Andy Franck - CS 163
Program 5 - Efficiency Writeup

Like the programs before, the data structure was assigned, so there was no freedom in choice. There was little data to store, and we were allowed to use strings, so I created two nodes, one for each vertex (which stores the name of the location), and one for each edge (which stores the name of the street connecting the two vertices/places).

Because of how different this data structure is compared to the rest I have worked with; my node structures were completely different than they have been in previous programs. Like I quickly mentioned above, there are two node structs, one for the edges, and one for the vertices. The vertex node stores a string value for the name, and a head pointer to the list of adjacent places. The other node is the edge node, which stores a string value for the street name, as well as two pointers, one to the vertex node, and one to the next node.

In regards to data structure, I believe this is clearly the best one for the job. I cannot think of a way we could use anything other than an adjacent list. Since we need to be able to add as many connections as we want, I think that the other data structures would not work, as they would limit us in that regard. I'd like to conduct further research into data structures that can be used for graphs (I have read that stacks/queues can be incorporated), but with my current knowledge it seems like there is no better method than the one we are currently using.

When discussing memory overhead and usage, again I think that there is not a better, more efficient option than this one. Because of my small amount of knowledge of graphs, I do not know many other more efficient methods than our current one, however I can note that there are very few pointers at use here (at least compared to something like a Binary Search Tree). We do not appear to be wasting any pointers, unless we create a vertex without any connections. Other than that, it appears we are making full use of everything without much memory waste.

I think the most efficient part of my program is the edge insert function. Inspired by the lab, I created a "find_location" function that finds the location in the array of the current string that is passed in. I call that function twice to find the two connection points, then check if either are null. If they both exist, then I just create a new edge node and connect it both ways. If I want to add capability for a one-way street, I could just create a boolean value to pass into the edge function, and if true/false, the edge node can be created in such a way that the street can go one way or two ways. This could also be customized so the user could choose which way the one-way street goes.

Similarly, I was able to shorten and simplify my code by using for loops in almost all of my functions. Because we already know the size, I do not have to worry about using while loops and temp nodes to traverse. I can just create a for loop, and use my parameter to access every entry in the adjacency list. Although this shortened my code and made it considerably easier to read, by far the most impactful method I used was changing from char * to string for my names. This made it considerably easier to assign names to certain places, because I did not need to use strcpy or strcmp to compare the names of my vertices. I also did not need to actively deallocate the memory for each name. This was by far the most effective method to simplify my code that I took advantage of.

The main difficulty I faced was my edge function. I originally did not think to make a function to find the location of both of the vertices, so my edge function was extremely long and confusing. When I was able to take inspiration from the lab, I realized how much I could condense my code if I had two locations to create the edge node around. This meant that my actual node creation was only 5 lines, when before I had over double when trying to find the locations.

If I had more time to work on this program, I would spend it creating more functions to expand the capabilities of my ADT. I would like to add a function to remove both edges and vertices, because currently there is no way to remove either the vertices or edges. This makes it extremely inconvenient to test large amounts of data, because I have to re-run the program every time I need to get rid of inserted data. I think this would also make more sense if this were a real ADT being used, because the client needs to be able to remove pathways if they become inconvenient, or remove places if they close down/no longer exist/are not used.