Assignment 4 – Surfin' U.S.A

Noah Meisel

CSE 13S – Spring 2023

Purpose

This program runs through the interface tsp.c and is designed to solve the travelling salesman problem for any given .graph file. This is done by utilizing a graph structure and a Depth First Search(DFS) to create a path that traces a route through the specified graph. The program outputs the path it is taking to a user-specified file(defaults to stdout) along with the total weight of this path.

How to Use the Program

To compile the program you need the Makefile that is found inside the git-repository for the assignment. The make file compiles and links stack.c, path.c, graph.c, and tsp.c into the executable file tsp.

Using this program requires the user to have a .graph text-file in the following format.

```
Number of vertices in the graph
List of Names for all the vertices in the graph
Number of edges in the graph
List of edges in the following format(start node, end node, weight)
```

This text file is read by the program and is used to build and then traverse a graph.

The command-line options for this program are specified as follows.

```
-i filename: sets a file for the program to read from. Defaults to stdin if no file is specified.
-o filename: sets a file for the program to write to. Defaults to stdout if no file is specified
-d: treats the graph as a directed graph.
-h: Prints a help message with these command-line option descriptions to the console
```

Program Design

The bulk of this program is split across three different files-stack.c, path.c, and graph.c, which are linked together and run through tsp.c which acts as the main interface for this program. Stack.c implements a stack data structure which is used in the path.c program to track the traversal of a given graph. All three of these data structures and the functions they used are specified in more detail below. Tsp.c takes command-line options through optarg, and parses them using a switch statement. After the command line-options are handled, the main function starts by reading input from the variable infile, which is set by default to stdin but can be changed by the user as specified above. After the input has been read into a graph, the program runs a depth-first-search on the the graph and then prints the optimal path, while also adding an edge that connects back to the start. If there is no edge back to the start, the program prints that Alissa is lost. The program then frees all allocated memory and exits.

Data Structures

Stack

Stack.C implements the stack ADT. A stack is a LIFO structure. In this program, it is implemented using an array in which only the last element of the array can be altered(as specified in Function Descriptions). The stack data type is ustilized by the the path data structure to track the order with which DFS traverses a given graph. As DFS traverses the graph, our path structure will add each vertex it is visiting to the stack, and remove vertices if DFS backtracks, so that upon completing the search our path is storing the vertices visited and the order in which they are visited.

Path

Path.c implements a path ADT. The path consists of a stack of different vertex names taken from the graph that is being traversed. As the DFS traverses a graph, each vertex visited will be pushed to the stack, and the weight of the edge from the previously pushed vertex to the current one will be added to the total weight measurement.

Graph

Graph.c implements a graph ADT. The graph data type requires the number of vertices the graph will contain to be specified upon creation, as well as whether the graph is directed(whether there is a direction of movement along each edge) or un-directed. The graph data type also tracks whether a vertex has been visited, the names of all the vertices, and an adjacency matrix that specifies all of the edges in the graph and their weights.

Algorithms

Depth First Search, Pseuodocode provided by Jess Srinivas, Ben Grant, and Dr. Kerry Veenstra

```
def dfs(node n, graph g):
    mark n as visited
    add n to the path
    for every one of n's edges:
        if (edge is not visited):
            dfs(edge, g)
            if the current path is shorter than the best path:
                copy the current path into the best path
            remove node from the current path
    mark n as unvisited
    return weight of the best path
```

Function Descriptions

0.1 Stack Functions

Stack *stack_create(uint32_t capacity)

Takes in a 32 bit number and returns a Stack with that many potential elements in it.

void stack_free(stack **sp)

Frees all of the memory used by a specified stack. Returns nothing

bool stack_push(Stack *s, uint32_t val)

Takes in a stack and a 32 bit number. Adds the number to the stack, and returns true if this is completed succesfully. If the stack is full, it won't add the value and will return false.

bool stack_pop(Stack *s, uint32_t *val)

Takes in a stack and a value to pass the popped value into. The function pops(removes) the top value from the stack and passes it into val. The function returns true if this was done successfully, false otherwise.

bool stack_peek(const Stack *s, uint32_t *val)

Takes in a stack and a a value to pass the top value to. The function simply passes the top value of the stack into value, and returns true if this is done successfully. False otherwise

bool stack_empty(const Stack *s)

Takes in a stack, returns true if the top of the stack is equal to 0, false otherwise/

bool stack_full(const Stack *s)

Takes in a stack, returns true if the top of the stack equals the capcity o the stack. Otherwise it returns false

uint32_t stack_size(const Stack *s)

Takes in a stack and returns the number of elements in the stack. The number of elements in the stack is the top of the stack - 1.

void stack_copy(Stack *dst, const Stack *src) Takes in two stack, dst and src. Copies the contents of src into dst.

```
stack_copy:
    if stack_size(dst) is greater than stack_size(src):
        while top value of dst < top value of src:
            set the value of dst to the the value below the top value of src
            increment the top of dst by 1
```

void stack_print(const Stack *s, FILE *outfile, char *cities[]) Not implemented yet/don't understand the implementation provided.

0.2 Path Functions

Path *path_create(uint32_t capacity) Takes in a size and returns a path ADT. A path consists of a stack that is designed to store vertices during a graph traversal, and the total weight of the path.

void path_free(Path **P)

Takes in the address of a path, and frees the memory that the path is occupying.

uint32_t path_vertices(const Path *p)

Takes in a path and returns the number of vertices in the path.

uint32_t path_distance(const Path *p)

Takes in a path and returns the total weight of the path

void path_add(Path *p, uint32_t val, const Graph *g)

Takes in a path, the value of a vertex(in a graphs adjacency matrix) and a graph ADT. The function then updates the total weight of the path with the distance from the top item on the stack to the new vertex being visited. Then the new vertex is pushed to the stack.

uint32_t path_remove(Path *p, const Graph *g)

Takes in a path and a graph. The most recently added vertex is removed from the path, and the total weight is adjusted accordingly. The index of the removed vertex is returned.

void path_copy(Path *dst, const Path *src) Takes in two different paths, dst and src. The path in src is copied to dst, utilizing the function stack_copy to do this.

void path_print(const Path *p, FILE *outfile, const Graph *g) Function not implemented yet.

0.3 Graph Functions

Graph *graph_create(uint32_t vertices, bool directed)

Takes in the number of vertices desired for a graph and whether the graph should be directed or not, and returns a graph with that many vertices.

graph_free(Graph **gp)

Takes in the address of a graph, and frees the memory that the graph is being stored in.

uint32_t graph_vertices(const Graph *g)

Takes in a graph and returns the number of vertices in the graph.

void graph_add_vertex(Graph *g, const char *name, uint32_t v)

Takes in a graph, a string, and an index for the name to be stored at in the array of names. The vertex is then added to the graph.

const char *graph_get_vertex_name(const Graph *g, uint32_t v)

Takes in a graph and the index of a name, and returns the name of the vertex at that index.

char **graph_get_names(const Graph *g) Takes in a graph and returns an array of the names of all the vertices in the graph.

void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight

Takes in a graph, starting and ending vertex indices, and a weight for the edge to be added, and then adds this edge to the graphs adjacency matrix.

uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)

Takes in a graph, and starting/ending vertex indices, then returns the weight of the edge between those two vertices.

void graph_visit_vertex(Graph *g, uint32_t v)

Takes in a graph and the index of a vertex. Marks the vertex as visited by changing the value in the visited array of booleans to true.

void graph_unvisit_vertex(Graph *g, uint32_t v)

Takes in a graph and the index of a vertex. Marks the vertex as not-visited by changing the value in the visited array of booleans to false.

bool graph_visited(Graph *g, uint32_t v)

Takes in a graph and the index of a vertex. Returns true if the vertex has been visisted, false otherwise.

void graph_print(const Graph *g) Takes in a graph and prints a human readable representation of the graph. **uint32_t dfs(Path *p, Graph *g, uint32_t node, Path *best_path)**

The algorithm for this implementation of DFS is listed above in the algorithms section. To run the DFS, the function takes in an empty path which represents the current path in the context of this function, as well as a graph, and a starting node and another empty path called best_path.

Results

The program compiles without errors, and finds paths that match the binary files paths for all the provided graphs. It was interesting to see how quickly the dfs algorithms starts to slow down as the graph gets slightly more complex. For example, clique9.graph runs in less than a second, but clique 12.graph takes about 15 seconds to run, and clique13.graph takes over 2 minutes. This seems to indicate that the run time is non-linear. I think the run time is probably O(n**2) because each recursive call runs through the for loop again.

While working on this program my approach initially was to quickly complete the ADTs and then instantly jump into the main tsp.c file, without writing any tests for the data types. In hindsight this was a very silly idea and I ended up spending multiple hours working on the dfs algorithm only to realize my stack_copy method was not working as intended. For future projects I realize it is very important to use asserts more often to help track where a bug is in the code. On this assignment I started committing and pushing more liberally, and used the pipeline very often to help organize my workflow and figure out what I still needed to implement in my program.

Error Handling

An error I haven't been able to fix in the program occurs when a graph file with contradictory information is passed. For example, if a program only has 3 nodes but for one of the edges you specify node 4 the program will have a segmentation fault. This error, in my mind, is more of a user error though as passing such a value could not possibly result in a valid path/graph.

Numeric results

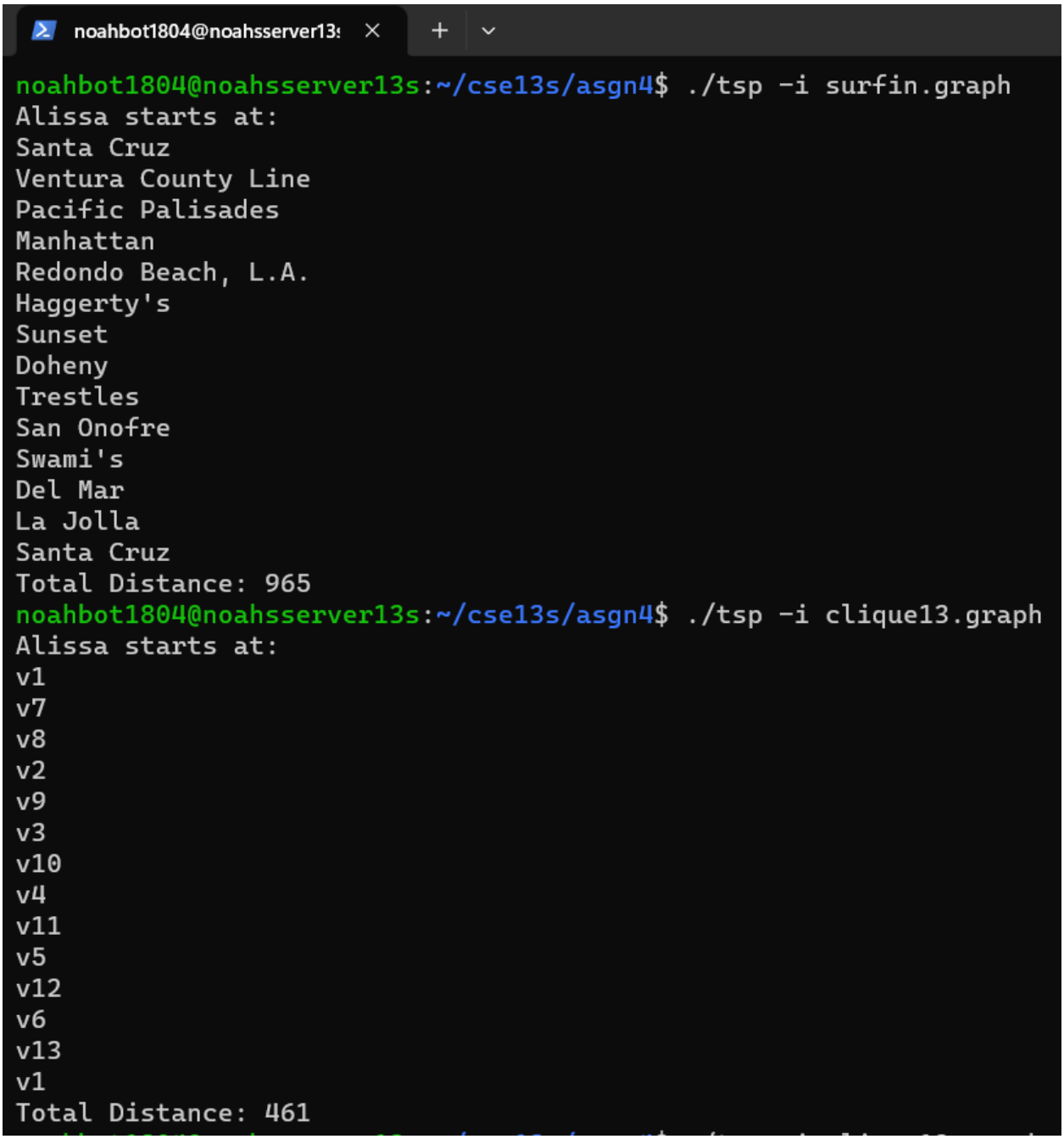

Figure 1: Screenshot of surfin.graph and clique13.graph output