Specification

Assignment 4: The Circumnavigations of Denver Long

Files needed:

vertices.h - for this lab we need to define the vertices macros in this header file to use in our graphs. Given to us in the resources repository

graph.h - sets out all the definitions for the graphs ADT. given to us in the resources repository.

graph.c - our c programming implementation of the graphs to be used in the path finder. The graph will display the vertices and we will be trying to find the shortest path through all the vertices. We are in charge of making all the functions that are in the assignment document.

stack.h - sets out all the definitions for the stack ADT

stack.c - our c programming implementation of the stack to be used in the path finder. Very similar to the one used in assignment 3 with a few other features in it. We are in charge of making all the functions that are in the assignment document.

path.h - sets out all the definitions for the spath ADT

path.c - our c programming implementation of the paths to be used to find the shortest path. We are in charge of making all the functions that are in the assignment document. I believe this is where the dfs algorithm will be

tsp.c - our main() for the program. This as always is where all the system arguments are and where the printing will occur.

Makefile: a make file to easily run the program by prompting `make` into the Linux terminal on our Ubuntu virtual machine.

README.md: describes how the program works and runs in addition to how to use the makefile

Description of the Program: In this program we will be using the DFS (Depth-first search) algorithm on a graph to find the quickest **hamiltonian path**. A hamiltonian path is when you start at a vertex (starting destination) and reach every other vertex exactly once until you reach the end vertex (ending destination). Because we can only hit every vertex once it is a little more difficult than your standard path finder problem. But like sorting a path finder is a common problem that all computer scientists must go through. I am familiar with breadth first search (BFS) algorithm as well, but this path finder algorithm will not be used at all in this assignment.

Tsp.c main():

```
Enumerator options{helper message,
                   Verbose printing
                   Undirected graph
                   Infile
                   Outfile} options;

const char *names[] = { "helper", "verbose", "graph", "infile",
"outfile" };
```

```
int main(int argc, char **argv) {
  while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        Set options = empty set
        switch (opt) {
        case 'h': set_insert (helper)
        Case ‘v’: set_insert (verbose)
        Case ‘u’: set_insert (undirected)
        Case ‘i’: set_insert (infile)
        Case ‘o’: set_insert (outfile)

 for (options i = helper; i <= outfile; i += 1) {
        if (set_member(options, i)) {
            //Do what is supposed to happen if these functions
            //command line arguments are triggered.
```

By now in the main file we have become pros at how to access certain command line functions, and like last assignment we will try to maintain good practices by utilizing the sets data structure that was given to us. In this assignment there are 5 command line arguments that all do different things as you might imagine. The first command line argument is -h which is the helper message. This will print a helper message that explains the graph ADT and show what other command line arguments it takes. Next we have -v which stands for verbose printing. When triggered it will print out all the hamiltonian paths and how many recursive calls are to the DFS algorithm. Then we have -u, which stands for undirected graph and does exactly what it says. Lastly we have -i and -o which can give some customization to a specific input and output. In the input you must list all the places and the <i,j,k> coordinate which gives the position in the graph and the weight of it (in this case distance). Also you can customize the output but if not specified they will use stdint or input and stdout for output.

<u>Stack.c</u>

Struct definition: SOURCE DARRELL LONG

```
Struct stack {
  Uint32_t top;
  Uint32_t capacity;
  Int64_t *items;
};
Stack create{
*create stack
}
Stack delete{
Deletes stack
}
Stack full{
*stack full = top of stack = capacity
}
Stack empty{
*Stack empty = top is at 0
}
Stack size{
Returns stack size
}
Stack push{
*push element to stack. LIFO
}
Stack peek{
*returns true if there is element in the stack
}
Stack pop{
*take last element in first out
}
Stack copy{
*the stack gets copied from dst to src
}
Stack print{
Print elements in a stack
}
```

The stack is primarily the same as the previous assignment but it will include a peek function which is a boolean function that returns the previous vertice. This is important when running the DFS. Also there's a copy function that copies the stack content from the dst to the src. This is useful in this assignment because you push the vertices to the stack so you can find the sequence of the hamiltonian path of the graph.

path.c

Struct definition: SOURCE DARRELL LONG

```
Struct path {
  stack* vertices
  Uint32_t length
};
path create{
*create path
}
path delete{
*Deletes path
}
path push vertex{
*pushes vertices stack
}
Path pop vertex{
*pops the vertices stack
}
```

```
path_vertices{
*Returns stack size
}
Path_length{
*Returns weights added up
}
path_copy{
*make a copy of the path dst to src
}
path_print{
*prints path, requires stack to print
}
```

The path is sort of like an inherited data structure that uses the stack to function. Per usual we have a path create and delete where we will need to allocate memory using malloc, and delete the path using the free method. Then we have the push and pop features of the stack that push the vertices to the stack. Next we have a vertices function that returns the stack size which is equivalent to how many vertices has been pushed onto the stack. Path length on the other hand returns the weights added up to tell you the distance covered in the hamiltonian path. And lastly just like in the stack we have a copy function to copy the path traveled to be printed in the stdout.

graph.c

Struct definition: SOURCE DARRELL LONG

```
Struct graph {
Uint32_t vertices;
Bool undirected
Bool visited[vertices];
Uint32_t matrix[vertices][vertices]
};
graph create{
*create graph
}
graph delete{
*Deletes graph
}
graph_vertices{
*returns number of vertices in graph
}
graph_add_edge{
*adds an edge from i, and j to another i and j and also calculates
 Mass of coordinate which resembles distance
}
Graph_has_edge{
*returns true if i and j are within bounds
}
Graph_edge_weight{
*returns the weight of the coordinate i and j. Weight is k
}
Graph visited{
*returns true if vertex has been visited
}
graph_mark_visited{
*if vertex is within bounds, mark v as visited
}
Graph_mark_unvisted{
*if vertex  v within bounds mark v as unvisited
}
Graph_print
*for debugging purposes
}
```

Lastly we have the graph data structure that has four variables. One for the vertices, one for the undirected graph, and lastly an array for a visited variable, and a two dimensional array to represent the matrix of the graph. Like path we have to have a constructor and destructor function where we allocate memory and free that memory once the program is done. Next we have another vertices function which returns how many vertices are on the graph. Also it is important to note that this graph can be up to a 26x26, but no more than that, and this macro is defined in the header file. Next we have an edge function which can add a k component or the weight of the graph which resembles distance. And the has edge function returns true if there's an edge within the 1 <= i <= 26 by 1 <= i <= 26 graph. I mentioned that the k variable and how the k variable determines weight but there are also two other variables that represent the position in i and j, and they are resembled altogether in a <i,j,k> format but in terms of the computer program it will solely be represented by Matrix[i][j]=k. The next function on the list is the graph_edge_weight which returns the weight of a vertex that is assigned with a nonzero positive number. Lastly we have a mark visited, unvisited function which is there solely for the DFS algorithm because it must travel to all the vertices or nodes and find the shortest hamiltonian path. There is also a print function which will be primarily used for debugging purposes.

DFS

```
Procedure DFS(G,v):
  Label v as visited
  For all edges from v to w in G.adjacentEdge(v) do
    If vertex w is not labeled as visited then
      Recursively call DFS(G,w)
  Label v as unvisited
```

The DFS algorithm as previously explained is how we are going to find the shortest path. It is recursively called so it can mark vertices as labeled once you have gotten to a vertice. But it also can unmark vertices if it runs into a dead end. However, a hamiltonian path requires us to visit each vertex once. You will be pushing and popping values before the recursion happens, and this algorithm is in charge of this whole assignment essentially.