Properties of Graphs

Graph theory is the study of the properties of graph structures. It provides us with a language with which to talk about graphs.

The key to solving many problems is identifying the fundamental graph-theoretic notion underlying the situation and then using classical algorithms to solve the resulting problem.

Graphs are made up of vertices and edges. The simplest property of a vertex is its *degree*, the number of edges incident upon it.

The sum of the vertex degrees in any undirected graph is twice the number of edges, since every edge contributes one to the degree of both adjacent vertices.

Trees are undirected graphs which contain no cycles. Vertex degrees are important in the analysis of trees. A *leaf* of a tree is a vertex of degree 1. Every n-vertex tree contains $n-1$ edges, so all non-trivial trees contain at least two leaf vertices.

Connectivity

A graph is *connected* if there is an undirected path between every pair of vertices.

The existence of a spanning tree is sufficient to prove connectivity. A breadth-first or depth-first search-based connected components algorithm can be used to find such a spanning tree.

However, there are other notions of connectivity. The most interesting special case when there is a single weak link in the graph. A single vertex whose deletion disconnects the graph is called an *articulation vertex*; any graph without such a vertex is said to be *biconnected*. A single edge whose deletion disconnects the graph is called a *bridge*.

Testing for articulation vertices or bridges is easy via brute force. For each vertex/edge, delete it from the graph and test whether the resulting graph remains connected. Be sure to add that vertex/edge back before doing the next deletion!

In directed graphs we are often concerned with *strongly connected components*, that is, partitioning the graph into chunks such that there are directed paths between all pairs of vertices within a given chunk. 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.

Cycles in Graphs

All non-tree connected graphs contain cycles. Particularly interesting are cycles which visit all the edges or vertices of the graph.

An *Eulerian cycle* is a tour which visits every edge of the graph exactly once. A mailman's route is ideally an Eulerian cycle, so he can visit every street (edge) in the neighborhood once before returning home.

An undirected graph contains an Eulerian cycle if it is connected and every vertex is of even degree. Why? The circuit must enter and exit every vertex it encounters, implying that all degrees must be even.

We can find an Eulerian cycle by building it one cycle at a time. We can find a simple cycle in the graph by finding a back edge using DFS. Deleting the edges on this cycle leaves each vertex with even degree. Once we have partitioned the edges into edge-disjoint cycles, we can merge these cycles arbitrarily at common vertices to build an Eulerian cycle.

A *Hamiltonian cycle* is a tour which visits every vertex of the graph exactly once. The traveling salesman problem asks for the shortest such tour on a weighted graph.

Unfortunately, no efficient algorithm exists for solving Hamiltonian cycle problems. If the graph is sufficiently small, it can be solved via backtracking.

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 the smallest possible.

Minimum spanning trees are the answer whenever we need to connect a set of points (representing cities, junctions, or other locations) by the smallest amount of roadway, wire, or pipe.

We will present Prim's algorithm here because we think it is simpler to program, and because it gives us Dijkstra's shortest path algorithm with very minimal changes.

We generalize the graph data structure to support edge-weighted graphs. Each edge-entry previously contained only the other endpoint of the given edge. We must replace this by a record allowing us to annotate the edge with weights:

```
typedef struct {
        int v;                          /* neighboring vertex */
        int weight;                     /* edge weight */
} edge;

typedef struct {
        edge edges[MAXV+1][MAXDEGREE];  /* adjacency info */
        int degree[MAXV+1];             /* outdegree of vertex */
        int nvertices;                  /* number of vertices */
        int nedges;                     /* number of edges in graph */
} graph;
```

Prim's Algorithm

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.

Our implementation keeps track of the cheapest edge from any tree vertex to every non-tree vertex in the graph. The cheapest 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.

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 total the weight of all selected edges in a variable for later return. Alternately, since the tree topology is encoded by the `parent` array it plus the original graph tells you everything about the minimum spanning tree.

Prim's Implementation

```
prim(graph *g, int start) {
   int i,j;                        /* counters */
   bool intree[MAXV];              /* is vertex in the tree yet? */
   int distance[MAXV];             /* vertex distance from start */
   int v;                          /* current vertex to process */
   int w;                          /* candidate next vertex */
   int weight;                     /* edge weight */
   int dist;                       /* shortest current distance */

   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;
       for (i=0; i<g->degree[v]; i++) {
            w = g->edges[v][i].v;
            weight = g->edges[v][i].weight;
            if ((distance[w] > weight) && (intree[w]==FALSE)) {
                    distance[w] = weight;
                    parent[w] = v;
            }
       }

       v = 1;
       dist = MAXINT;
       for (i=2; i<=g->nvertices; i++)
            if ((intree[i]==FALSE) && (dist > distance[i])) {
                    dist = distance[i];
                         v = i;
                 }
        }
}
```

Dijkstra's Algorithm for Shortest Paths

BFS does *not* suffice for finding shortest paths in weighted graphs, because the shortest weighted path from a to b does not necessarily contain the fewest number of edges.

Dijkstra's algorithm is the method of choice for finding the shortest path between two vertices 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.

The basic idea is very similar to Prim's algorithm. In each iteration, we are going to add exactly one vertex to the tree of vertices for which we *know* the shortest path from s.

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

In fact, this 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!

Implementation of Dijkstra

```
dijkstra(graph *g, int start)          /* WAS prim(g,start) */
{
   int i,j;                        /* counters */
   bool intree[MAXV];              /* is vertex in the tree yet? */
   int distance[MAXV];             /* vertex distance from start */
   int v;                          /* current vertex to process */
   int w;                          /* candidate next vertex */
   int weight;                     /* edge weight */
   int dist;                       /* shortest current distance */

   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;
       for (i=0; i<g->degree[v]; i++) {
            w = g->edges[v][i].v;
            weight = g->edges[v][i].weight;
/* CHANGED */    if (distance[w] > (distance[v]+weight)) {
/* CHANGED */            distance[w] = distance[v]+weight;
                         parent[w] = v;
                 }
       }
       v = 1;
       dist = MAXINT;
       for (i=2; i<=g->nvertices; i++)
            if ((intree[i]==FALSE) && (dist > distance[i])) {
                    dist = distance[i];
                    v = i;
            }
   }
}
```

How do we use `dijkstra` to find the length of the shortest path from `start` to a given vertex t? This is exactly the value of `distance[t]`. How can we reconstruct the actual path? By following the backward `parent` pointers from t until we hit `start` (or `-1` if no such path exists)

Unlike Prim's, Dijkstra's algorithm only works on graphs without negative-cost edges. Most applications do not feature negative-weight edges, making this discussion academic.

All-Pairs Shortest Path

Many applications need to know the length of the shortest path between all pairs of vertices in a given graph. For example, suppose you want to find the "center" vertex, the one which minimizes the longest or average distance to all the other nodes. This might be the best place to start a new business.

We could solve this problem by calling Dijkstra's algorithm from each of the n possible starting vertices. But Floyd's all-pairs shortest-path algorithm is an amazingly slick way to construct this 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 have to 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;
```

A critical issue in any adjacency matrix implementation is how we denote the edges which are not present in the graph. For unweighted graphs, a common convention is that graph edges are denoted 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`.

```
initialize_adjacency_matrix(adjacency_matrix *g)
{
        int i,j;                    /* counters */

        g -> nvertices = 0;

        for (i=1; i<=MAXV; i++)
                for (j=1; j<=MAXV; j++)
                        g->weight[i][j] = MAXINT;
}
```

Floyd's algorithm starts by numbering the vertices of the graph from 1 to n, using these numbers not to label the vertices but to order them.

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. When $k = 0$, we are allowed no intermediate vertices, so the only allowed paths consist of the original edges in the graph. Thus the initial all-pairs shortest-path matrix consists of the initial adjacency matrix. At each iteration, we allow a richer set of possible shortest paths. Allowing the kth vertex as a new possible intermediary helps only if there is a short path that goes through k, so

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

Implementation of Floyd's Algorithm

The correctness of this is somewhat subtle, and we encourage you to convince yourself of it. But there is nothing subtle about how short and sweet 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;
           }
}
```

Transitive Closure

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

For example, consider the *blackmail graph* defined on a set of n people, where there is a directed edge (i, j) if i has sensitive-enough private information on j so that i can get him 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 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 using breadth-first or depth-first search. But the whole batch can be computed as an all-pairs shortest-path problem. If the shortest path from i to j remains `MAXINT` after running Floyd's algorithm, you can be sure there is no directed path 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.

Bipartite Matching and Network Flow

Any edge-weighted graph can be thought of as a network of pipes, where the weight of edge (i,j) measures the *capacity* of the pipe. For a given weighted graph G and two vertices s and t, the *network flow problem* asks for the maximum amount of flow which can be sent from s to t while respecting the maximum capacities of each pipe.

While the network flow problem is of independent interest, its primary importance is that of being able to solve 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 in E' share a vertex. Thus a matching pairs off certain vertices such that every vertex is in at most one such pair.

Graph G is *bipartite* or *two-colorable* if the vertices can be divided into two sets, say, 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, suppose certain vertices represent jobs to be done and the remaining vertices people who can potentially do them. The existence of edge (j, p) means that job j can potentially done by person p. Or let certain vertices represent boys and certain vertices girls, with edges representing compatible pairs. Matchings in these graphs have natural interpretations as job assignments or as marriages.

The largest possible bipartite matching can be found using network flow. See the textbook for details.

Assigned Problems

111001 (Freckles) – Connect the dots using as little ink as possible. What classical graph problem does this correspond to?

111002 (The Necklace) – Does there exist a way to lace up bicolored beads so that each pair of neighboring bead-faces share a color? What classical graph problem does this correspond to? What *efficiently computed* classical graph problem does this also correspond to?

111006 (Tourist Guide) – What vertices in the graph separate the graph into two pieces, i.e. all paths between a and b must go through them for any a and b? How can we efficiently test whether v is such a vertex?

111007 (The Grand Dinner) – Match team members to tables so that no two team members sit at the same table. Can this be done using bipartite matching/network flow?