Assignment 4 – Surfin' U.S.A

Eric Lee

CSE 13S – Spring 2023

Purpose

This program will find the optimal path between cities mentioned in the song "Surfin' U.S.A." by the Beach Boys excluding Waimea Bay and Narrabeen. The optimal path will have the least total distance and therefore use the least gas. It will also start and end in Santa Cruz. The new path is output in a new file.

How to Use the Program

To run the program, the user needs to compile and run the tsp.c file. To set parameters for the program, the user can use different command-line options to change the cities, paths, the weight for each path, and whether or not the paths are directed.

The user can use the option "-i: " to set the input file with all the details about the path like the city names, number of paths, and their weight. The option requires the name of the file as an argument. The default filename is "stdin". The option "-o: " requires a filename to set as the output file. The default output filename is "stdout".

The option "-d" sets all the paths as directed, while the default is for all the paths to be undirected.

The option "-h" outputs a help message.

Program Design

The main function and helper functions are all found in the file tsp.c. The main function interfaces with auxiliary files for three ADTs: graph.c, stack.c, and path.c. Further details on these files can be found below.

Data Structures

The file graph.c uses a struct type called graph. The type graph contains two 4 bit integers “vertices” and “**weights” , two booleans “directed” and “*visited” , and a char “**names”. The asterisks mean that the variable is a pointer to another value, and two asterisks mean that it is a pointer to another pointer.

The file stack.c also uses a struct called stack. The struct takes contains three 4 bit integer variables: capacity, top, and *items. The maximum size of the stack is stored in “capacity”, and the end of the stack is stored in “top”. The pointer “*items” points to an array of city names.

The file path.c uses a struct called path. The struct only contains one 4 bit integer, “total_weight”, and a pointer to the array of vertices in the stack. The total length of the path is found in “total_weight”.

Algorithms

All of the edges and weights that are stored in graph.c are in an adjacency matrix. To read the arguments from the textfiles, they are organized into four sections. The first line is an integer for the number of vertices. The lines after that are all the names of the vertices/cities. Then there is one line with an integer for the number of edges. The rest of the lines are all edges, formatted as an adjacency list. Each edge consists of three integers, the number for the starting vertex, the ending vertex, and the weight of the edge.

Stack is organized like an array, but the program can only look at the last element in the list. Items can also only be inserted or removed at the end.

Path uses both stack and graph ADTs. The vertices, or cities, are stored into a stack. To find the distance between cities, Path uses the edges set in the graph.

To traverse through every edge in the path, TSP uses a depth-first search algorithm. The algorithm uses recursion to look through every edge that has not been traversed by looking at all edges connected to the vertex it is currently looking at.

Function Descriptions

The file graph.c contains 12 functions.

- The function “*graph_create” takes in an integer for the number of vertices and a boolean to tell it if the graph is directed or not. The function creates a new graph struct and returns a pointer to it.
- The function “graph_free” takes in a double pointer and frees all the memory in the graph of the pointer.

- The function “graph_verticies” takes in a pointer to the graph and returns the number of vertices in there.
- The function “graph_add_vertex” takes in the value of a vertex and a name, and sets that vertex as that city’s name.
- The function “graph_get_vertex_name” returns the name of a city given a vertex value.
- The function “**graph_get_names” returns all the city names in the graph as a double pointer.
- The function “graph_add_edge” takes in 3 int values to add an edge between two of the values as vertices and uses the last value as the edge’s weight.
- The function “graph_get_weight” takes in 2 vertex values and returns the weight of the edge between the two vertices.
- The function “graph_visit_vertex” takes in a vertex value and adds it to a list of visited vertices.
- The function “graph_unvisit_vertex” takes in a vertex value and removes it from the list of visited vertices.
- The function “graph_visited” takes a vertex value and returns TRUE if it is in the list of visited vertices and FALSE otherwise.
- The function “graph_print” takes in the pointer to the graph and prints a readable version of it.

The file stack.c contains 10 functions.

- The function “*stack_create” takes in a value for the storage of the stack, creates the stack, dynamically allocates memory for it, and returns a pointer to it.
- The function “stack_free” takes in a pointer for the stack and frees all of its memory, setting the pointer to NULL.
- The function “stack_push” takes in a value and adds it to the top of the stack and increments the counter for the top. It returns TRUE if it is successful and FALSE otherwise.
- The function “stack_pop” takes in a pointer and sets the last integer in the stack to the pointer. It also removes that item from the stack and returns TRUE if successful and FALSE otherwise.
- The function “stack_peek” does the exact same as “stack_pop” except it does not modify the stack.
- The function “stack_empty” returns true if the given stack is empty and false if it is not.
- The function “stack_full” returns true if the given stack is full and false if it is not.
- The function “stack_size” returns the number of elements in the stack.
- The function “stack_copy” takes in two stack pointers. The first stack is overwritten with the elements in the second stack.
- The function “stack_print” takes a pointer to the stack, a pointer to the output file, and a pointer to a list of the city names. Then it prints a list of the cities to the file.

The file path.c contains 8 functions and uses functions from stack.c and graph.c.

- The function “*path_create” creates a new path struct and returns a pointer to it.
- The function “path_free” takes a double pointer to the path and clears all the memory used by it.
- The function “path_verticies” returns the total number of vertices in the path.
- The function “path_distance” returns the distance of a path.
- The function “path_add” takes a pointer to the path and the graph, as well as a uint32_t value. The value from the graph is added to the path and the distance and length of the path is updated.
- The function “path_remove” removes the newest vertex and updates the path accordingly.
- The function “path_copy” takes in two path pointers and rewrites the first one with the data from the second one.
- The function “path_print” takes the pointer for the path, graph and outfile, printing the vertices into the outfile.

Results

There is a segmentation fault in graph.c even though valgrind returns no errors, so I’m not exactly sure why it is not working in the pipeline. The pipeline also says that stack_copy failed, even though everything works and is set equal. I also did not finish the implementation of tsp.c because I ran out of time.