Shortest Paths

Sample Problem: Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a `perfect' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the `worst' point in the maze (the point that is `farther' from either exit when walking optimally to the closest exit).

Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|     | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

The Abstraction

Given:

- A directed graph with nonnegative weighted edges
- A path between two vertices of a graph is any sequence of adjacent edges joining them
- The shortest path between two vertices in a graph is the path which has minimal cost, where cost is the sum of the weights of edges in the path.

Problems often require only the cost of a shortest path not necessarily the path itself. This sample problem requires calculating only the costs of shortest paths between exits and interior points of the maze. Specifically, it requires the maximum of all of those various costs.

Dijkstra's algorithm to find shortest paths in a weighted graph

Given: lists of vertices, edges, and edge costs, this algorithm `visits' vertices in order of their distance from the source vertex.

- Start by setting the distance of all notes to infinity and the source's distance to 0.
- At each step, find the vertex u of minimum distance that hasn't been processed already. This vertex's distance is now frozen as the minimal cost of the shortest path to it from the source.
- Look at appending each neighbor v of vertex u to the shortest path to u. Check vertex v to see if this is a better path than the current known path to v. If so, update the best path information.

In determining the shortest path to a particular vertex, this algorithm determines all shorter paths from the source vertex as well since no more work is required to calculate *all* shortest paths from a single source to vertices in a graph.

Reference: Chapter 25 of [Cormen, Leiserson, Rivest]

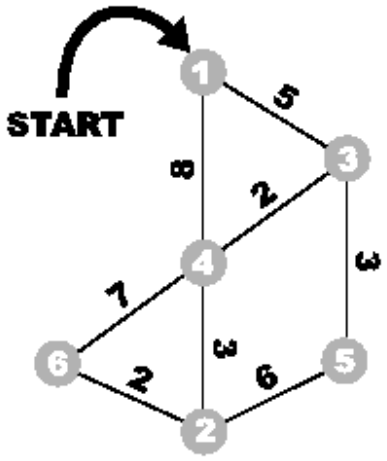
Pseudocode:

```
# distance(j) is distance from source vertex to vertex j
# parent(j) is the vertex that precedes vertex j in any shortest path
#                  (to reconstruct the path subsequently)

 1 For all nodes i
 2     distance(i) = infinity          # not reachable yet
 3     visited(i) = False
 4     parent(i) = nil # no path to vertex yet

 5 distance(source) = 0 # source -> source is start of all paths
 6 parent(source) = nil
 7   8 while (nodesvisited < graphsize)
 9     find unvisited vertex with min distance to source; call it vertex i
10     assert (distance(i) != infinity, "Graph is not connected")

11     visited(i) = True # mark vertex i as visited

    # update distances of neighbors of i
12     For all neighbors j of vertex i
13         if distance(i) + weight(i,j) < distance(j) then
14             distance(j) = distance(i) + weight(i,j)
15             parent(j) = i
```

Running time of this formulation is O(V^2). You can obtain O(E log V) (where E is the number of edges and V is the number of vertices) by using a heap to determine the next vertex to visit, but this is considerably more complex to code and only appreciably faster on large, sparse graphs.

Sample Algorithm Execution

Consider the graph below, whose edge weights can be expressed two different ways:

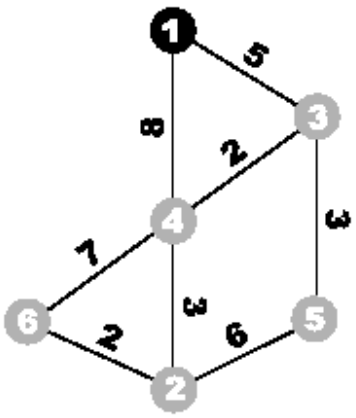


Edge	Weight
(1, 3)	5
(1, 4)	8
(3, 4)	2
(3, 5)	3
(4, 2)	3
(4, 6)	7
(5, 2)	6
(2, 6)	2

1 2 3 4 5 6
1 0 0 5 8 0 0
2 0 0 0 3 6 2
3 5 0 0 2 3 0
4 8 3 2 0 0 7
5 0 6 3 0 0 0
6 0 2 0 7 0 0

Here is the initial state of the program, both graphically and in a table:

		Distance to	
Node	Visited	Source	Parent
1	T	0	nil
2	F	infinity	nil

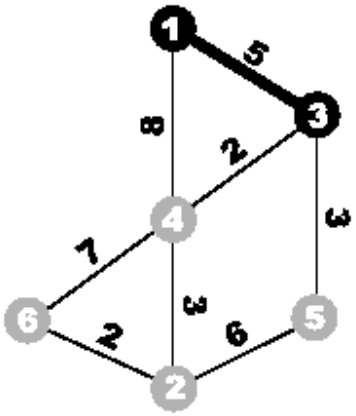


3	F	infinity	nil
4	F	infinity	nil
5	F	infinity	nil
6	F	infinity	nil

Updating the table, node 1's neighbors include nodes 3 and 4.

Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	F	**5**	**1**
4	F	**8**	**1**
5	F	infinity	nil
6	F	infinity	nil

Node 3 is the closest unvisited node to the source node (smallest distance shown in column 3), so it is the next visited:

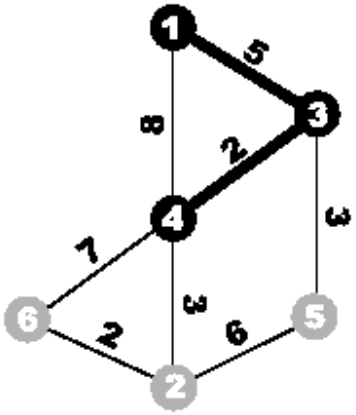


Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	**T**	5	1
4	F	8	1
5	F	infinity	nil
6	F	infinity	nil

Node 3's neighbors are nodes 1, 4, and 5. Updating the unvisited neighbors yields:

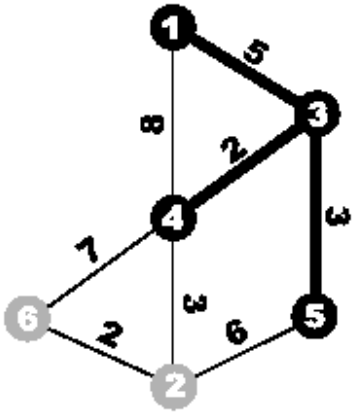
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	T	5	1
4	F	**7**	**3**
5	F	**8**	**3**
6	F	infinity	nil

Node 4 is the closest unvisited node to the source. Its neighbors are 1, 2, 3, and 6, of which only nodes 2 and 6 need be updated, since the others have already been visited:



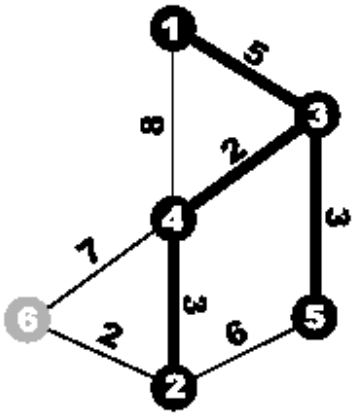
		Distance to	
Node	Visited	Source	Parent
1	T	0	nil
2	F	**10**	**4**
3	T	5	1
4	**T**	7	3
5	F	8	3
6	F	**14**	**4**

Of the three remaining nodes (2, 5, and 6), node 5 is closest to the source and should be visited next. Its neighbors include nodes 3 and 2, of which only node 2 is unvisited. The distance to node 2 via node 5 is 14, which is longer than the already listed distance of 10 via node 4, so node 2 is not updated.



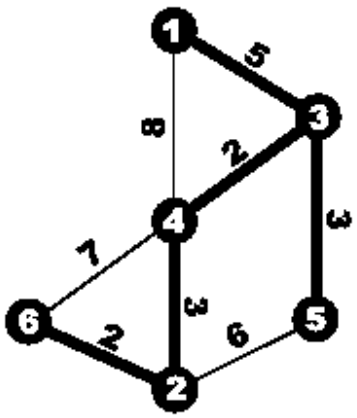
		Distance to	
Node	Visited	Source	Parent
1	T	0	nil
2	F	10	4
3	T	5	1
4	T	7	3
5	**T**	8	3
6	F	14	4

The closest of the two remaining nodes is node 2, whose neighbors are nodes 4, 5, and 6, of which only node 6 is unvisited. Furthermore, node 6 is now closer, so its entry must be updated:



		Distance to	
Node	Visited	Source	Parent
1	T	0	nil
2	**T**	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	F	**12**	**2**

Finally, only node 6 remains to be visited. All of its neighbors (indeed the entire graph) have now been visited:



Node	Visited	Distance to Source	Parent
1	T	0	nil
2	T	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	**T**	12	2

Sample Problem: Package Delivery

Given a set of locations, lengths of roads connecting them, and an ordered list of package dropoff locations. Find the length of the shortest route that visits each of the package dropoff locations in order.

Analysis: For each leg of the required path, run Dijkstra's algorithm to determine the shortest path connecting the two endpoints. If the number of legs in the journey exceeds N, instead of calculating each path, calculate the shortest path between all pairs of vertices, and then simply paste the shortest path for each leg of the journey together to get the entire journey.

Extended Problem: All Pairs, Shortest Paths

The extended problem is to determine a table a, where:
$a_{i,j}$ = length of shortest path between i and j, or infinity if i and j aren't connected.

This problem is usually solved as a subproblem of a larger problem, such as the package delivery problem.

Dijkstra's algorithm determines shortest paths for one source and all destinations in $O(N^2)$ time. We can run it for all N sources in $O(N^3)$ time.

If the paths do not need to be recreated, there's an even simpler solution that also runs in $O(N^3)$ time.

The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm finds the length of the shortest paths between all pairs of vertices. It requires an adjacency matrix containing edge weights, the algorithm constructs optimal paths by piecing together optimal subpaths.

- Note that the single edge paths might not be optimal and this is okay.
- Start with all single edge paths. The distance between two vertices is the cost of the edge between them or infinity if there is no such edge.
- For each pair of vertices u and v, see if there is a vertex w such that the path from u to v through w is shorter than the current known path from to u to v. If so, update it.
- Miraculously, if ordered properly, the process requires only one iteration.

- For more information on why this works, consult Chapter 26 of [Cormen, Leiserson, Rivest].

Pseudocode:

```
# dist(i,j) is "best" distance so far from vertex i to vertex j

# Start with all single edge paths.
For i = 1 to n do
    For j = 1 to n do
        dist(i,j) = weight(i,j)

For k = 1 to n do # k is the `intermediate' vertex
    For i = 1 to n do
        For j = 1 to n do
            if (dist(i,k) + dist(k,j) < dist(i,j)) then # shorter path?
                dist(i,j) = dist(i,k) + dist(k,j)
```

This algorithm runs in $O(V^3)$ time. It requires the adjacency matrix form of the graph.

It's very easy to code and get right (only a few lines).

Even if the solution requires only the single source shortest path, this algorithm is recommended, provided the time and memory and are available (chances are, if the adjacency matrix fits in available memory, there is enough time).

Problem Cues

If the problem wants an optimal path or the cost of a minimal route or journey, it is likely a shortest path problem. Even if a graph isn't obvious in a problem, if the problem wants the minimum cost of some process and there aren't many states, then it is usually easy to superimpose a graph on it. The big point here: shortest path = search for the minimal cost way of doing something.

Extensions

If the graph is unweighted, the shortest path contains a minimal number of edges. A breadth first search (BFS) will solve the problem in this case, using a queue to visit nodes in order of their distance from the source. If there are many vertices but few edges, this runs much faster than Dijkstra's algorithm (see Amazing Barn in Sample Problems).

If negative weight edges are allowed, Dijkstra's algorithm breaks down. Fortunately, the Floyd-Warshall algorithm isn't affected so long as there are no negative cycles in the graph (if there is a negative cycle, it can be traversed arbitrarily many times to get ever `shorter' paths). So, graphs must be checked for them before executing a shortest path algorithm.

It is possible to add additional conditions to the definition of shortest path (for example, in the event of a tie, the path with fewer edges is shorter). So long as the distance function can be augmented along with the comparison function, the problem remains the same. In the example above, the distance function contains two values: weight and edge count. Both values would be compared if necessary.

Sample Problems

Graph diameter

Given: an undirected, unweighted, connected graph. Find two vertices which are the farthest apart.

Analysis: Find the length of shortest paths for all pairs and vertices, and calculate the maximum of these.

Knight moves

Given: Two squares on an NxN chessboard. Determine the shortest sequence of knight moves from one square to the other.

Analysis: Let the chessboard be a graph with 64 vertices. Each vertex has at most 8 edges, representing squares 1 knight move away.

Amazing Barn (abridged) [USACO Competition Round 1996]

Consider a very strange barn that consists of N stalls (N < 2500). Each stall has an ID number. From each stall you can reach 4 other stalls, but you can't necessarily come back the way you came.

Given the number of stalls and a formula for adjacent stalls, find any of the `most central' stalls. A stall is `most central' if it is among the stalls that yields the lowest average distance to other stalls using best paths.

Analysis: Compute all shortest paths from each vertex to determine its average distance. Any $O(N^3)$ algorithm for computing all-pairs shortest paths would be prohibitively expensive here since N=2500. However, there are very few edges (4 per vertex), making a BFS with queue ideal. A BFS runs in O(E) time, so to compute shortest paths for all sources takes O(VE) time - about:
$2{,}500 \times 10{,}000 = 2.5 \times 10^7$ things, much more reasonable than $2500^3 = 1.56 \times 10^{10}$

Railroad Routing (abridged) [USACO Training Camp 1997, Contest 1]

Farmer John has decided to connect his dairy cows directly to the town pasteurizing plant by constructing his own personal railroad. Farmer John's land is laid out as a grid of one kilometer squares specified as row and column.

The normal cost for laying a kilometer of track is $100. Track that must gain or lose elevation between squares is charged a per-kilometer cost of $100 + $3 x meters_of_change_in_elevation. If the track's direction changes 45 degrees within a square, costs rise an extra $25; a 90 degree turn costs $40. All other turns are not allowed.

Given the topographic map, and the location of both John's farm and the plan, calculate the cost of the cheapest track layout.

Analysis: This is almost a standard shortest path problem, with grid squares as vertices and rails as edges, except that the direction a square is entered limits the ways you can exit that square. The problem: it is not possible to specify which edges exist in advance (since the path matters).

The solution: create eight vertices for each square, one for each direction you can enter that square. Now you can determine all of the edges in advance and solve the problem

as a shortest path problem.

Back to USACO Training Gateway | Comment or Question