual capacity into positive flow. This amount is limited by the path-edge with the smallest amount of residual capacity, just as the rate at which traffic can flow is limited by the most congested point.

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
     edgenode *e;                    /* edge in question */
     edgenode *find_edge();

     if (parents[end] == -1) return(0);

     e = find_edge(g,parents[end],end);

     if (start == parents[end])
             return(e->residual);
     else
             return( min(path_volume(g,start,parents[end],parents),
                         e->residual) );
}


edgenode *find_edge(flow_graph *g, int x, int y)
{
        edgenode *p;                    /* temporary pointer */

        p = g->edges[x];

        while (p != NULL) {
                if (p->v == y) return(p);
                p = p->next;
        }

        return(NULL);
}
```

Sending an additional unit of flow along directed edge (i, j) reduces the residual capacity of edge (i, j) but *increases* the residual capacity of edge (j, i). Thus, the act of augmenting a path requires modifying both forward and reverse edges for each link on the path.

```
augment_path(flow_graph *g, int start, int end, int parents[], int volume)
{
        edgenode *e;                    /* edge in question */
        edgenode *find_edge();

        if (start == end) return;

        e = find_edge(g,parents[end],end);
        e->flow += volume;
        e->residual -= volume;

        e = find_edge(g,end,parents[end]);
        e->residual += volume;

        augment_path(g,start,parents[end],parents,volume);
}
```

Initializing the flow graph requires creating directed flow edges (i, j) and (j, i) for each network edge $e = (i, j)$. Initial flows are all set to 0. The initial residual flow of (i, j) is set to the capacity of e, while the initial residual flow of (j, i) is set to 0.

Analysis

The augmenting path algorithm above eventually converges on the the optimal solution. However, each augmenting path may add only a little to the total flow, so, in principle, the algorithm might take an arbitrarily long time to converge.

However, Edmonds and Karp [EK72] proved that always selecting a *shortest* unweighted augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization. In fact, the Edmonds-Karp algorithm is what is implemented above, since a breadth-first search from the source is used to find the next augmenting path.

6.6 Design Graphs, Not Algorithms

Proper modeling is the key to making effective use of graph algorithms. We have defined several graph properties, and developed algorithms for computing them. All told, about two dozen different graph problems are presented in the catalog, mostly in Sections 15 and 16. These classical graph problems provide a framework for modeling most applications.

The secret is learning to design graphs, not algorithms. We have already seen a few instances of this idea:

- The *maximum* spanning tree can be found by negating the edge weights of the input graph G and using a *minimum* spanning tree algorithm on the result. The most negative weight spanning tree will define the maximum weight tree in G.

- To solve bipartite matching, we constructed a special network flow graph such that the maximum flow corresponds to a maximum cardinality matching.

The applications below demonstrate the power of proper modeling. Each arose in a real-world application, and each can be modeled as a graph problem. Some of the modelings are quite clever, but they illustrate the versatility of graphs in representing relationships. As you read a problem, try to devise an appropriate graph representation before peeking to see how we did it.

Stop and Think: The Pink Panther's Passport to Peril

Problem: "I'm looking for an algorithm to design natural routes for video-game characters to follow through an obstacle-filled room. How should I do it?"

Solution: Presumably the desired route should look like a path that an intelligent being would choose. Since intelligent beings are either lazy or efficient, this should be modeled as a shortest path problem.

But what is the graph? One approach might be to lay a grid of points in the room. Create a vertex for each grid point that is a valid place for the character to stand; i.e. , that does not lie within an obstacle. There will be an edge between any pair of nearby vertices, weighted proportionally to the distance between them. Although direct geometric methods are known for shortest paths (see Section 15.4 (page 489)), it is easier to model this discretely as a graph. ∎

Stop and Think: Ordering the Sequence

Problem: "A DNA sequencing project generates experimental data consisting of small fragments. For each given fragment f, we know certain other fragments are forced to lie to the left of f, and certain other fragments are forced to be to the right of f. How can we find a consistent ordering of the fragments from left to right that satisfies all the constraints?"

Solution: Create a directed graph, where each fragment is assigned a unique vertex. Insert a directed edge $\overrightarrow{(l, f)}$ from any fragment l that is forced to be to the left of f, and a directed edge $\overrightarrow{(f, r)}$ to any fragment r forced to be to the right of f. We seek an ordering of the vertices such that all the edges go from left to right. This is a *topological sort* of the resulting directed acyclic graph. The graph must be acyclic, because cycles make finding a consistent ordering impossible. ■

Stop and Think: Bucketing Rectangles

Problem: "In my graphics work I need to solve the following problem. Given an arbitrary set of rectangles in the plane, how can I distribute them into a minimum number of buckets such that no subset of rectangles in any given bucket intersects another? In other words, there can not be any overlapping area between two rectangles in the same bucket."

Solution: We formulate a graph where each vertex is a rectangle, and there is an edge if two rectangles intersect. Each bucket corresponds to an *independent set* of rectangles, so there is no overlap between any two. A *vertex coloring* of a graph is a partition of the vertices into independent sets, so minimizing the number of colors is exactly what you want. ■

Stop and Think: Names in Collision

Problem: "In porting code from UNIX to DOS, I have to shorten several hundred file names down to at most 8 characters each. I can't just use the first eight characters from each name, because "filename1" and "filename2" would be assigned the exact same name. How can I meaningfully shorten the names while ensuring that they do not collide?"

Solution: Construct a bipartite graph with vertices corresponding to each original file name f_i for $1 \leq i \leq n$, as well as a collection of acceptable shortenings for each name $f_{i1}, \ldots, f_{ik}$. Add an edge between each original and shortened name. We now seek a set of n edges that have no vertices in common, so each file name is mapped to a distinct acceptable substitute. *Bipartite matching*, discussed in Section 15.6 (page 498), is exactly this problem of finding an independent set of edges in a graph. ■

Stop and Think: Separate the Text

Problem: "We need a way to separate the lines of text in the optical character-recognition system that we are building. Although there is some white space between the lines, problems like noise and the tilt of the page makes it hard to find. How can we do line segmentation?

Solution: Consider the following graph formulation. Treat each pixel in the image as a vertex in the graph, with an edge between two neighboring pixels. The weight of this edge should be proportional to how dark the pixels are. A segmentation between two lines is a path in this graph from the left to right side of the page. We seek a relatively straight path that avoids as much blackness as possible. This suggests that the *shortest path* in the pixel graph will likely find a good line segmentation. ∎

> *Take-Home Lesson:* Designing novel graph algorithms is very hard, so don't do it. Instead, try to design graphs that enable you to use classical algorithms to model your problem.

Chapter Notes

Network flows are an advanced algorithmic technique, and recognizing whether a particular problem can be solved by network flow requires experience. We point the reader to books by Cook and Cunningham [CC97] and Ahuja, Magnanti, and Orlin [AMO93] for more detailed treatments of the subject.

The augmenting path method for network flows is due to Ford and Fulkerson [FF62]. Edmonds and Karp [EK72] proved that always selecting a *shortest* geodesic augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization.

The phone code reconstruction system that was the subject of the war story is described in more technical detail in [RS96].

6.7 Exercises

Simulating Graph Algorithms

6-1. *[3]* For the graphs in Problem 5-1:

(a) Draw the spanning forest after every iteration of the main loop in Kruskal's algorithm.

(b) Draw the spanning forest after every iteration of the main loop in Prim's algorithm.

(c) Find the shortest path spanning tree rooted in A.

(d) Compute the maximum flow from A to H.

Minimum Spanning Trees

6-2. *[3]* Is the path between two vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.

6-3. *[3]* Assume that all edges in the graph have distinct edge weights (i.e. , no pair of edges have the same weight). Is the path between a pair of vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.

6-4. *[3]* Can Prim's and Kruskal's algorithm yield different minimum spanning trees? Explain why or why not.

6-5. *[3]* Does either Prim's and Kruskal's algorithm work if there are negative edge weights? Explain why or why not.

6-6. *[5]* Suppose we are *given* the minimum spanning tree T of a given graph G (with n vertices and m edges) and a new edge $e = (u, v)$ of weight w that we will add to G. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit.

6-7. *[5]* (a) Let T be a minimum spanning tree of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Do the edges of T form a minimum spanning tree of G'? Prove the statement or give a counterexample.

(b) Let $P = \{s, \ldots, t\}$ describe a shortest weighted path between vertices s and t of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Does P describe a shortest path from s to t in G'? Prove the statement or give a counterexample.

6-8. *[5]* Devise and analyze an algorithm that takes a weighted graph G and finds the smallest change in the cost to a non-MST edge that would cause a change in the minimum spanning tree of G. Your algorithm must be correct and run in polynomial time.

6-9. *[4]* Consider the problem of finding a minimum weight connected subset T of edges from a weighted connected graph G. The weight of T is the sum of all the edge weights in T.

(a) Why is this problem not just the minimum spanning tree problem? Hint: think negative weight edges.

(b) Give an efficient algorithm to compute the minimum weight connected subset T.

6-10. *[4]* Let $G = (V, E)$ be an undirected graph. A set $F \subseteq E$ of edges is called a *feedback-edge set* if every cycle of G has at least one edge in F.

(a) Suppose that G is unweighted. Design an efficient algorithm to find a minimum-size feedback-edge set.

(b) Suppose that G is a weighted undirected graph with positive edge weights. Design an efficient algorithm to find a minimum-weight feedback-edge set.

6-11. *[5]* Modify Prim's algorithm so that it runs in time $O(n \log k)$ on a graph that has only k different edges costs.

Union-Find

6-12. *[5]* Devise an efficient data structure to handle the following operations on a weighted directed graph:

(a) Merge two given components.

(b) Locate which component contains a given vertex v.

(c) Retrieve a minimum edge from a given component.

6-13. *[5]* Design a data structure that can perform a sequence of, m *union* and *find* operations on a universal set of n elements, consisting of a sequence of all *unions* followed by a sequence of all *finds*, in time $O(m + n)$.

Shortest Paths

6-14. *[3]* The *single-destination shortest path* problem for a directed graph seeks the shortest path *from* every vertex to a specified vertex v. Give an efficient algorithm to solve the single-destination shortest paths problem.

6-15. *[3]* Let $G = (V, E)$ be an undirected weighted graph, and let T be the shortest-path spanning tree rooted at a vertex v. Suppose now that all the edge weights in G are increased by a constant number k. Is T still the shortest-path spanning tree from v?

6-16. *[3]* Answer all of the following:

(a) Give an example of a weighted connected graph $G = (V, E)$ and a vertex v, such that the minimum spanning tree of G is the same as the shortest-path spanning tree rooted at v.

(b) Give an example of a weighted connected directed graph $G = (V, E)$ and a vertex v, such that the minimum-cost spanning tree of G is very different from the shortest-path spanning tree rooted at v.

(c) Can the two trees be completely disjointed?

6-17. *[3]* Either prove the following or give a counterexample:

(a) Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

(b) Suppose that the minimum spanning tree of the graph is unique. Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

6-18. *[5]* In certain graph problems, vertices have can have weights instead of or in addition to the weights of edges. Let C_v be the cost of vertex v, and $C_{(x,y)}$ the cost of the edge (x, y). This problem is concerned with finding the cheapest path between vertices a and b in a graph G. The cost of a path is the sum of the costs of the edges and vertices encountered on the path.

(a) Suppose that each edge in the graph has a weight of zero (while non-edges have a cost of ∞). Assume that $C_v = 1$ for all vertices $1 \leq v \leq n$ (i.e. , all vertices have the same cost). Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

(b) Now suppose that the vertex costs are not constant (but are all positive) and the edge costs remain as above. Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

(c) Now suppose that both the edge and vertex costs are not constant (but are all positive). Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

6-19. *[5]* Let G be a weighted *directed* graph with n vertices and m edges, where all edges have positive weight. A directed cycle is a directed path that starts and ends at the same vertex and contains at least one edge. Give an $O(n^3)$ algorithm to find a directed cycle in G of minimum total weight. Partial credit will be given for an $O(n^2m)$ algorithm.

6-20. *[5]* Can we modify Dijkstra's algorithm to solve the single-source *longest* path problem by changing *minimum* to *maximum*? If so, then prove your algorithm correct. If not, then provide a counterexample.

6-21. *[5]* Let $G = (V, E)$ be a weighted acyclic directed graph with possibly negative edge weights. Design a linear-time algorithm to solve the single-source shortest-path problem from a given source v.

6-22. *[5]* Let $G = (V, E)$ be a directed weighted graph such that all the weights are positive. Let v and w be two vertices in G and $k \leq |V|$ be an integer. Design an algorithm to find the shortest path from v to w that contains exactly k edges. Note that the path need not be simple.

6-23. *[5]* *Arbitrage* is the use of discrepancies in currency-exchange rates to make a profit. For example, there may be a small window of time during which 1 U.S. dollar buys 0.75 British pounds, 1 British pound buys 2 Australian dollars, and 1 Australian dollar buys 0.70 U.S. dollars. At such a time, a smart trader can trade one U.S. dollar and end up with $0.75 \times 2 \times 0.7 = 1.05$ U.S. dollars—a profit of 5%. Suppose that there are n currencies $c_1, ..., c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j. Devise and analyze an algorithm to determine the maximum value of

$$R[c_1, c_{i1}] \cdot R[c_{i1}, c_{i2}] \cdots R[c_{ik-1}, c_{ik}] \cdot R[c_{ik}, c_1]$$

Hint: think all-pairs shortest path.

Network Flow and Matching

6-24. *[3]* A matching in a graph is a set of disjoint edges—i.e., edges that do not share any vertices in common. Give a linear-time algorithm to find a maximum matching in a tree.

6-25. *[5]* An *edge cover* of an undirected graph $G = (V, E)$ is a set of edges such that each vertex in the graph is incident to at least one edge from the set. Give an efficient algorithm, based on matching, to find the minimum-size edge cover for G.

Programming Challenges

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

6-1. "Freckles" – Programming Challenges 111001, UVA Judge 10034.

6-2. "Necklace" – Programming Challenges 111002, UVA Judge 10054.

6-3. "Railroads" – Programming Challenges 111004, UVA Judge 10039.

6-4. "Tourist Guide" – Programming Challenges 111006, UVA Judge 10199.

6-5. "The Grand Dinner" – Programming Challenges 111007, UVA Judge 10249.