Assignment 5 – Surfin’

Asaveri Rao

CSE 13S – Fall 2023

Purpose

The program in its core is a fairly simple program. In this program, we are utilizing and implementing code that solves the infamous travelling sales problem(tsp). In simple terms, TSP is basically finding the shortest and most efficient route from one destination to another, without repeating or going backwards in the path. This logic is now translated to Alissa, a character in this assignment, where she will be visiting each city in the song ’Surfin’ by Beach Boys. She has to start from Santa Cruz and ends in Santa Cruz without repeating any cities on her journey, all while running on the least amount of gas.

How to Use the Program

This program has a lot of moving parts, but the overall process to implement and use the code is on the simpler side. Graphs, stacks, and other structures are used in this program. We were provided with certain header files, like graph.h which contains the functions for the logistics behind using a graph, like calculating the vertices, weights, and edges. There is a c file that was created in correspondence to the graph.h file, called graph.c, which includes the header file and implements the functions defined in the header file. Similarly, we were also provided stack.h file and path.h which resulted in stack.c and path.c files to correspond and implement the functions in these files. But the main file that is crucial for this program to run is the tsp.c file which contains the main function that utilizes all these moving parts and executes the program. To most efficiently execute the TSP problem, we are utilizing graph theory. To run and compile the code, I had to create a separate Makefile that consisted of specific targets that pertain to the header files and c files that were created in the process of this assignment. More specifically, to run the and format the program, the programmer has to run make, make format, and make clean to have a properly formatted program. The -g flag was included to run valgrind correctly in the Makefile. To run the program, the use enters ./tsp -d -i maps/bayarea.graph but replace the bayarea.graph part with any of the other .graphs that need to be read. Other than bayarea.graph, there are basic.graph, clique10.graph, clique12.graph, clique9.graph, surfin.graph, clique11.graph, clique13.graph, and lost.graph

Program Design

The program has a few different functions and data structures that are are used. These consist of graphs, stacks, and paths. The main purpose of the graphs are to store the data of Alissa’s journey through an adjacency matrix. This helps organize and store the vertices and weight of the graph in an orderly manner. Stacks are generally used to store elements, and in this program to keep track of Alissa’s journey. Paths are more of a data structure we’ll get into later, but their general purpose is to make sure Alissa is taking the shortest path.

Data Structures

As mentioned before, the two major data structures that are used are graphs and paths. Within some of the c files, arrays, strings, and integers are also used. The main reason why these data structures were chosen was because of efficiently and functionality. For the specific task at hand, of recording the journey Alissa is taking, graphs are the most ideal data structure to record and execute a tsp style problem. To keep track of the shortest path, paths would organize the information most efficiently. Here are brief pseudo codes of the stack header file.

Pseudocode for the stack.c file:

```
Structure Stack:
    capacity, top, items

Function stack_create(capacity):
    s = create Stack with capacity, top = 0, items array of size capacity
    return s

Function stack_free(sp):
    if sp and *sp: free items array in *sp, free *sp; set *sp to null

Function stack_push(s, val):
    return s and not stack_full(s): items[top++] = val; true, otherwise false

Function stack_pop(s, val):
    return s and not stack_empty(s): val = items[--top]; true, otherwise false

Function stack_peek(s, val):
    return s and not stack_empty(s): val = items[top - 1]; true, otherwise false

Function stack_empty(s): return s is null or top is 0

Function stack_full(s): return s is null or top is capacity

Function stack_size(s): return 0 if s is null, else top

Function stack_copy(dst, src):
    if dst and src: assert capacity in dst >= top in src; copy src items to dst;
    set dst top to src top

Function stack_print(s, outfile, cities):
    if s and outfile and cities: print cities[item] for each item in s.items
    up to top in s
```

Algorithms

The main purpose of using the graph data structure is to keep track of where Alissa travels, which cities she goes to, and the distance between each city. Here is a short snippet of the pseudocode for the current graph.c file:

```
Function graph_create(vertices, directed):
    g = create Graph with vertices, directed
    g.visited = create array, initialize with false
    g.names = create array, initialize with null
    g.weights = create matrix, initialize with zeroes
    return g

Function graph_free(gp):
    if *gp: free arrays and matrix, *gp = null

Function graph_vertices(g): return g.vertices

Function graph_add_vertex(g, name, v):
    if v < g.vertices: free existing name, g.names[v] = copy of name

Function graph_get_vertex_name(g, v):
    if v < g.vertices: return g.names[v]

Function graph_add_edge(g, start, end, weight):
    if start < g.vertices and end < g.vertices:
        g.weights[start][end] = weight
        print "Added edge:", start and end, "Weight:", weight
        if not g.directed: g.weights[end][start] = weight; print "Added edge:",
        end, "->", start, "Weight:", weight, "(Undirected)"

Function graph_get_weight(g, start, end):
    if start is less than g.vertices and end < g.vertices: return g.weights[start][end]

Function graph_visit_vertex(g, v):
    if v is less than g.vertices: g.visited[v] = true
```

Path.c was created in correspondence to path.h, where a path data structure is used to keep track of Alissa's journey. Though the actual file is a bit long itself, the main purpose behind the path.c file is to make sure Alissa's journey is as short as possible. Here is a snippet of the path.c file:

```
function path_free(pp):
    if pp is not null and *pp is not null:
        free memory allocated for vertices stack in (*pp)
    if pp is not null:
        set (*pp) to null
function path_vertices(p):
    return size of the vertices stack in p
function path_distance(p):
    return the total_weight field of p
function path_add(p, val, g):
    assert p is not null and g is not null
    if path_vertices(p) > 0:
        prev_vertex = peek the top of the vertices stack in (p)
        weight = get weight of the edge from (prev_vertex) to (val) in (g)
        add weight to total_weight field of (p)

    push (val) to the vertices stack in (p)
```

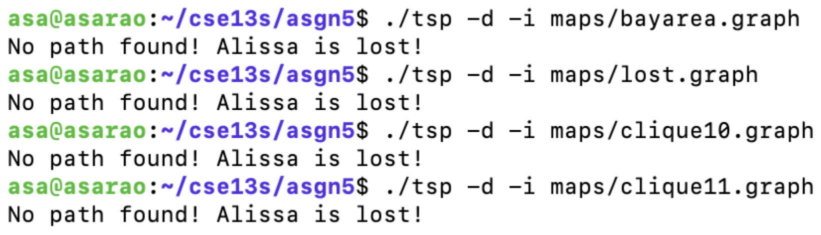
Function Descriptions

The main function for this program is defined in tsp.c, which executes the main program. To briefly explain the code, I put the psuedocode for the main function in the tsp.c file. Pseudo code for the tsp.c file where the main function is:

```
function readGraphFromFile(file):
    initialize an empty graph
    read the number of vertices from the file
    create a graph with the given number of vertices
    for each vertex in the graph:
        read vertex name from the file
        add the vertex to the graph
    read the number of edges from the file
    for each edge in the graph:
        read edge information from the file
        add the edge to the graph
    return the created graph
function dfs(graph, currentVertex, currentPath, bestPath, numBeaches):
    mark currentVertex as visited
    add currentVertex to currentPath
    if all beaches have been visited exactly once:
        if the current path is a valid Hamiltonian cycle:
            if the current path is more fuel-efficient than the best path:
                update the best path with the current path
    else:
        for each unvisited neighbor of currentVertex:
            update distance in the current path
            recursively call dfs for the neighbor
            remove currentVertex from currentPath
    mark currentVertex as unvisited
function solveTSP(graph):
    initialize an empty currentPath
    initialize an empty bestPath
    perform depth-first search starting from a predefined start vertex
    return the bestPath
function main():
    initialize inputFilename and outputFilename to null
    initialize infile to stdin and outfile to stdout
    if inputFilename is not null:
        open infile for reading
    if outputFilename is not null:
        open outfile for writing
    Read and process the graph:
    graph = readGraphFromFile(infile)
    bestPath = solveTSP(graph)
    print bestPath to outfile
    close infile
    close outfile
```

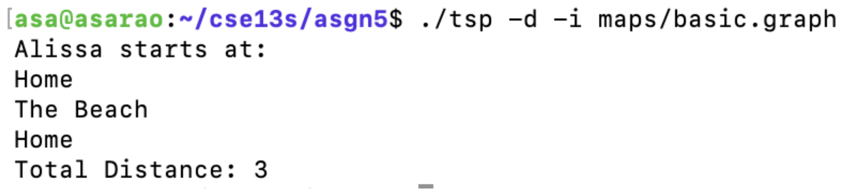
Results

Overall, the program is working. There were a few debugging errors that I needed to fix previously. However, the tsp file executes successfully and as can be seen in the screenshot below of the results, it outputs the correct output. The makefile compiled properly, and gave no compilation errors. I did have a valgrind functionality error before, but I also fixed that. I learned a few different things about the tsp algorithm with this assignment. One of the observations that I had was about clique13.graph because it took a longer time to execute because there was more content in the .graph. The higher the amount of vertices in the .graph files, the longer it took for the tsp file to execute. Accordingly, the less amount of vertices present in the .graph file, the run time was much shorter, like the basic.graph or lost.graph. Something that I also took note of in terms of debugging was how I implemented strategies for this assignment. I added print



```
asa@asarao:~/cse13s/asgn5$ ./tsp -d -i maps/bayarea.graph
No path found! Alissa is lost!
asa@asarao:~/cse13s/asgn5$ ./tsp -d -i maps/lost.graph
No path found! Alissa is lost!
asa@asarao:~/cse13s/asgn5$ ./tsp -d -i maps/clique10.graph
No path found! Alissa is lost!
asa@asarao:~/cse13s/asgn5$ ./tsp -d -i maps/clique11.graph
No path found! Alissa is lost!
```

Figure 1: Screenshot of basic graph



```
[asa@asarao:~/cse13s/asgn5$ ./tsp -d -i maps/basic.graph
Alissa starts at:
Home
The Beach
Home
Total Distance: 3
```

Figure 2: Screenshot of clique graphs

statements for debugging after certain functions and structures to understand what was working and what was not working in my code.