Assignment 4 – Surfin' U.S.A.

Mateo Garcia

CSE 13S – Spring 2023

Purpose

The purpose of this program is to simulate the Travelling Salesman Problem. "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". In our program, Jessie's friend, Alissa, wants to visit all of the places mentioned in the Beach Boys' 1963 song, *Surfin U.S.A.*. The program uses graph theory in order to get Alissa to each city in the shortest distance. It uses an algorithm called Depth First Search in order to find the most efficient/fastest route.

How to Use the Program

First you must go into the cse13s/asgn4 folder where all your files are located. Then run the command **make** and then you can run the main command. By running **./tsp *-[flag1] -[flag2]...*** you can add 4 different flags and they are listed below.

- -i : Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is **stdin**
- -o : Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is **stdout**
- -d : Treats all graphs as *directed*. The default is to assume an *undirected* graph, which means that any edge (i, j) that is specified should be added as both (i, j) and (j, i). So if -d is specified, then (i, j) will be added, but (j, i) won't.
- -h : Prints a help message to **stdio**

Program Design

Data Structures

There are only three big abstract data types used for this assignment. Graphs, Stack, Path. The graphs consists of vertices and edges, you can travel the edges to reach vertices and that is how you can traverse the graph. The graph will be used to map out the locations of cities and the distance between them. The stack is a LIFO data structure and will be used to store information in the program. A path will be used to track the cities that Alissa visits, and it is made of a stack data structure. The program will also use getopt() to pass in the flags, as well as use file functions to read in and write out files.

Algorithms

The main algorithm we will use is Depth First Search.

```
def dfs(node n, graph g):
    mark n as visited
    for every one of n's edges:
        if (edge is not visited):
            dfs(edge, g)
    mark n as unvisited
```

Function Descriptions

These are the functions of the three main ADT's.

This is the stack's constructor:

```
typedef struct stack
    uint32_t capacity;
    uint32_t top;
    uint32_t *items;
```

The stack module will have these functions:

- **Stack *stack_create(uint32_t capacity)** Creates a stack, dynamically allocates space for it, and returns pointer for it. are equal to 0.
- **void stack_free(Stack **sp)** Frees all the space used by the stack given, and sets the pointer to NULL.
- **bool stack_push(Stack *s, uint32_t val)** Adds val to the top of the stack s, and increments the counter (the size value). Returns true if successful, false otherwise. (ex: the stack is full).
- **bool stack_pop(Stack *s, uint32_t *val)** Since val here is a pointer, it sets the integer pointed by val to the last item on the stack s, and removes that item from the stack as well.
- **bool stack_peek(const Stack *s, uint32_t *val)** This function sets the integer pointed by val to the last item on the stack but does not modify the stack. Returns true if successful, false otherwise.
- **bool stack_empty(const Stack *s)** Returns true if the stack is empty, otherwise false.
- **bool stack_full(const Stack *s)** Returns true if the stack is full, otherwise false.
- **uint32_t stack_size(const Stack *s)** Returns the number of elements in stack s.
- **void stack_copy(Stack *dst, const Stack *src)** Stack dst will be overwritten with all the items in Stack src.
- **void stack_print(const Stack* s, FILE *outfile, char *cities**[]**)** This function will print out the list of elements in the stack, starting with the bottom of the stack.

This is the graph's constructor:

```
Graph *graph_create (uint32_t vertices, bool directed) {
    Graph *g = calloc(1, sizeof(Graph));
    g->vertices = vertices;
    g->directed = directed;
    // use calloc to initialize everything with zeroes
    g->visited = calloc(vertices, sizeof(bool));
    g->names = calloc(vertices, sizeof(char *));
    // allocate g->weights with a pointer for each row
    g->weights = calloc(vertices, sizeof(g->weights[0]));
    // allocate each row in the adjacency matrix
    for (uint32_t i = 0; i < vertices; ++i) {
        g->weights[i] = calloc(vertices, sizeof(g->weights[0][0]));
    }
    return g;
}
```

The graph module will have the following functions

- **Graph *graph_create(uint32_t vertices, bool directed)** Creates a new graph struct, and returns a pointer to it. Initializes all items in the visited array to false.
- **void graph_free(Graph **gp)** Frees all the memory use by graph
- **uint32_t graph_vertices(const Graph *g)** Finds the number of vertices in a graph
- **void graph_add_vertex(Graph *g, const char *name, uint32_t v)** Gives the city at vertex v the name passed in.
- **const char* graph_get_vertex_name(const Graph *g, uint32_t v)** Gets the name of the city with vertex v from the array of city names.
- **char **graph_get_names(const Graph *g)** Gets the names of every city in an array.
- **void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)** Adds an edge between start and end with weight **weight** to the adjacency matrix of the graph.
- **uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)** Looks up the weight of the edge between start and end and returns it.
- **void graph_visit_vertex(Graph *g, uint32_t v)** Adds the vertex v to the list of visited vertices
- **void graph_unvisit_vertex(Graph *g, uint32_t v)** Removes the vertex v from the list of visited vertices.
- **bool graph_visited(Graph *g, uint32_t v)** Returns true if vertex v is visited in graph, false otherwise.
- **void graph_print(const Graph *g)** Prints a human-readable representation of a graph

This is the path's constructor

```
typedef struct path {
    uint32_t total_weight;
    Stack *vertices;
} Path;
```

The path module will have these functions:

Results

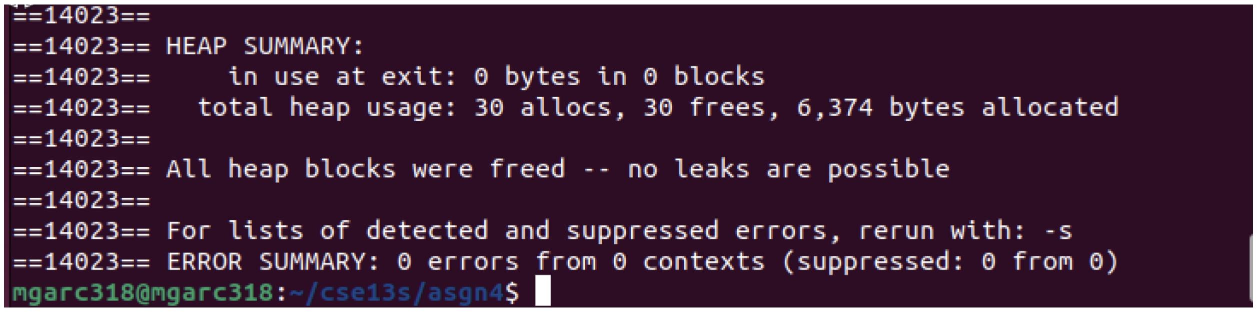
Although I have turned in an incompleted project, I still managed to finish my stack.c, graph.c, path.c, and half of the tsp.c functionality. My main problems were getting the Depth-First-Search algorithm, and the writing in and writing out files. Everything works except for tsp.c. The following section goes into my errors more deeply.

Error Handling

Towards the beginning I had a lot of problems with valgrind, with a bunch of memory leaks and I found out it was a problem with my free functions for each ADT. I didn't free each variable that used dynamic memory properly. After fixing that my main problem was in tsp.c. I followed the DFS pseudocode but it does not work the way I want it to, I believe it is a logic error with the path ADT, I didn't understand it fully and didn't implement it correctly.

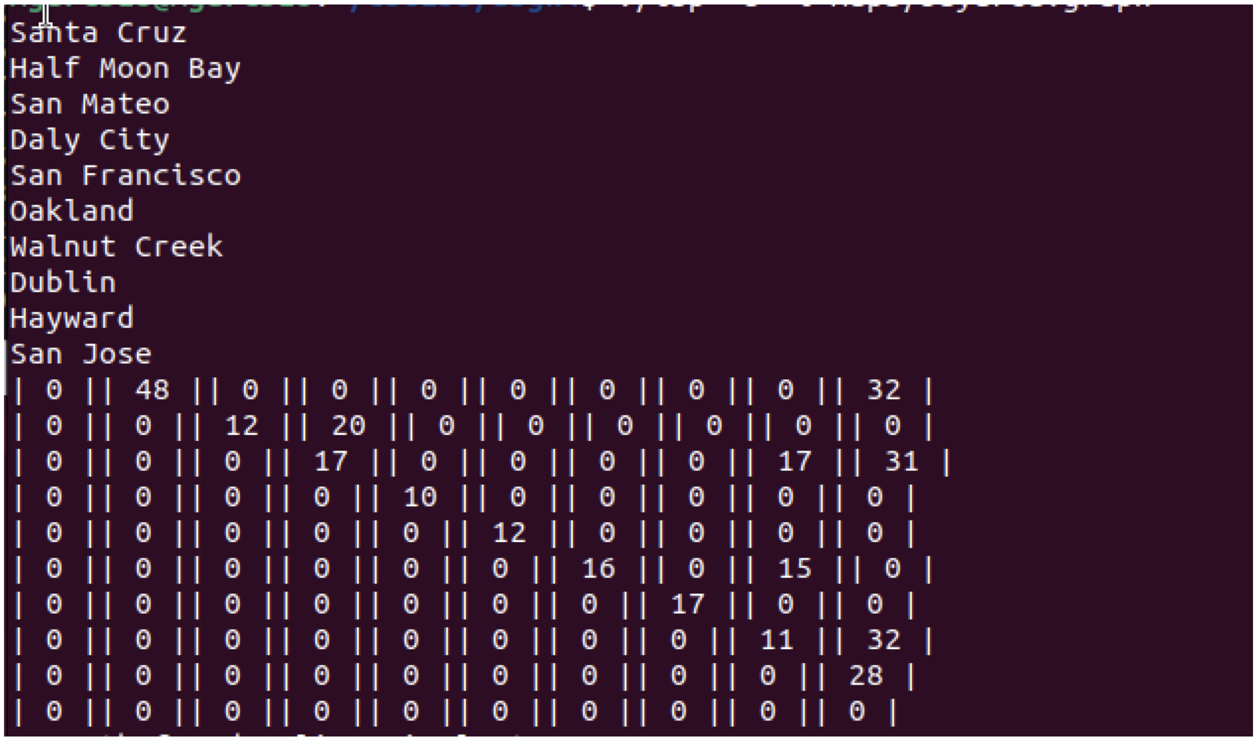
Numeric results

I don't have many results for my tsp.c but I have the graph print and stack print to show.



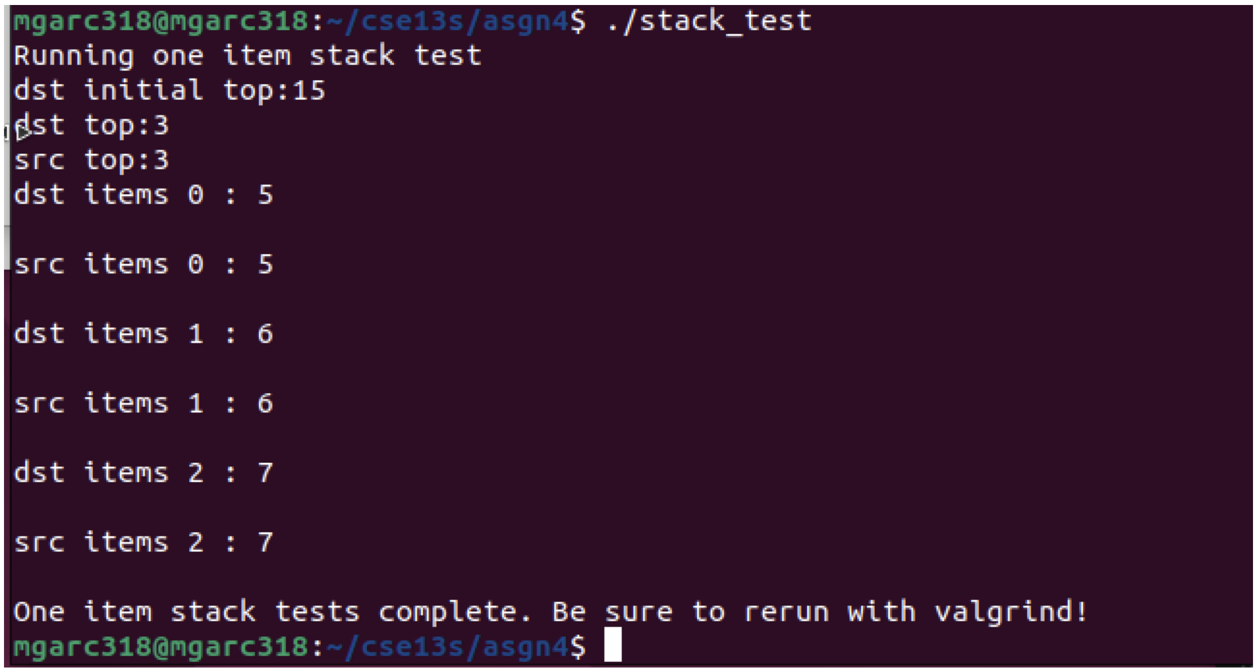
```
==14023==
==14023== HEAP SUMMARY:
==14023==     in use at exit: 0 bytes in 0 blocks
==14023==   total heap usage: 30 allocs, 30 frees, 6,374 bytes allocated
==14023==
==14023== All heap blocks were freed -- no leaks are possible
==14023==
==14023== For lists of detected and suppressed errors, rerun with: -s
==14023== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
mgarc318@mgarc318:~/cse13s/asgn4$
```

Figure 1: Screenshot of valgrind producing no errors (no memory leaks).



```
Santa Cruz
Half Moon Bay
San Mateo
Daly City
San Francisco
Oakland
Walnut Creek
Dublin
Hayward
San Jose
| 0 || 48 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 32 |
| 0 || 0 || 12 || 20 || 0 || 0 || 0 || 0 || 0 || 0 |
| 0 || 0 || 0 || 17 || 0 || 0 || 0 || 0 || 17 || 31 |
| 0 || 0 || 0 || 0 || 10 || 0 || 0 || 0 || 0 || 0 |
| 0 || 0 || 0 || 0 || 0 || 12 || 0 || 0 || 0 || 0 |
| 0 || 0 || 0 || 0 || 0 || 0 || 16 || 0 || 15 || 0 |
| 0 || 0 || 0 || 0 || 0 || 0 || 0 || 17 || 0 || 0 |
| 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 11 || 32 |
| 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 28 |
| 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 || 0 |
```

Figure 2: Screenshot of graph print function.



```
mgarc318@mgarc318:~/cse13s/asgn4$ ./stack_test
Running one item stack test
dst initial top:15
dst top:3
src top:3
dst items 0 : 5

src items 0 : 5

dst items 1 : 6

src items 1 : 6

dst items 2 : 7

src items 2 : 7

One item stack tests complete. Be sure to rerun with valgrind!
mgarc318@mgarc318:~/cse13s/asgn4$ 
```

Figure 3: Screenshot of stack_test.