Natalie Valett, nvalett
CSE13S
Prof. Darrell Long
April 19, 2021

Asgn4 Design:

Summary:
This lab's objective is to find the shortest path that can be taken to visit each city given in the input exactly once. The program will first read in the input of city names and take in their coordinates and distances from each other. Then we'll perform a recursive depth-first search which will calculate all possible paths between all the vertices, and save only the shortest path. The program will then print out the fastest path found.

Abstract Data Types:

- Graph
    - Composed of vertices and edges
    - Holds the matrix representing all connected vertices and their weights
- Path
    - Contains a stack holding vertices that've been transversed to get to the current path
    - Keeps track of the path length in order to compare it to shortest path
- Stack
    - Same as from asgn3 but with stack_cpy

Command-line Options:

- -h: Prints out a help message
- -v: Enables verbose printing. If enabled, the program prints out all Hamiltonian paths found as well as the total number of recursive calls to dfs().
- -u: Specifies the graph to be undirected
- -i infile: Specify the input file path containing the cities and edges of a graph. If not specified,the default input should be set as stdin.
- -o outfile: Specify the output file path to print to. If not specified, the default output should be set as stdout.

Input:
```
4          // number of vertices
Asgard     // city 1
Elysium    // city 2
Olympus    // city 3
Shangri-La //city n (4)
```

```
0 3 5       // edge 1 <i, j, k> ->
            // there’s an edge going from vertex 0 to 3 with weight of 5
3 2 4       // edge 2 <i, j, k>
            // there’s an edge going from vertex 3 to 2 with weight of 4
2 1 10      // edge 3 <i, j, k>
            // there’s an edge going from vertex 2 to 1 with weight of 10
1 0 2       // edge n (4) <i, j, k>
            // there’s an edge going from vertex 1 to 0 with weight of 2
```

Using fopen() and fclose():
Eugene lab 4/27 timestamp ~100:00

<u>Pass from Console:</u>

```
./io-test < test.txt
```

This passes test.txt to the program to read

```
./io-test < test.txt > test2.txt
```

This reads from test.txt and prints to test2.txt

Code:

```
//prints everything typed in the console to standard output
while (fgets(buffer, KB, stdout)) {
   fputs(buffer, stdout);
}

//reading from a file: line looks like “1 2 3”
while ((c = fscanf(stdin, “%d %d %d\n”, &i, &j, &k)) != EOF) { //or EOF
   //if line c isn’t composed of 3 ints
   if(c != 3) {
      printf(“malformed line”);
   }
   fprintf(“i = %d\n”, i);
   fprintf(“j = %d\n”, j);
   fprintf(“k = %d\n”, k);
}
```

<u>Open file in code:</u>

```
FILE *infile = fopen(“test.txt”, “r”); //r for read
if (infile == NULL) {
        printf(“stderr, “failed to open test.txt”);
        Return 1;
}
```

```
FILE *outfile = fopen("out.txt", "w");
if (outfile == NULL) {
        printf("stderr, "failed to open out.txt");
        Return 1;
}
while ((c = fscanf(stdin, "%d %d %d\n", &i, &j, &k)) != EOF) { //or EOF
   //if line c isn't composed of 3 ints
   if(c != 3) {
      printf("malformed line");
   }
   fprintf(outfile, "i = %d\n", i);
   fprintf(outfile, "j = %d\n", j);
   fprintf(outfile, "k = %d\n", k);
}
```

<u>tsp.c</u>

<u>Psuedocode:</u>

```
city_names = []
//Take in first line of input
n = getopt()
if n > VERTICES: //max num of vertices
        Print error
For i in n: //for number of cities
        //read in city names, append to list
        // use strdup()
        fgets(buffer, BLOCK, stdin); //takes in memory of size BLOCK into buffer
        City_names[i] = strdup(buffer); //saves copy of string into citynames
        If string is malformed:
                Print error
        Replace "\n" at end of string with "\0"

//have to free name space b/c of strdup() function
For i in n:
        free(city_names[i]);
G = new Graph()
While ((input = fscanf()) != EOF):
        If line is malformed:
                Print error
        graph_add_edge(g, input[0], input[1], input[2])
current_path = new Path()
shortest_path = new Path()
For vertex in range(START_VERTEX, g->vertices): // for all vertices
        //depth-first search on g
        dfs(g, vertex, current_path, shortest_path, city_names, outfile)
        Print shortest_path.length
        // print out path
        Print “Path: “
        For vertex in shortest_path.vertices:
                Print vertex + “ -> “
        Print total num of recursive calls
```

<u>Depth-First Search:</u>

```
dfs(G, v):
        graph_mark_visited(G, v)
        //if path visits all vertices and is shorter than shortest path
        If current_path is hamiltonian and current_path.length < shortest_path.length:
                path_copy(shortest, current) //copy current path into shortest
                //print all Hamiltonian paths as they’re found if -v (verbose printing)
                If (extern var) verbose == true:
                        For vertex in current_path.vertices:
                                Print current_path.length
                                Print (vertex + “ -> “)

        For i in range(G->vertices):
                if(graph_has_edge(v, w) and w is unvisited):
                        path_push_vertex(p, w, G)
                        dfs(G, w)
                        path_pop_vertex(p, w, G)
        graph_mark_unvisited(G, v)
```

```
path_is_hamiltonian():
        Return path_vertices == vertices //true if path contains all vertices
```

Where to start:

graph -> stack -> path -> dfs