Minimum Spanning Trees & Shortest Path Trees

Not a paper about arborology

Joshua Pondrom

November 28, 2017

1 Motivation

Say there is a problem in which we need to connect everything using only certains paths, such as networking, electrical grids, or trying to go to multiple places in a minimal amount of time. These are all problems that can be reduced down to minimum spanning trees, or to shortest path trees then formally and mathematically solved using graph algorithms. Like many graph algorithms, minimum spanning trees and shortest path trees, are very applicable to real, tangible problems found in our everyday life.

2 Background

Formally a minimum spanning tree is defined on a weighted graph, G(v, e), such that all vertices, v, are connected through edges, e, and in a way which produces a acyclic tree with a minimum possible total weight of edges in the tree. The edge weights can represent anything which is applicable to the problem, such as distance, resistance, cost, *et cetera*.

Likewise, a shortest path tree is defined on a weighted graph with a source, s. The shortest path tree will be a tree branching from s connecting the source to every vertex in the graph with a minimum total weight.

Problems like this have been being worked on since around the 1920's, such as Boruvka's algorithm for minimum spanning trees developed in 1926. Later there was algorithms developed by Jarnik (1930), Prim (1957), and Dijkstra (1959). More recently there was improvements to the algorithm made by Karger, Klein, and Tarjan (1995).

3 Pseudocode

There will be pseudocode and implementation of both Kruskal's and Dijkstra's algorithm for minimum spanning trees.

First, there will be Kruskal's algorithm. This algorithm was first seen in 1956 and written by Joseph Kruskal, American mathematician and computer

scientist. The algorithm uses greedy techniques but will always give a correct answer. In simple terms what the algorithm does is sort the edge lengths, takes the minimum edge if it satisfies the tree properties, and repeats those steps until there is a minimum spanning tree. The complexity for this algorithm is $O(|E|log|E|)$.

Last but definitely least, we have Dijkstra's algorithm. Dijkstra's algorithm is not exactly the same as Kruskal's. Dijkstra's gives the shortest path between a source vertex and all the rest of the vertices in a graph. This was a algorithm published in 1959 by famous Dutch computer scientist, Edsger W. Dijkstra. In simple terms the way this algorithm works is as follows; Start with the source. Let distance to all other vertices to be infinite, look at all neighbors of source and if the distance through this path is less that what is stored, save it was the new path and distance to that vertex. Repeat and keep placing the newly visited nodes in the queue. Continue until the queue is empty. The complexity for this algorithm is $O(|E| + |V|log|V|)$.

The pseudocode is as follows; for Kruskal and Dijkstra respectively.

```
Kruskal(Graph G):
        A = Empty Set
        P = Partition of vertices of G
        E = Sorted edges of G, ascending
        While E is not empty
                #Invariant : A should satisfy tree
                #            properties
                (u, v) = E.pop
                #Invariant : Edge (u, v) should be
                #            minimum edge remaining
                if P.find(u) does not equal P.find(v)
                        A = A Union (u,v)
                        #Invariant : A contains
                        #            minimum trees
        #Invariant : A is a minimum spanning tree on G
        return A
```

```
Dijkstra(Graph G, Source S):
        A = Empty Set
        For each vertex in G
                distance = Infinity
                previous = Undefined
                A = A Union vertex
        distance to S = 0
        While A is not empty
                #Invariant : all vertices in A are connected to our
                #            current
                v = vertex in A with minimum distance
                remove v from A
                for each vertex connected to v, u
                        #Invariant : edge(v, u) union A should
                        #            maintain tree property
                        new = distance to u + length(u,v)
                        if new < distance to v
                                distance to v = new
                                previous to v = u
                        #Invariant : edge(v, u) should be the
                        #            shortest edge to connect
                        #            v and u seen so far
        return A
```

4 Testing Plan

For the program, Javascript will be good for a implementation language. The reason for this is the abundance of libraries for easily showing graphs and the simplicity of the language, allowing for more focus on the algorithm. Additionally it should be easy to make to make a interface to present the code with, with functionality like interacting with the graph and showing changes. The program can be timed by using the GNU Unix tool *time*. This is a simple lightweight progrgam that can be used to see how much time the program uses. It can be combined with a small shell script if there is more data needed, such as a dataset to get an average.

4.1 Invariants

Luckily, Javascript has a assert function to help code invariants into the code and guarantee code correctness. In Javascript we can also give a description to assert statements to better see where any possible errors in the code are. There can also be function definitions to put in assert statements to help do more verbose and powerful invariants.

5 Testing

The data gathered by timing the code on data sets is as follows:

trial	1	2	3
Kruskal	.11s	.09s	.10s
Dijkstra	.12s	13s	10s

Overall, they were close in runtime with Dijkstra's algorithm taking slightly more time on average. This makes sense because the runtime of Dijkstra's algorithm is $O(|V|^2)$ while Kruskal is $O(|E|log|V|)$

6 Conclusion