Assignment 6 – Surfin' U.S.A. Design

Caden Roberts

CSE 13S – Winter 24

Purpose

In this assignment we will tackle the classic Travelling Salesman Problem, in our case we are going to find the shortest/easiest path to visit all the common California beaches mentioned in "Surfin' U.S.A." only once, starting and ending in Santa Cruz. Our tsp.c program will utilize user-defined types graph, stack, and path, and then use a dfs algorithm to find the shortest hamiltonian cycle, which will be our solution.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- What benefits do adjacency lists have? What about adjacency matrices?

  ANS

  Traversing an adjacency matrix can be much quicker than traversing an entire adjacency list. The adjacency matrix can be more efficient by saving space.

- Which one will you use. Why did we chose that (hint: you use both)

  ANS

  I will use an adjacency matrix for storing the weights between beaches. A graph is passed in in an adjacency list format.

- If we have found a valid path, do we have to keep looking? Why or why not?

  ANS

  Yes, there will be many valid paths, but only one or few shortest paths.

- If we find 2 paths with the same weights, which one do we choose?

  ANS

  It isn't important for this assignment to choose between the two. We can let our program choose the first shortest as the second won't be short enough to override it.

- Is the path that is chosen deterministic? Why or why not?

ANS

The chosen path is deterministic to each individual, but between individuals, if multiple paths are the same length, it may not be deterministic,=.

- What type of graph does this assignment use? Describe it as best as you can.

ANS

This assignment uses both directed and undirected graphs. Directed graphs have different weight between 1 and 2 and 2 and 1, whereas an undirected graph has the same weight both ways.

- What constraints do the edge weights have (think about this one in context of Alissa)? How could we optimize our dfs further using some of the constraints we have?

ANS

The edge weights can be either directed or undirected, and they should be a reasonable size. Our dfs algorithm could factor a graph being directed vs undirected, and then find the most optimal route in either case.

Testing

I will test many circumstances of directed and undirected graphs, with few and with many vertices. I will test multiple aspects of command line functionality, as well as each function individually to ensure proper functionality.

How to Use the Program

Running ./tsp *commands* maps/map.graph is the basic format to run our tsp program on the command line.

Program Design

We utilize user-defined types graph, stack, and path. These structs rely on arrays, arrays of arrays, chars, uint32_t's, and bools. The dfs algorithm will do the leg work of finding the shortest path through a given graph, calling itself recursively among other functions that manipulate path and graph.

Pseudocode

Psuedocode for my tsp logic is as follows:

```
# Define constants
START_VERTEX = 0

# Define function to perform depth-first search (DFS)
function DFS(graph, vertex, current_path, optimal_path):
    Add vertex to current_path
    Mark vertex as visited

    # If current path is a complete path covering all vertices
    if (number of vertices in current_path == total vertices in graph) and
       (there exists an edge from current vertex to start vertex):
        # If no optimal path found yet, or if current path is shorter than the optimal path
        if (number of vertices in optimal_path == 0) or
```

```
           (length of current_path < length of optimal_path):
               Copy current_path to optimal_path
           # Else, do nothing since current_path is not shorter than the optimal path

       # Explore unvisited neighbors of current vertex
       for each neighbor of vertex in graph:
           if (neighbor is unvisited) and (there exists an edge from vertex to neighbor):
               DFS(graph, neighbor, current_path, optimal_path)

       Remove vertex from current_path
       Mark vertex as unvisited

# Main function
function main():
    Set default values for input and output files
    Parse command-line arguments
    Open input and output files
    Read number of vertices from input file
    Create graph with given number of vertices
    Read vertex names and add them to the graph
    Read number of edges from input file
    Add edges to the graph
    Create empty current_path and optimal_path
    Perform DFS starting from START_VERTEX
    if (optimal_path is not empty):
        Add start vertex to optimal_path
        Print "Alissa starts at:"
        Print optimal_path
        Print "Total Distance: distance of optimal_path"
    else:
        Print "No path found! Alissa is lost!"
    Close input and output files
    Free memory allocated for paths and graph
```

Function Descriptions

- Graph *graph_create(uint32_t vertices, bool directed); Takes a uint32_t and a boolean to initialize the values of the graph struct, returning a pointer to a graph.
- void graph_free(Graph **gp); Takes a pointer to a pointer to a Graph and frees all memory associated with the graph and sets the pointer to the Graph to NULL.
- uint32_t graph_vertices(const Graph *g); Takes a pointer to a graph and returns the number of vertices in that graph.
- void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight); Takes a pointer to a graph and adds an edge of weight between start and end.
- uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end); Takes a pointer to a graph and return the weight between start and end.
- void graph_visit_vertex(Graph *g, uint32_t v); Takes a pointer to a graph and a uint32_t v and updates the visited array of g to true for that vertex.
- void graph_unvisit_vertex(Graph *g, uint32_t v); Takes a pointer to a graph and a uint32_t v and updates the visited array of g to false for that vertex.
- bool graph_visited(const Graph *g, uint32_t v); Takes a pointer to a graph and a uint32_t v and returns the visited array of g for that vertex.

- char **graph_get_names(const Graph *g); Takes a graph pointer and returns the array of strings of vertex names for that graph.
- void graph_add_vertex(Graph *g, const char *name, uint32_t v); Takes a graph pointer and assign a string name to a vertex v.
- const char *graph_get_vertex_name(const Graph *g, uint32_t v); Takes a graph pointer and returns the string associated with the vertex v.
- void graph_print(const Graph *g); Takes a pointer to a graph and prints the contents of that graph.
- Stack *stack_create(uint32_t capacity); Takes a uint32_t and initializes the variables associated with a Stack, returning a Stack pointer.
- void stack_free(Stack **sp); Takes a pointer to a pointer to a Stack and frees all the memory assosciated with the Stack, and sets the pointer to the Stack to NULL.
- bool stack_push(Stack *s, uint32_t val); Takes a pointer to a stack and a val and pushes the val onto the stack. Returns if the operation was successful or not
- bool stack_pop(Stack *s, uint32_t *val); Takes a pointer to a stack and a pointer to a uint32_t, and removes the top item from the stack, updating the uint32_t to the value of the popped item and returning whether or not the operation was successful.
- bool stack_peek(const Stack *s, uint32_t *val); Takes a pointer to a stack and a pointer to a uint32_t, and copies the top item from the stack to the uint32_t, returning whether or not the operation was successful.
- bool stack_empty(const Stack *s); Takes a pointer to a stack and returns whether or not the stack has 0 items in it.
- bool stack_full(const Stack *s); Takes a pointer to a stack and returns whether or not it has s-¿capacity items in it.
- uint32_t stack_size(const Stack *s); Takes a pointer to a stack and returns how many items are currently in the stack (s-¿top).
- void stack_copy(Stack *dst, const Stack *src); Takes a pointer to a stack and another pointer to a stack and copies the first stack onto the second.
- void stack_print(const Stack *s, FILE *f, char *vals[]); Takes a stack pointer, a pointer to a File, and an array of strings associated with the vertices contained in the stack, printing the contents of the stack.
- Path *path_create(uint32_t capacity); Takes a uint32_t and initializes the assosciated variables of a Path, returning a pointer to the created Path.
- void path_free(Path **pp); Takes a pointer to a pointer to a path and frees all the assosciated memory, setting the pointer to the Path to NULL.
- uint32_t path_vertices(const Path *p); Takes a pointer to a path and returns the number of vertices currently present in the path.
- uint32_t path_distance(const Path *p); Takes a pointer to a path and returns the total weight associated with the current path (p-¿total_weight).
- void path_add(Path *p, uint32_t val, const Graph *g); Takes a pointer t a path and a uint32_t, adding the vertex and the weight of that vertex from the Graph pointer to the path.
- uint32_t path_remove(Path *p, const Graph *g); Takes a path pointer and a graph pointer and removes the most recent vertex from the path, returning the value/index of the removed vertex.

- void path_copy(Path *dst, const Path *src); Takes a pointer to a path and another pointer to a path and copies the second path to the first.
- void path_print(const Path *p, FILE *f, const Graph *g); Takes a pointer to a path, a pointer to a File, and a pointer to a Graph, and prints the contents of the path to the File.