Senior Computer Team Lectures 2000-2001: Shortest Path

"Just use variables that mean something!!" - G. Galperin

by Gary Sivek

In Euclidean geometry, the shortest path between two points is a straight line. What if you don't have a straight line connecting two points? What if you don't care about distance, but cost? This lecture will answer precisely those questions.

But first, some basic terminology: a *graph* consists of vertices connected by edges. The edges may or may not be weighted and may or may not have a specified direction; when not specified, an edge may be traversed in either direction. A graph is usually denoted $G(V, E)$. Furthermore, when finding a shortest path between two vertices, or the amount of flow that can be pushed from one vertex to another, or any other graph theory-ish thing like this, the starting vertex is the *source* and the destination is the *sink*. For the purposes of this lecture you don't really need much more than this, but this is used throughout graph theory, which we will be covering more in future lectures.

Example 1: [traditional]
Farmer John is buying cows in a secret underground market. Farmer John starts at station 0 and can travel through tunnels to get to other stations. Due to high security, he is not allowed to dig any tunnels. Farmer John hears from a secret source (hereby known as Deep Udder) that one specific cow is better than the others. He also wishes to minimize the distance he must walk to get that cow. Given a number n of stations (not including station 0), a number k that tells which station FJ must reach, a number t denoting the number of existing tunnels, and then t lines with three numbers each (the two stations connected by the tunnel and the distance between them), find the least distance FJ must travel to reach the cow.

Thus the shortest path problem is introduced. This is as straightforward as it gets. We will present two algorithms here: the Floyd-Warshall Algorithm and Dijkstra's Algorithm.

1 Floyd-Warshall

This is an example of DP(!) at its best. Here's the basic algorithm:

- Start with the already defined path between two vertices A and B. If no path exists, call the distance infinity.
- Add vertex V_1 into our little sub-graph. Does going from A to V_1 to B produce a shorter path than going directly from A to B? If so, keep that distance instead.
- Add vertex V_2 into our graph. Do we get a better distance? Repeat this process on all vertices in the graph.

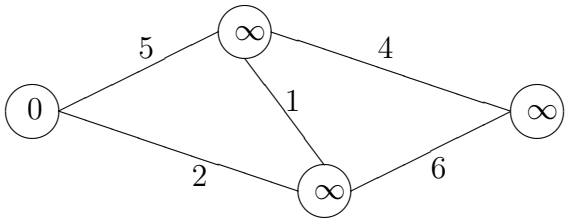
This boils down to a true statement: If the entire graph $G(V, E)$ produces a shortest distance D between to vertices A and B, then that will also be the shortest distance for some subgraph of G, even if that subgraph is G itself. So how do we code this? Here's the Floyd-Warshall algorithm in its entirety:

```
for (mid = 0; mid <= N; mid++)
  for (start = 0; start <= N; start++)
    if (dist[start][mid] != INF)
      for (end = 0; end <= N; end++)
        dist[start][end] = min(dist[start][end],
                               dist[start][mid]+dist[mid][end]);
```

Wow! you're thinking. *That's so easy to remember and code!* You are correct! This has its obvious advantages. Furthermore, it is only necessary to run this algorithm once to obtain all shortest paths, as unobvious as that may seem. Still, there are a few things to watch out for: you absolutely MUST have the `mid` variable on the outside of all the for loops (the order of the other two does not matter). Also, be careful of memory and time constraints, because this algorithm requires $O(V^2)$ space and $O(V^3)$ time (where V is the number of vertices), which may exceed the limits for some problems in some competitions. So this is one approach to the example problem. On to the next:

2 Dijkstra

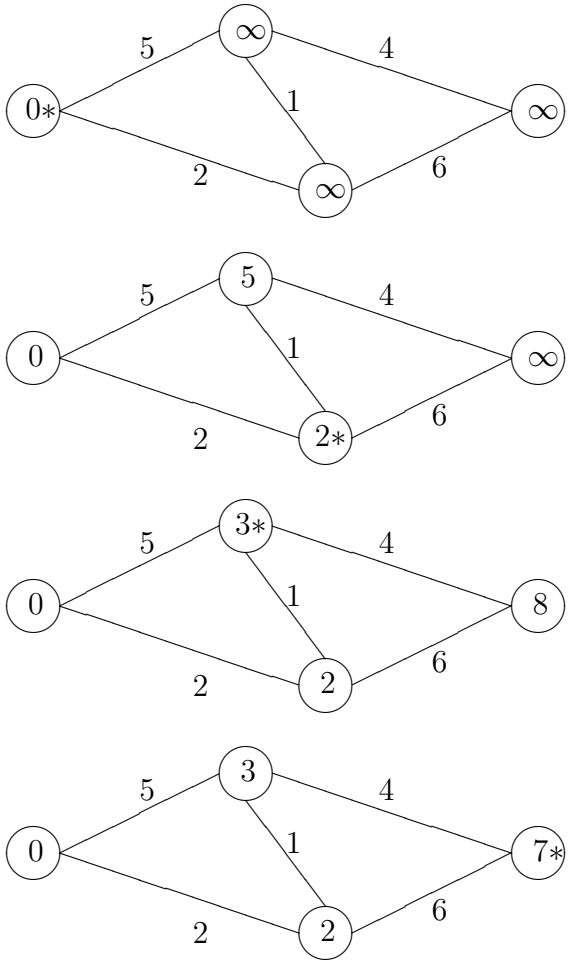
This isn't nearly as easy to code, but may be worth it. Imagine the following graph:



We want to get from the leftmost vertex to the rightmost. The distances between vertices are marked on the edges, and the current best distance to each vertex is inside the circles denoting the vertices. Note that the source has distance 0 and the others are initialized to infinity. In this algorithm, we do the following:

- Pick the vertex with the least distance from the source that has not yet been picked. If this vertex is the sink, then stop; the current distance from source to sink is the best one. Otherwise:
- Update the distances to all the neighbors of the current vertex. Updating is the same as in Floyd-Warshall ($dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])$).
- Repeat.

So this is what the graph would look like after each iteration. At each step, the vertex being processed is denoted by an asterisk:



and we are done; the shortest path from source to sink is 7. So how do we code this? Well...

```
long dijkstra(int source, int sink)
{
  int i, curr, visited[MAXN];
  long dist[MAXN];
  // Initialize: no vertices have been visited, and since we
  // haven't traveled any edges, the distances may as well
  // be infinite for all we know. Be sure to define INF so
  // that it exceeds the maximum distance possible in any case!
  for (i = 0; i <= N; i++)
  {
    visited[i] = 0;
    dist[i] = INF;
  }
  dist[source] = 0;
```

```
  while (1)
  {
    // Choose the *unvisited* vertex with the least distance.
    curr = 0;
    while (visited[curr]) curr++;
    for (i = curr+1; i <= N; i++)
      if (!visited[i] && dist[i] < dist[curr])
         curr = i;

    // If we hit the sink, we're done, so break out.
    if (curr == sink) break;

    // If the least distance is infinite, then we can't get
    // to the sink. Return some garbage value like -1.
    if (dist[curr] == INF) return -1;

    // Do the distance update thing. the arrays nadj, adj,
    // and dadj were set elsewhere (use global variables!)
    // and contain the number of vertices adjacent to a
    // given vertex, the vertices adjacent to a given vertex,
    // and the distances between those adjacent vertices,
    // respectively.
    for (i = 0; i < nadj[curr]; i++)
      dist[adj[curr][i]] = min(dist[adj[curr][i]],
                               dist[curr]+dadj[curr][i]);
  }

  return dist[sink];
}
```

Of course, the code here is incomplete; you need to do all the other stuff (i.e., file input and output, assigning the [n/d]adj arrays, etc.) elsewhere. Also, the code given here uses a selection to pick the best vertex; a heap takes longer to code, but can run faster. What are the runtimes for these algorithms? Assume we are only finding one path; then for a selection we have $O(V^2)$ and for a heap we have $O(VlgV)$. If you want to find all shortest paths on a graph $G(V, E)$, then Dijkstra becomes $O(V^3)$ with selection and $O(V^2lgV)$ with a heap.

3 Analysis

So which algorithm do you use? That depends entirely on the problem description. If you have enough memory and time, Floyd-Warshall is clearly better because it takes much less time to code. But if you don't want every possible path, Floyd-Warshall may waste time by calculating too many unwanted shortest paths. In that case, you can go with Dijkstra.

But which version do you program? Dijkstra with selection is of course easier to write, but also runs slower. Dijkstra with a heap runs faster, but takes longer to code, so you also have to take into account how much time you have to write your program. There is yet another factor: is the graph *sparse* (meaning, does it have many edges or few?). If it is, Dijkstra works just fine; if it's not, Dijkstra may take too long for finding many paths because it has too many edges to check and recheck, and the heap operations add in even more wasted time.

Oh, and one other thing. Consider the following:

Farmer John is still searching the underground black market for cows. However, some of the "agents" have set up cars to make travel easier. They all charge varying amounts of money, but some of the newer ones are so desperate for business that *they* actually pay *Farmer John* to use their cars. Since FJ is no longer walking, distance is no longer a concern; find the least expensive route to the best cow.

Which algorithm do you use here? Consider Dijkstra. Suppose you process one vertex with its "minimal" distance, but another later vertex has an edge with negative weight (i.e., you get paid to use the car) that leads to this one. Then the original distance found to this vertex is *not* the least, an assumption which Dijkstra's algorithm relies upon, and the algorithm collapses. That's why whenever you have a shortest path problem with edges with negative weights, you use Floyd-Warshall.

Conclusion? There is none. Well, almost - overall, Floyd-Warshall is preferable. But it really depends on the problem. Below are some challenge problems; half the challenge is figuring out which algorithm to use!

1. (Traditional) Given an undirected, unweighted, connected graph, find the two vertices which are the farthest apart.
2. (Traditional) Given two squares on an NxN chessboard, what is the least number of moves a knight can make to get from one to the other? (A knight moves 2 spaces in one direction, then turns 90 degrees and moves 1 space.)
3. (USACO 2000 Winter Contest, Burch+Cox) The cows hate calisthenics (exercises). They do, however, enjoy jogging a little bit. They have surveyed their pasture and have created a list of all the possible paths in the field. Interestingly, the paths are "directed" – paths can be traversed only in one direction. Sometimes, two different paths connect a pair of points, one path going one way and one path going the other way. Your job is to help the cows find the shortest "circular route" around which they can jog. A circular route is one which starts and ends at the same location. The length of a route is the sum of the lengths of the paths one must traverse to jog around the route.

References

1. Kolstad et al. USACO 2000 Training Camp Notes.
2. John Danaher.

Appendix: Dijkstra with Heap

```
/*
 * This program does Dijkstra's algorithm with a heap. It
 * works on vertices numbered from 1 to n and finds the
 * shortest path between any two specified vertices. Note
 * how long the code is! Practice it once or twice before
 * you ever consider using it in a contest...
 */

#include <fstream.h>

// Maximum input values specified in the problem.
// Note that MAXN is increased by 1!

#define MAXN 1001
#define MAXPATH 2000

/*
 * Everything we need to do Dijkstra:
 *
 * - heap[] is a heap containing the vertices, sorted by
 *      distance to the source in a given path.
 * - index[i] stores the location of value i in the heap.
 * - nadj[i] is the number of vertices connected to node i.
 * - adj[i][j] and dadj[i][j] are the jth element connected
 *      to i and the given distance between i and adj[i][j].
 * - n is the number of vertices in the graph.
 */

const long INF = 1000000000;
int heap[MAXN], index[MAXN];
long dist[MAXN];
int n, nadj[MAXN], *adj[MAXN];
long *dadj[MAXN];

// Standard useful functions.

long min(long a, long b)
{ return ((a < b) ? a : b); }
void swap(int &a, int &b)
{
  int temp = a;
  a = b;
```

```
  b = temp;
}

// Heap operations.

void heapUp(int i)
{
  while (i > 1)
  {
    if (dist[heap[i]] >= dist[heap[i/2]])
      break;
    swap(heap[i], heap[i/2]);
    index[heap[i]] = i;
    index[heap[i/2]] = i/2;
    i /= 2;
  }
}
void heapDown(int i)
{
  int j;
  while (2*i <= heap[0])
  {
    j = i;
    if (dist[heap[i]] > dist[heap[2*i]])
      j = 2*i;
    if (2*i+1 <= heap[0] &&
      dist[heap[j]] > dist[heap[2*i+1]])
      j = 2*i+1;
    if (i==j) break;
    swap(heap[i], heap[j]);
    index[heap[i]] = i;
    index[heap[j]] = j;
    i = j;
  }
}
void heapDeleteTop()
{
  index[heap[1]] = 0;
  index[heap[heap[0]]] = 1;
  heap[1] = heap[heap[0]--];
  heapDown(1);
}

// Let's do some Dijkstra! Finds the shortest path
```

```
// between the source and sink.

long dijkstra(int source, int sink)
{
  int i, curr;
  for (i = 1; i <= n; i++)
  {
    heap[i] = i;
    index[i] = i;
    dist[i] = INF;
  }
  heap[0] = n;
  dist[source] = 0;
  heapUp(source);

  while (1)
  {
    curr = heap[1];
    if (curr == sink) break;
    if (dist[curr] == INF) return -1;

    for (i = 0; i < nadj[curr]; i++)
    {
      dist[adj[curr][i]] =
        min(dist[adj[curr][i]],
            dist[curr] + dadj[curr][i]);
      heapUp(index[adj[curr][i]]);
    }
    heapDeleteTop();
  }

  return dist[sink];
}

int main()
{
  /*
   * Start the tedious task of processing all input data.
   * This saves all paths and distances temporarily, then
   * allocates only as much memory as needed to store all
   * information about the graph.
   */

  int a, b, paths;
```

```
  int temp[MAXPATH][2], count[MAXN];
  long d, tempd[MAXPATH];
  int i, j, k;

  ifstream infile("dijkstra.in");
  infile >> n >> paths;
  infile >> a >> b;
  for (k = 0; k < paths; k++)
  {
    infile >> i >> j >> d;
    nadj[i]++; nadj[j]++;
    temp[k][0] = i; temp[k][1] = j;
    tempd[k] = d;
  }
  infile.close();

  for (k = 1; k <= n; k++)
  {
    count[k] = 0;
    adj[k] = new int[nadj[k]];
    dadj[k] = new long[nadj[k]];
  }
  for (k = 0; k < paths; k++)
  {
    i = temp[k][0]; j = temp[k][1];
    adj[i][count[i]] = j;
    adj[j][count[j]] = i;
    dadj[i][count[i]++] = tempd[k];
    dadj[j][count[j]++] = tempd[k];
  }

  // Reading input is finally finished. Let
  // Dijkstra work his magic.

  ofstream outfile("dijkstra.out");
  outfile << dijkstra(a,b) << endl;
  outfile.close();

  return 0;
}
```