# Assignment 4 – Surfin' USA

Priya Chaudhari

CSE 13S – Spring 2023

## Purpose

The purpose of this assignment is to utilize graph theory to help Alissa visit each city mentioned in Surfin' USA using the least amount of gas. There will be 3 abstract data types (ADT) used in path, stack, and graph. These data types and graph theory will employ concepts like depth-first search, breadth-first search, stacks, graphs, and paths in their respective source files. The program will utilize infile and outfile features to streamline user inputs and outputs and command-line options will be utilized for the user to maneuver in the terminal. There will also be underlying, compounding concepts that my functions will exercise that play a role in the stacks, with operations like pop, peek, or push, or in paths, with concepts like vertices, weights, indexes, and edges.

## How to Use the Program

In order to use the program, the user needs to enter the command "make" within the terminal which will compile the Makefile and then they must enter one of the 4 main command line options to bring data into my program. Below is a list of all the command line options (inputs and outputs) and what the user would need to enter for a specific purpose.

1. -i: the input file which my program will read, the user must enter their filename as an argument, and it will default to stdin
2. -o: the output file that my program will write to, the user must enter their filename as an argument, and it will default to stdout
3. -d: it is assumed all graphs will be undirected but this command line option treats it as directed
4. -h: will print a help message to stdio

## Program Design

The program is designed with 4 main source files- 3 of which are ADT files (graph, stack, and path) and one main file which will be the executable for this program (path).

*Main Files:*

1. source files (authored by me): path.c, graph.c, and stack.c
2. executable/main file (authored by me): tsp.c
3. header files (provided and unmodified): graph.h, stack.h, path.h, vertices.h (specifies which vertex the path starts at utilized in the program)

*Main Data Structures/Algorithms:*

1. 3 Abstract Data Types (ADT): Graph, stack, and path

2. Stacks and Paths
3. Graph Theory which includes vertices, indexes, edges, and weight

**Stack ADT Design (stack.c):**

- Includes a complete typedef struct which defines uint32_t variables capacity, top, and items which are referenced from stack.h
- Include function stack_create: which will create a stack, dynamically allocate memory to it with the use of malloc or calloc, and return the pointer to the stack
- Include function stack_free: free all the space used in the stack by setting the pointer to NULL
- Create bools for the functions stack_pop and stack_peek which will return true if these functions are successful and false if the result is anything else
- Create bools for the functions stack_empty and stack_full which will similarly return true if the stack is full/empty and false if otherwise
- Create a uint32_t for stack_size which will return the size of the stack
- Create a void stack_copy function which will copy the contents of one stack to another destination stack but also make sure that stack has enough storage/memory allocated to store these items
- Finally, create a void function named stack_print which will print all the contents within the stack that contains vertices and their names

**Path ADT Design (path.c):**

- Includes a complete typedef struct which defines uint32_t variables total_t and a pointer to the Stack typedef with vertices.
- Include function path_create: which will create a path, dynamically allocate memory to it with the use of malloc or calloc, and return the pointer to the path
- Include function path_free: free all the space used in the path by setting the pointer to NULL
- path_vertices: returns amount of vertices in path
- path_distance: retrieves the total weight/distance in the path
- path_add: adds to path
- path_remove: removes from the path
- path_copy: copies the path
- path_print: prints my path

**Graph ADT Design (graph.c):**

- Includes a complete typedef struct which defines uint32_t variables vertices and weights and 2 bool variables directed and visited, and one char variable names.
- Include function graph_create: which will create a graph, dynamically allocate memory to it with the use of malloc or calloc, and return the pointer to the graph
- Include function graph_free: free all the space used in the graph by setting the pointer to NULL
- graph_vertices: returns amount of vertices in path
- graph add edge: will add an edge between the start and end

- graph get weight: will look up and return the distance/weight between the start and end in the graph
- graph visit vertex : checks to see if a vertex was visited
- graph unvisit vertex : checks to see if a vertex was not visited
- graph visited : checks to see if a graph was visited
- graph get names: returns the names of every city in the graph array
- graph add vertex: copies vertex v
- graph get vertex name: optionally prints out graph's info that's readable

**tsp.c setup:**

- dfs function which conducts not only the dfs algorithm given to us in the assignment but also includes code to check if p has n vertices: if last vertex in path is adjacent to start and if so add to path and if shorter than best or best distance is equal to 0 the best should be equal to the current path. Otherwise, remove from path.
- begin main function
- declare variables at top
- switch case
- create graph and 2 paths with our functions created in the source files and iterate through the file with fscanf to read whatever file is inputted by the user
- utilize dfs to find the shortest path and print this using if statements
- free memory and then close all files
- return 0 and finish main function

### Bugs

I do have some memory leaks and some segmentation faults. I don't have enough time to get rid of them completely, sorry.

## Results

One key thing I learned is how important it memory management and ensuring you have your memory allocation freed and dynamically allocated properly. Little things I was doing actually caused some big problems like exiting before freeing my memory. Or closing my files in my main function before even freeing. Albeit, I still have plenty of issues with this program and there is many bugs to debug but I learned how useful valgrind is and maneuvering with lldb is super helpful in pinning down what functions have the biggest issues.