Dijkstra's Algorithm

Evan Travers

April 2, 2010

Experiment 1:

Problem:
Given the graph supplied, find the shortest paths between each of the following vertices in the graph.

Solution:
Having implemented Dijkstra's algorithm, I wrote a driver program that went through the results of a Dijkstra for a given Node and output just the values for the target nodes as a CSV file, which I then put into this table.

	1	18	29	126	212	272	289	336
1	0.0	18.238632	6.2477917	2.1983675	13.104577	16.122883	4.8217029	9.2255903
18	18.2386	0.0	11.99084	16.984634	5.1340551	2.1157485	15.128843	9.0708296
29	6.24779	11.99084	0.0	4.9937937	6.856785	9.8750916	3.1380027	3.0247328
126	2.19837	16.984634	4.9937937	0.0	11.850579	14.868885	2.8006704	7.9715923
212	13.1046	5.1340551	6.856785	11.850579	0.0	3.0183066	9.9947877	4.6503949
272	16.1229	2.1157485	9.8750916	14.868885	3.0183066	0.0	13.013094	6.9550811
289	4.8217	15.128843	3.1380027	2.8006704	9.9947877	13.013094	0.0	6.1158013
336	9.22559	9.0708296	3.0247328	7.9715923	4.6503949	6.9550811	6.1158013	0.0

The Dijkstra I coded is contained in the appendix 1, but the pseudocode is below:

```
set distance of all nodes to infinite
set distance of start node to 0

create a priority queue of nodes weighed by their distance computed
add the start to queue

while queue not empty
      // the queue is sorted by weight, so the lightest value is lifted.
      current = poll the queue

      getneighbors
            // my modified getneighbors contains the bulk of the algorithm
            get the neighbors of current
                  if not visited
                        add to queue
                        compute its distance, if its better than computed,
                              update distance

      set current as visited
return the distances
```

Experiment 2:

Problem:
Assuming that each of these vertices lie along a minimal, non-branching, non-cyclical path through the graph, in what order do they appear on the path? This will require finding, for each vertex in the list, the two other vertices closest to it (that haven't already been seen). Hint: start at vertex 1.

Solution:
1, 126, 289, 29, 336, 212, 272, 18

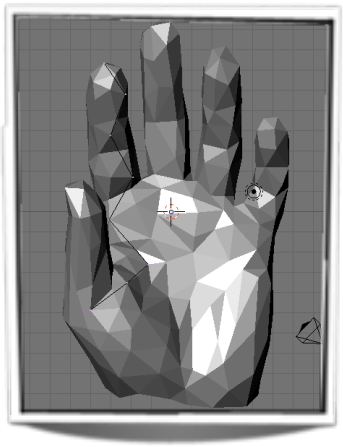
Explanation:

To arrive at this conclusion, I tested two methods. For the first, I simply looked at the table I computed in the first experiment, and wrote down the nodes in the order that they appeared from node 1. This gave the above result. I was not yet sure if this was the correct solution, so I wrote a method called dd(), which is also in the appendix, but the pseudocode is below:

```
create a todo list of target nodes

// start at 1
current equals 1

// until the path is traversed
while todo not empty

      // get the weight of every node
      run dijkstra on current

      // for the remaining target nodes
      for each node in todo

            // find the weight of each
            // of the nodes to pass through
            get value from dijkstra results, put in priority queue

      // get the lightest node, mark it as passed, continue
      remove the lightest node from the queue
      remove that node from todo, set as current
      put current in output

// return list of nodes in order
return output
```

This “Dijkstra on top of a Dijkstra,” as I consider it, although overkill, not surprisingly returns the same order as the first method.

Bonus:



Problem:
The graph has a secret. What is it? How did you find it? Visually demonstrate it. How does the path found above relate to the graph as a whole?

Solution:
The graph is derived from a 3-D model of a hand, and the path we found in the previous two problems follows a line from the thumb to the tip of the index finger. (Larger image in appendix.)

Explanation:

This was my favorite part of the whole programming assignment. Possibly my favorite programming assignment I've ever done at UAB. Thanks for the puzzle.

When you told us during lab that there was a secret in the graph, Justin Marbutt and I looked at each other, and immediately started digging. I started getting suspicious when I noticed your graph building routine referred to vertices, edges, and faces. I used to dabble in 3-D modeling, and I recognized these as terms from 3-D, not from graphs. I started looking at the .off file, and realized that nearly all the lines were of floats, always in groups of three for x,y,z. I looked you up, and discovered you had done work at UAB for VR. I was certain that there was a 3-D model hidden in the graph... and the clincher is when I FINALLY googled .off file format. It was then a race to find a program that would open the file. I immediately launched my copy of Blender. I had a hunch that you being the python guru that you are, would choose a format Blender could open. And I was right. I opened the file, and rendered an image just under five minutes after you gave the assignment. It was great. For fun, I added multi-res and set the faces to smooth and ran off a quick render for the front-piece of the report.

The line of nodes itself took a little more work. I read up on the .off format, and as soon as I realized where the vertices were declared, I started eliminating all the nodes other than the eight target nodes. The first time, I forgot about zero-indexing and got a very strange result, but when I realized my mistake, I was able to make a second .off file with only the eight nodes I needed. I imported this to blender, and then made edges between them to make the path more apparent.

Appendix 1: Code

```
     public void getNeighbors(Node node, PriorityQueue<Node> neighbors) {
        int degree = adjacency.get(node.get()).size();
             Double newDistance;
        for(int i=0;i<degree;i++) {
            int neighbor = adjacency.get(node.get()).get(i);
            if (neighbor < 0)
                neighbor = -(neighbor + 1);
                    // CHANGED fix this
                    if (!getVisited(neighbor)) {
                           newDistance = getDistance(node.get(), neighbor)+node.weight
();
                           neighbors.add(new Node(neighbor,newDistance));
                           // if the new distance is better than the old
                           if (distance.get(neighbor)>newDistance) {
                                  // replace recorded distance
                                  distance.set(neighbor,newDistance);
                           }
                    }

        }
    }
     public Vector<Double> d(int start) {
             clearVisited();
             // set the distances properly
             for (int i=0;i<visited.size() ;i++ ) {
                    distance.set(i,Double.POSITIVE_INFINITY);
             }
             // set origin to be 0.0
             distance.set(start,0.0);

             // last bit of setup
             PriorityQueue<Node> queue = new PriorityQueue<Node>();
             Node curr = new Node(start,0.0);
             queue.add(curr);

             // CHANGED finish the loop
             while (queue.size()>0) {
                    // consider the one on the top
                    curr = queue.poll();
                    // get all the neighbors, and if they haven't been visited, add
                    them to the queue.
                    // this modified get neighbors also replaces the recorded weight,
                    if it is lower.
                    getNeighbors(curr,queue);
                    setVisited(curr.get());
             }

             return distance;
     }

     public Vector<Integer> dd() {
             Vector<Integer> output = new Vector<Integer>();
             // create a list of the answers
             ArrayList<Integer> todo = new ArrayList<Integer>();
             for (int i=0;i<answers.length ;i++ ) {
                    todo.add(answers[i]);
             }
             // call dijkstra on the first answer, looking for only the remaining
             answers
             int currentValue=1;
```

```
                while (todo.size()>0) {
                        Vector<Double> results = new Vector<Double>();
                        results = d(currentValue);
                        // System.out.println(results);
                        // make a queue of just the results we want
                        PriorityQueue<Node> valuesFound = new PriorityQueue<Node>();
                        for (int i=0;i<todo.size();i++) {
                                // load all the values we are looking for into the queue to
                                be sorted
                                valuesFound.add(new Node(todo.get(i),results.get(todo.get
(i))));
                        }
                        // take the shortest path off, repeat
                        // remove the one you just found
                        currentValue=valuesFound.poll().get();
                        todo.remove(todo.indexOf(currentValue));
                        output.add(currentValue);
                }

                return output;
        }
```

Appendix 2: