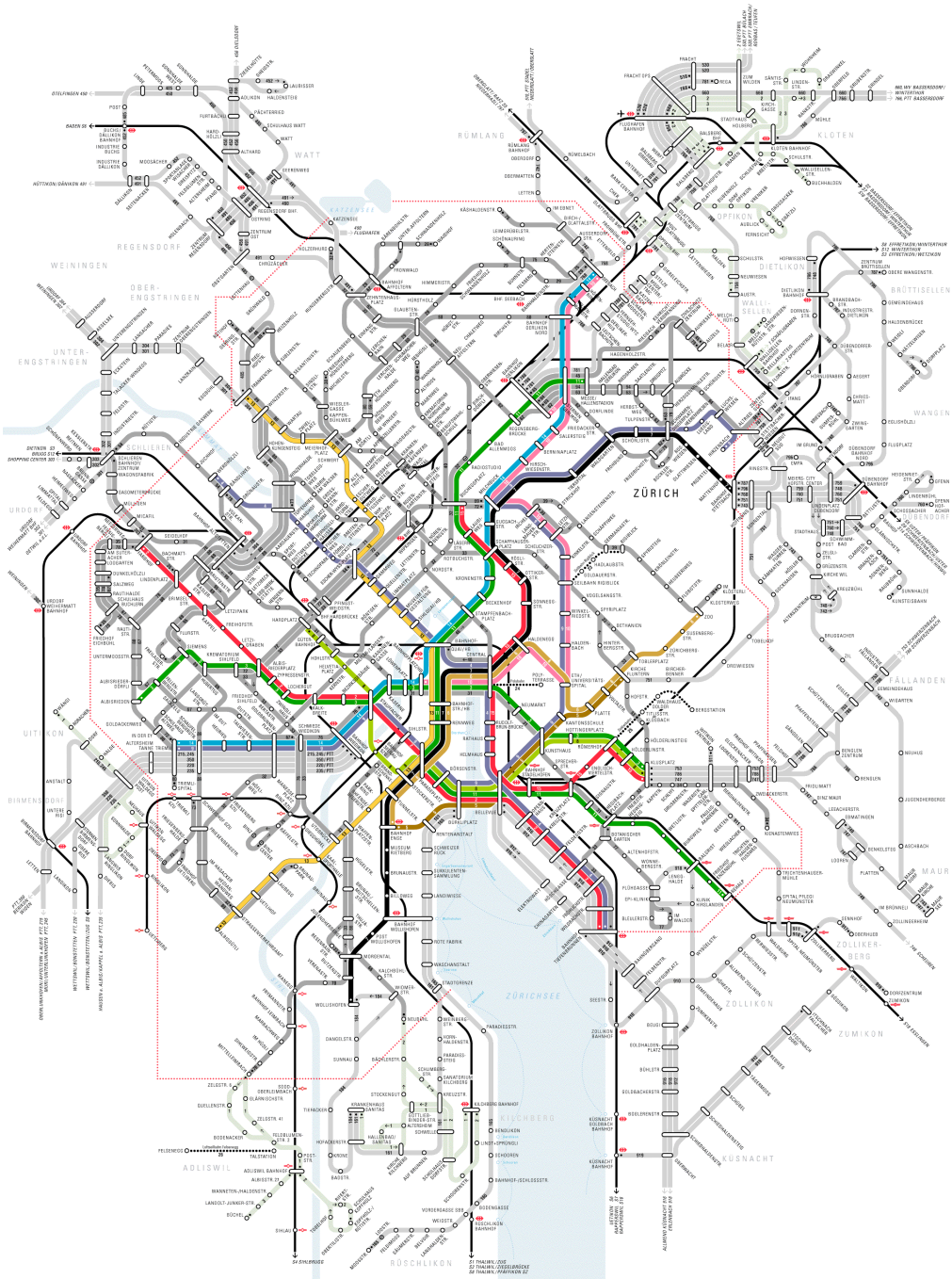
Prim's Algorithm and Maps

Evan Travers

April 17, 2010



Purpose:

To implement Prim's Algorithm in order to find the acyclical minimal spanning tree of a graph.

Introduction:

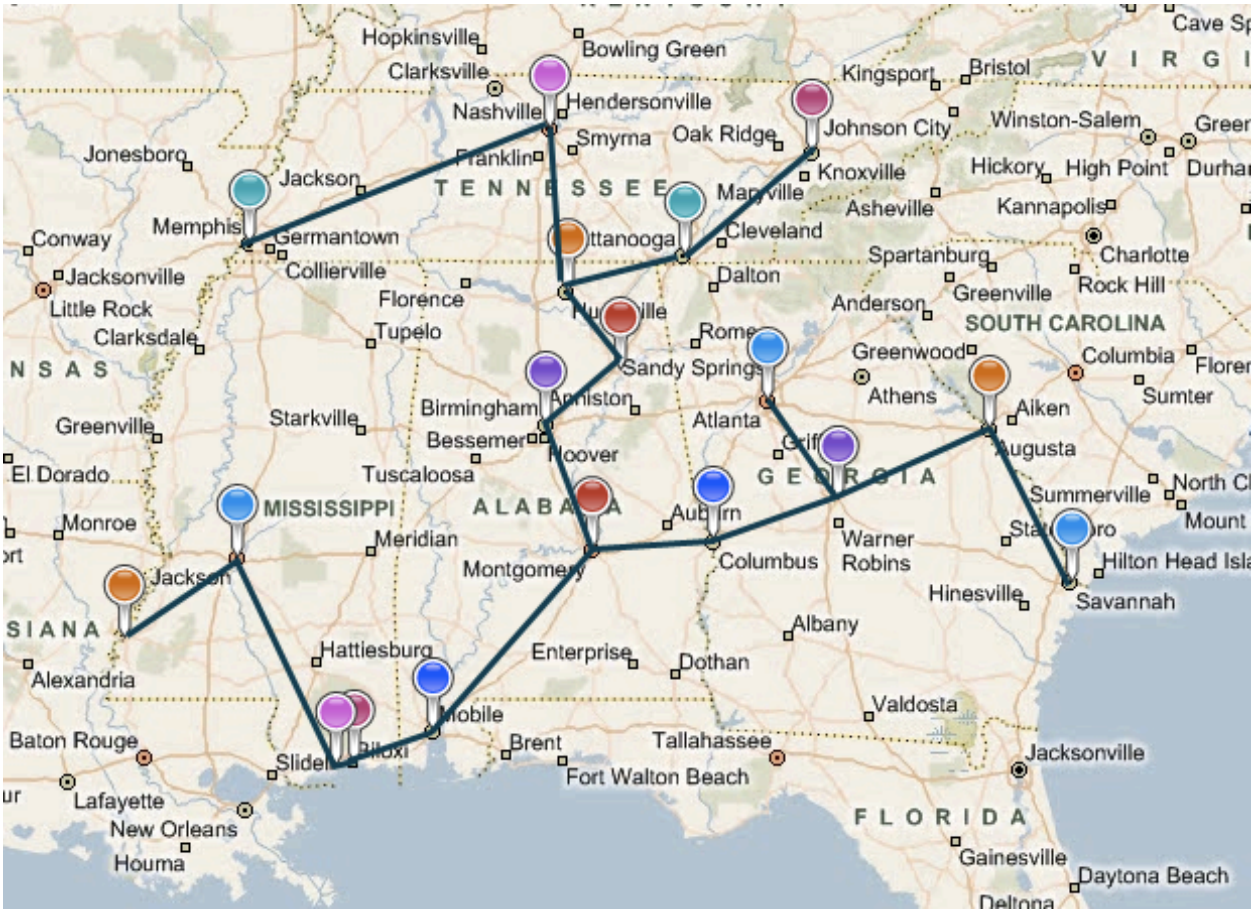
This program went through several different phases for me. The first iteration loaded all possible edges into a priority queue, then tried to just pick out the necessary edges.This was pretty close, but didn't make a full connected tree. Through this build, I made the Edge class I was to use throughout the rest of the program. I kept evolving the program until I realized I only needed to consider possible edges from my current position, not all edges. I then completely rewrote my code, trying to match Prim's pseudocode, and it worked perfectly that time.

Results:

Starting from Atlanta, (though my code doesn't care where you start.)

Atlanta, GA Macon, GA 76.0
Macon, GA Columbus, GA 83.0
Columbus, GA Montgomery, AL 76.0
Montgomery, AL Birmingham, AL 84.0
Birmingham, AL Gadsden, AL 57.0
Gadsden, AL Huntsville, AL 59.0
Huntsville, AL Chattanooga, TN 75.0
Huntsville, AL Nashville, TN 99.0
Chattanooga, TN Knoxville, TN 100.0
Macon, GA Augusta, GA 105.0
Augusta, GA Savannah, GA 108.0
Montgomery, AL Mobile, AL 154.0
Mobile, AL Biloxi, MS 54.0
Biloxi, MS Gulfport, MS 12.0
Gulfport, MS Jackson, MS 148.0
Jackson, MS Natchez, MS 87.0
Nashville, TN Memphis, TN 196.0

On a map:



Conclusions:

The above tree is the tree with the lightest weight according to the data in Cities.java. When I initially mapped it out, I was slightly perturbed as there appear to be some errors, such as that it would seem that the distance between Nashville and Memphis is greater than the distance from Chattanooga and Memphis, but the data in the distance table tells my algorithm that the distance is in fact less, and that the path is indeed the correct one.

As to whether the tree I have created follows actual highway information, there are highways on the paths that I found. There differ slightly in some places, but it is pretty accurate. In fact, to test my map, I asked Google maps and Bing maps how to get from one location to another on some of the more interesting paths, such as Chattanooga to Memphis. In each instance, it always followed the path I mapped out using my Prim's. Such paths were: Mobile to Natchez, Macon to Savannah, and Gadsen to Knoxville.

Appendix A: Psuedocode

visited = empty list of nodes that have been visited
not_visited full list of nodes that have not been visited

while visited length != not_visited length
    add all possible edges that start in visited and end in not_visited to priority queue
    pop one off the top
        add the edge to the result
        set its nodes as visited
    clear the queue

Appendix B: Code

```
public class Prim {
     private static Cities c;

     public Prim(Cities cityGraph) {
             c=cityGraph;
     }

     public ArrayList<Edge> search(int begNode) {
             // build list of edges for the result
             ArrayList<Edge> result = new ArrayList<Edge>();

             // you have a list of nodes that have been visited
             ArrayList<Integer> done = new ArrayList<Integer>();

             // populate the todo
             ArrayList<Integer> todo = new ArrayList<Integer>();
             for (int i=0;i<18 ;i++ ) {
                    todo.add(i);
             }

             // stick the start node into done, remove from todo
             done.add(begNode);
             todo.remove(begNode);

             // this will be used for the getting of options
             PriorityQueue<Edge> edges = new PriorityQueue<Edge>();

             // until all the nodes are in done
             while (done.size()!=18) {
                    // System.out.println("todo:                    "+todo);
                    // System.out.println("done:                    "+done);
                    // System.out.println("current result: \t"+result);
                    // System.out.println("");
                    // get the options currently available, starting at done nodes
                    // and ending on "todo" nodes, and load them into a priority queue
                    for (int i=0;i<done.size() ;i++ ) {
                           int start = done.get(i);
                           for (int j=0;j<todo.size() ;j++ ) {
                                  int end = todo.get(j);
                                  // System.out.println(" edge we are adding is " +
start + " to " + end);
                                  edges.add(new Edge(c, start, end));
                           }
                    }

                    // take the one off the top, add it to result
```

```
                        Edge winner = edges.poll();
                        // System.out.println("The winner is: " + winner);
                        result.add(winner);

                        // set it's new node as visited
                        done.add((Integer)winner.end());
                        todo.remove((Integer)winner.end());

                        // clear the edges at the end
                        edges.clear();
                }

                return result;
        }

        public static String printmap(ArrayList<Edge> edges) {
                //ArrayList<Edge> edges2 = Collection.sort(edges);
                String result = "";
                for (int i=0;i<edges.size() ;i++ ) {
                        result+= c.getName(edges.get(i).ind(1)) + " " + c.getName
(edges.get(i).ind(2)) + " " + edges.get(i).weight() + "\n";
                }
                return result;
        }
```