**Euclidean Traveling Salesman Problem**

Connor Johnson

December 2018

**Introduction**

The Euclidean Traveling Salesman Problem is a graph theory problem that is as follows:

Given a list of cities in a two-dimensional space, what is the shortest possible route that can be followed to visit every city and return to the starting position?

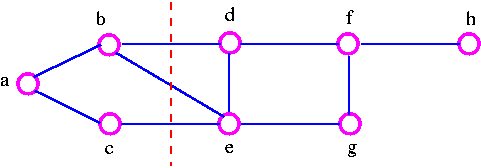
The brute force solution to this problem is not reasonable as there are over n! different possible routes that can be taken. Currently, there isn’t a known solution to this problem that works in polynomial time. So, we are going to use an algorithm that finds a very good solution that my not be the best. This algorithm works in a much more reasonable amount of time, so it is arguably the best solution currently known.

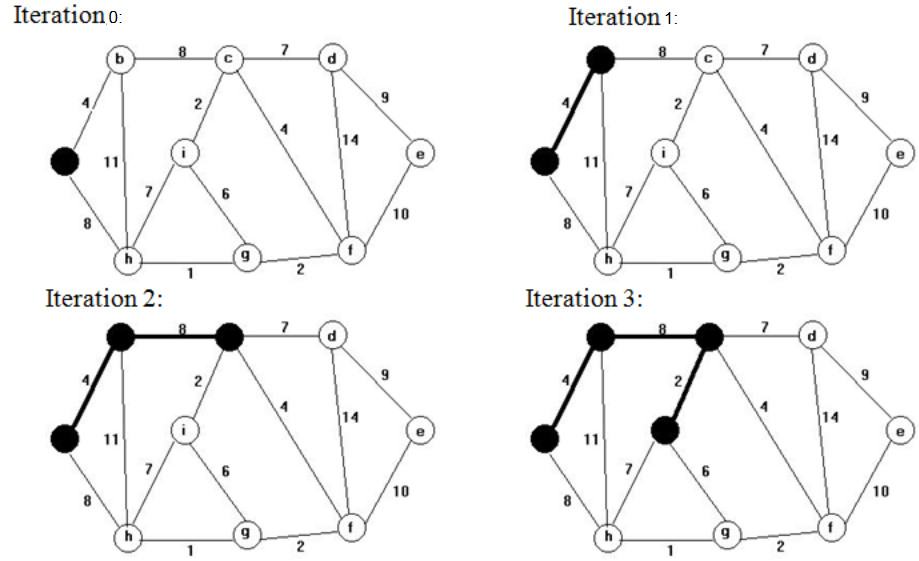
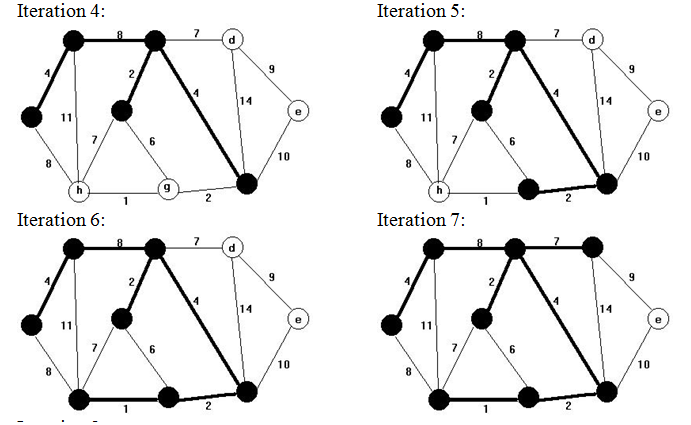
**Minimum Spanning Tree**

First, we must look at the type of path that is going to be found. A loop that goes through every city in the graph can be thought of as a tree with one extra edge added between the first and last city. If we remove one edge from a loop, we get a spanning tree. This is important because we can use a minimum spanning tree(MST) to get an approximation of our solution. A minimum spanning tree is the smallest tree in a graph that goes to every vertex. Since an edge needs to be added to a tree to create a cycle, we know that the length if the solution must be longer than the length of the MST. This gives us a starting point for the solution.

**Prim’s Algorithm**

Prim’s algorithm is the most efficient way to find the MST of a graph. This algorithm is based on the use if cuts in a graph. A cut is removing a group of edges to split the graph into multiple components. This can be cutting the graph into two independent graphs or removing a vertex from a graph. A picture of a cut is shown below.\



This is important because the shortest edge in a cut **MUST** be a part of the MST. This is used in Prim’s algorithm. The algorithm starts by picking a vertex. It then finds the shortest edge connected to it and adds that vertex to the tree. It does this again with all the edges that are part of the cut which separates the current tree of two vertices. Then, the shortest edge in the cut is added and this is done until all vertices in the graph are connected to the tree. A diagram of this is shown below.

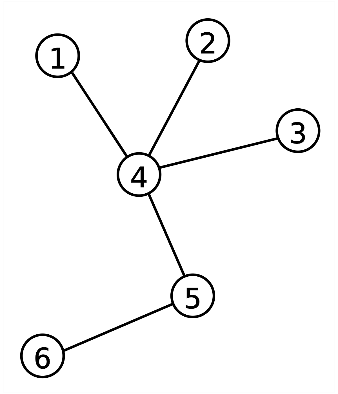
This is known as the lazy implementation of prims algorithm. It has runtime of O(ElogE). For our problem, every vertex is connected to every other vertex, so the number of edges is V(V-1). Because of this, the lazy implementation is not very efficient. So, the eager implementation is used instead. The eager implementation keeps better track of connections between tree vertices and non-tree vertices. The minimum edge between the tree and each non-tree vertex is saved so less comparisons need to be made. For example, in the diagram above, vertex h is connected to the tree in one way with the edge of length 8 at Iteration 0. After Iteration 1, vertex h now has two connections to the tree. The new edge has length 11 which is longer than the original. So, the new edge is removed from the possible edges because it can’t be part of the MST. This happens again at Iteration 3. Vertex h has a new connection to the MST of length 7 and the old edge of length 8 would be removed because it is longer. This method of searching through the tree is much more efficient because fewer comparisons need to be made when edges are removed. The new runtime is O(VlogE) which is much efficient considering we have to deal with a lot of edges.

**Creating the Loop from the MST**

After using Prim’s algorithm, we have the group of edges that make up the MST. One way to find a loop is using a full walk depth first search. A depth first search is a recursive algorithm that accesses every vertex in the graph. It does this by going to a vertex and marking it to show that it has already been visited. It then visits each unmarked vertex connected do it and repeats that process. Since this is a tree, there are no cycles so it is impossible to reach the same vertex twice so they don’t need to be marked. When each vertex is visited, it is added to a list before and after the call to the next vertex. The pseudocode and a diagram of how it traverses the graph is shown below.

Pseudocode:

*List<Vertex> path;*



*Void dfs(Vertex v){*

*path.add(v);*

*for each connected vertex p{*

*dfs(p);*

*path.add(v);*

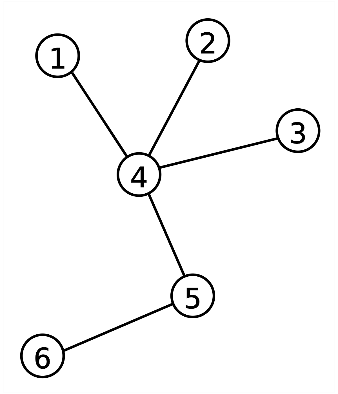
*}*

*}*

This creates a loop that goes to every vertex at least once. The length of this loop is:

*|Length of full walk| = 2 x |Length of MST|*

This is because it crosses every edge in the MST exactly twice.To make a more efficient loop we can remove the second path.add(v) to create a preorder walk so each vertex is added to the loop only once. Then, an edge is added from the final vertex to the starting to complete the trip. This loop is shown below.



The big difference is when a vertex in the MST has multiple edges. Instead of returning back to that original vertex after each call, it moves straight to the next one. This is always less than or equal to the distance of going back to the original vertex. This can be seen in vertices 2, 3, and 4 above. The three vertices create a triangle. The full walk uses two sides of the triangle while the preorder walk only uses one. It is a common rule in trigonometry that adding two sides of a triangle is always greater than the third side. Another way to look at is through the depth first search, at some point path must go from 2 to 3. The fastest way to get from one point to another is always a straight. This is shown in the formula:

*distance(a,b) <= distance(a,c) + distance(c,b)*

Traveling to another point first will never make the total distance shorter. Using this, we get the length of our final path to be:

*|Length of MST| < |Length of Path| <= 2 x |Length of MST|*

**Conclusion**

Obviously, this isn’t always the fastest path that goes through every city. The above equation shows that it is close. It is not worth it to use an algorithm with a longer runtime to get a solution that is only marginally better. So, Prim’s algorithm and the preorder depth first search is the best method because if its faster runtime and approximately correct solution.

https://en.wikipedia.org/wiki/Tree\_(graph\_theory)

https://www.tutorvista.com/content/math/prims-algorithm/