**Project Title:** Minimum Spanning Tree Generator

**Project Members:** Kyle Anderson and Cole Pierce

**Accomplished:** We accomplished all of the aspects from our initial project proposal. The random graph generator (gengraph.hs) takes in a number of vertices and outputs a random graph to the file graph.txt. That file is then parsed with Grammar.Y and Tokens.X into our UndirectedGraph type (Graph.hs) of [V][E] where E is (cost,V,V). The UndirectedGraph type is passed to the Prim and Kruskal function in Prim.hs and Kruskal.hs. Each step of the algorithm is outputted in txt which is just show of the UndirectedGraph and gv which is the GraphViz format of the UndirectedGraph. The gv files are then converted to images using the script from hw8.

**Not Completed:** Everything we put in our initial proposal is completed. Next steps might be to allow the user to specify an initial graph file instead of generating a random one, show the edge(s) being considered in current step in the output, and implementing other graph related algorithms.

**Code Highlight:** One interesting component of our project is the random graph generator. The graph generator takes in one argument from the user, a number (greater than one) that determines the number of nodes in the graph to be generated. We used the State monad (similarly to what we did in homework 2) and a pseudo-random number generator to construct a graph in which the edges between two nodes have random values. The graph is constructed in the following way: after the user enters some number *n*, the first *n* letters of the alphabet are grabbed and passed to a function which takes two lists of characters and produces a list of edges (which are tuples of Char, Char, Int). The first list begins with only the character A; the second list contains the rest of the characters to be added. The function grabs the first letter in the second list, selects a random letter from the first list, and generates a random edge cost between the two selected letters before proceeding recursively, each time returning the new “edge” and adding the new letter to the first list. In this way, we can easily build a graph that is completely connected and has redundancies that will be eliminated by the MST algorithms. When we initially ran the graph generation, we noticed that for small inputs, the number of redundancies was generally pretty limited; modifications were made so that there would be additional extraneous edges so that the results of the MST algorithms were clearer.

**Sample Inputs and Outputs:** In the repository we left the output after running the program with 10 vertices in the Sample\_Output folder.

**Instructions for Running Code:** To run the code open gengraph.hs in Emacs, compile and type :main n where n is the number of vertices. Then in the directory that the files are generate to, run the rundot script to convert the .gv files to JPEG. Every time the program runs, it overwrites the previous files already there. So if you run with n=3, then n=5, the n=3 files will be overwritten with the n=5 files. However, if run with n=5 then n=3, the n=3 output will only overwrite part of the n=5 output files (the later steps will be the leftover from n=5). Deleting output files from a previous computation prevents any confusion.