Matthew A. Krieger

November 17, 2015

CS 253: Data & File Structures

Final Project

Neli P. Zlatareva, Ph.D.

Final Project

Given the task of finding a minimum spanning tree, Prim’s algorithm is one of the simplest and most intuitive approaches to the problem. For a minimum spanning tree problem, you are concerned with coming up with a tree that minimizes the total weight and contains all of the vertices. Minimum spanning trees are often used to construct a network, where you may want to minimize the total amount of wire or cable used. Given an undirected, weighted connected graph, Prim’s algorithm will produce a minimum spanning tree. Prim’s is a greedy algorithm which “is based on choosing objects to join a growing collection by iteratively picking an object that minimizes some cost function” (Goodrich, 662). The basic idea of Prim’s algorithm, is that you “begin with some vertex s, defining the initial ‘cloud’ of vertices C. Then, in each iteration, we choose a minimum-weight edge e = (u,v), connecting a vertex u in the cloud C to a vertex v outside of C. The vertex v is then brought into the cloud C and the process is repeated until a spanning tree is formed” (Goodrich, 664).

Based on the way the minimum spanning tree is constructed with Prim’s algorithm, it seemed like a natural choice for our application which models the spread of an infectious disease around the world. We wanted the order in which vertices were added to the minimum spanning tree to represent the progression of how the disease spread. We decided to model the infection and how it spread throughout the world using 3 main variables; the country of origin, the rate of infection and the number of days to be simulated. However, due to the inability to obtain complete demographical information from all countries, we knew we would be unable to model this as precisely as we would've liked to. The necessary data and variables used in our program included; populations, infection rates, rate of travel, rate of interactions, combined to create a mathematical equation used to represent the spread of an infection. In addition to these factors, we chose wisely when establishing the foundation of our program and considered that we would not have time to create an adjacency matrix containing data for every airport, in every country, in the world.

As a result we based our program on the simple information that can be seen in the table above. We determined that if we could make a program to model the spread of an infectious disease throughout just 10 countries then it could be later revised to include either more countries and/or more accurate information. To choose our ten countries we referenced the top ten countries listed according to the 2013 National Power Index (NPI). These countries can be seen listed above in ascending order. As a result of the problem at hand, we deemed it most realistic to establish a connection between each and every vertex (without any loops). Our reasoning behind doing so was that in real life you have the ability to (theoretically) travel from any one airport to another, and this is why each vertex has a degree of 9. The next column in the table is labeled IATA, short for International Air Transport Association. In this column you have a reference to the IATA airport code for the airport that we chose in each country. We chose the biggest airports in each of our ten countries, based on passenger traffic from 2013 public records. The next two columns are the longitude and latitude coordinates (as determined by Google Maps) for each of the ten airports. It was these values that are used in our calculations of the weight (distances) between two vertices (airports). In the last column you can see the population for each country, estimated in millions (as determined by public online records).

Since we are assuming direct flights are possible from any location, if we have V vertices, there are (V-1)2 edges, making it O(V2) efficiency. Due to the density of our graph, the disadvantages of an adjacency matrix implementation are minimal, so we chose to use an adjacency matrix implementation. This allows for a simple implementation of Prim’s algorithm. We use two arrays vertex arrays, and the adjacency matrix. One array holds all the vertices with the array index corresponding to the adjacency matrix row or column. The second array holds the vertices that are currently in the MST. So, starting with a root vertex, we add it to a list of vertices in the MST. Next, we search the adjacency matrix with the root vertex’s index, for the edge with the minimum weight. We use the index of the edge with the minimum weight to obtain the corresponding vertex from the vertex array, and we add that vertex to the MST array (and set a boolean variable called visited for that vertex to true, which allows us to ensure while finding minimum a minimum edge that it is to a vertex not yet in the MST). Next, we search columns of the adjacency matrix corresponding to vertices in the MST array, for a minimum edge. While finding the minimum edge, we also check that the edge is between a vertex in the MST, and one that is not. Once an eligible minimum edge has been found, we add the corresponding vertex to the MST array. This process repeats until all vertices are in the MST array. So as the MST grows, more columns are being searched for a minimum edge, but fewer rows within those columns represent eligible edges, because we can ignore edges to already visited vertices.

*Ineligible edge (red), eligible edge (white), minimum edge found (yellow)*

The table above shows the conditions the algorithm considers when all vertices but one are in the MST array. The loop searches each column (corresponding to a vertex in the MST) for a minimum edge to a vertex not in the MST. The red cells represent the edges that can be ignored because they are to vertices already in the tree, the white cells represent eligible edges, and the yellow cell shows the edge found to be the minimum. This does however, illustrate the potential inefficiencies, because the first eligible edge is the minimum but there is no way of knowing that without searching the remaining edges. Additionally, while the red cells represent ineligible edges, the while loop still goes to every one of them and ignores them as a possible minimum edge only after locating the vertex that the row/column corresponds to, and determining that it is already in the MST tree (via the visited boolean variable of the vertex).

*Geographical representation of the MST constructed from Prim’s Algorithm*

After we implemented Prim’s algorithm, the biggest problem we encountered was generating an equation that would reference and change variables associated with specific vertices (countries) in a realistic fashion. Admittedly, although we did introduce randomization to provide variation, the numbers produced by our application could be much more accurate. Our current equation has two distinct operations, the first occurring on day 1 of the infection and the second repeating itself recursively until we reach the total number of days originally input by the user. Due to these two operations we knew from the start that our program was not going to be modifying data, based on the order of its traversal, after day 1. However, we concluded this to be sufficient enough for our elementary implementation of such a complex application. Once the infected population(s), based off the order traversed, were calculated for day 1 our program used a static equation (with some randomized variables) to recursively increment them based on predetermined factors like interaction rate, percent of population currently infected, and other variables affected directly by values stored in each of the vertices. Unfortunately it was only post implementation that we were presented with a far superior method. It was only then that we came across a publication regarding The Differential Equation Model1, which is recognized by the Mathematical Association of America to be a plausible model for the spread of an infection. It was at this point that we realized we were not only not calculating for individuals that may have died, but also for those that may have recovered. Ideally, (in retrospect) we should have used a model, like The Differential Equation Model, to calculate the spread of our infection. With that being said, we knew that its complexity, in terms of the number of variables used, was far beyond the scope of this class. Therefore we opted to continue using our inferior methodology, rather than add more numbers and variables to the equation. In addition to this flaw, I also felt that to truly model the spread of an infection one would need to take into account those who may be infected and traveling- not only from each and every country, but for each and every day as well. As a result, logically speaking, I believe it would be most sensible to update the infection information based the traversal obtained by Prim’s algorithm using every vertex as the root, one time per day. Theoretically this would allow us to model the amount of infection that is spread from every country each day, based on their orientation of their surrounding countries, rather than just one time, for one root, on day one. It is important to point out the problems we are aware of prior to showing you our output, which allows me to better explain our results at this stage in the application.

**Day 1 Output (root: USA)**

As discussed above, on day 1 our program makes its traversal (using Prim's algorithm) and infects every vertex depending on its distance from the origin. To do this we used the equation *long temp = Math.round((this.infected \* percentTravel \* rate)/(double)d);* The equation begins by using infected, the number of people infected in said Vertex, and multiplying it by the percentTravel (1.0032 estimated daily), and then multiplying once more by the user input rate of the infection. Finally, in order to have the distance away from the origin factor into the amount infected we divided our product by a value which increased, by one, each time a vertex was visited. This meant that the last country the infection reaches will have its product, for the number of population infected, divided by 10 because it was the final vertex visited via Prim's algorithm. Another notable fact in this method specifically is that if the population infected returns lower than 1000 our program generates a new integer instead between 500 and 1,500. Through many debugging tests we realized that some of our countries, as a result of having very low infected populations after day 1, increased extremely too slow for our liking and this is why we chose to artificially impose a randomized integer value here, only when necessary. These results, after just one day, have quite the variation when compared with those of multiple days.

This is our output for 365 days (1 year) with a root set at the United States of America. The random values have done their job, as you can see reflected in the percent infected. We had an issue in the early versions of our application in which the percentage for each country was identical, but this was fixed using a few additional parameters in the equation along with random variables. In the end our equation for calculating infected population after multiple days was *double temp = (((this.interactionRate \* percentInfected +(Math.log(day))/(double)10)) \* this.population \* rate);* As you can clearly see, this equation is much different than the one used for day 1 infection. To begin we take a constant variable, interaction rate and multiply it by the percent infected. The interaction rate is meant to simulate the approximate percentage of the population that your infection would come in contact with, per day. For the purpose of our program we chose to use an infection rate that was .01% of the total population, assuming that was close to accurate. To put that number into perspective, in the United States we estimate you would have the ability to infect (via sneezing, coughing, etc.) 31,600 people. This is then multiplied by the percent infected, simply infected divided by population, plus the log of the day number. The reason we chose to include a logarithmic function was that it guaranteed us that, not only would we never exceed 100% of the population infected, but we also would see a decline in the spread of the disease over time. This product was divided by 10.0 to produce an integer in reasonable range for total infected per day. The result was finally stored into a variable called *temp* which we then opted to use to generate a random integer in the range of *(temp / 2)* and *temp*. In the end, the number returned by this method was added to that vertices already infected population- thus producing the total infected population for each vertex, day by day. Aside from the particulars of our application, our way of implementing Prim’s algorithm is one of several options.

Robert Sedgewick and Kevin Wayne outline a general approach to Prim’s algorithm, using what they call the “lazy implementation”:

“We use a priority queue to hold the crossing edges and find one of minimal weight. Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex. But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes *ineligible* (it is no longer a crossing edge because it connects two tree vertices). The lazy implementation leaves such edges on the priority queue, deferring the ineligibility test to when we remove them.” (http://algs4.cs.princeton.edu/43mst/)

They go on to introduce what they call the “*eager implementation”:*

“To improve the lazy implementation of Prim's algorithm, we might try to delete ineligible edges from the priority queue, so that the priority queue contains only the crossing edges. But we can eliminate even more edges. The key is to note that our only interest is in the *minimal* edge from each non-tree vertex to a tree vertex. When we add a vertex v to the tree, the only possible change with respect to each non-tree vertex w is that adding v brings w closer than before to the tree. In short, we do not need to keep on the priority queue all of the edges from w to vertices tree—we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update that minimum (because of an edge v-w that has lower weight), which we can do as we process each edge in s adjacency list. In other words, we maintain on the priority queue just one edge for each non-tree vertex: the shortest edge that connects it to the tree.” (http://algs4.cs.princeton.edu/43mst/)

While there is minimal wasted space in our adjacency list due to our assumption of direct flights to any vertex, if we changed this assumption or added more vertices it would be worth reconsidering how our graph is represented internally. If this application were to be truly useful, it would need to involve much more complex mathematics and possibly a different algorithm altogether. That being said, Prim’s algorithm provided us a simple way to logically model the spread of an infectious disease throughout the world. At particular points during the construction of the MST, the vertices in the MST represent the expanding infected region, which is a bit different than the traditional examples of applications of Prim’s algorithm.

**Works Cited**

Goodrich, Michael T., and Roberto Tamassia. *Data Structures and Algorithms in Java*. New York: John Wiley, 2014. Print.

Sedgewick, Robert, and Kevin Wayne. "4.3   Minimum Spanning Trees." Princeton, 23 May 2014. Web. 18 Dec. 2014.

Smith, David, and Lang Moore. "The SIR Model for Spread of Disease - The Differential Equation Model." *The SIR Model for Spread of Disease*. Mathematical Association of America, 1 Dec. 2004. Web. 10 Dec. 2014. <http://www.maa.org/publications/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation-model>.