Assignment 7 – Assignment 7 Report

Max Ratcliff

CSE 13S – Spring 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

Questions

- What benefits do adjacency lists have? What about adjacency matrices? **Adjecency lists are faster on a smaller data set, where theres less items to traverse, becuase they are relatively easy to traverse, but they have to scan the whole list to find the item they want to access, they are also easier to add to as you can just allocate more memory. Matrices on the other hand are faster with large datasets since you can easily acess individual entries, but they are hard to dynamically increase since you have to add to both the rows and collumns of the matrix**
- Which one will you use. Why did we chose that (hint: you use both) **the .graph files are formatted as an adjecency list and we will read from them to define our struct, but I will use the matrix in my main program cause we only need to initialize the graph at the begining and dont need to add to it which will make implementation and accessing the elements easier.**
- If we have found a valid path, do we have to keep looking? Why or why not? **we have to keep looking until we find the shortest path.**
- If we find 2 paths with the same weights, which one do we choose? **we will use the one we found first because once we find a path that works we'll only replace it with something shorter, ignoring other options with the same or bigger weight.**
- Is the path that is chosen deterministic? Why or why not? **yes it will be deterministic because we will always traverse the matrix in the same order. and choose the most optimal path based on the same criteria**
- What type of graph does this assignment use? Describe it as best as you can **this assignemnt uses either a directed or an undirected graph with a hamiltonian cycle and only positive edges.**
- 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? **since we're looking for the most efficient path we can skip iterations of our dfs if the path were currently investigating is longer than one we've already found.**

Testing

I will test to make sure all the flags work as intended, and make sure my program handles bad graph input as well as very large and very small graphs. I will test to make sure it throws the proper errors when it cant find a hamiltonian cycle as well as tests for to make sure each function works as intended.

How to Use the Program

to use this program, first compile with make, and then run with `.\tsp` **with the optional arguments** `'-i' to get input from a file other than stdio, '-o' to write to a file other than stdout, '-d' to treat graphs as directed instead of undirected, '-h' to print the help message to sdtio` **the program will then print the shortest path to either stdout or the file specified in the arguments**

Program Design

the main program will be written in tsp.c which will implement the traveling salesman algorithm, this program will depend on the implementations of the graph, stack and path abstract data types which will be implemented in graph.c, stack.c and path.c respectively and will contain helper functions to help tsp interact with them.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

0.0.1 graph_create

- Inputs: 32 bit int for vertices, and boolean for weather its directed or not
- Output: an an initialized Graph ADT
- initializes memory for the graph and sets its custom data fields for vertices and directed, then allocate memory for an array for visted vertices, names of vertices, and weights of edges as well as the vertices x vertices adjacency array.

0.0.2 graph_free

- Inputs: double ptr to a graph data type
- Output: all memory freed and ptr set to null
- frees all the allocated memory of the graph and sets the original pointer to null
- free the memory allocated to the visted, names, and weights array, then freethe adjecency matrix, then free the memory allocated to the graph and set its pointer to null.

0.0.3 graph_vertices

- Inputs: pointer to a graph
- Output: number of vertices in the graph
- helper function to return the count of the vertices of a graph so it doesnt have to be directly accessed via pointers

0.0.4 graph_add_vertex

- Inputs: pointer to a graph, name of the vertex, index in the name array where the vertex should be added
- Output: updated graph
- checks if the vertex already exits, if so free the allocated memory and set the name of the vertex to the name passed into the function

0.0.5 graph_get_vertex_name

- Inputs: pointer to a graph and the index to the vertex whose name we're getting
- Output: name of the vertex at intex v
- returns the name of the vertex v

0.0.6 graph_get_names

- Inputs: pointer to a graph
- Output: double pointer/an array of strings
- returns an array of all the names of the vertices in the graph

0.0.7 graph_add_edge

- Inputs: pointer to a graph, start vertex, end vertex, wieght of the edge
- Output: updated graph with edge added
- adds the edge to the list of edges of the graph

0.0.8 grap_get_weight

- Inputs: pointer to a graph, start vertex, end vertex
- Output: int representing the weight of the edge between the two vertices
- looks up the weight of the edge between the start and end vertices

0.0.9 graph_visit_vertex

- Inputs: pointer to a graph, index of a vertex
- Output: updated graph
- adds the vertex to the list of visited vertices

0.0.10 graph_unvisit_vertex

- Inputs: pointer to a graph, index for vertex to unvisit
- Output: updated graph
- remove the vertex from the list of visited vertexes

0.0.11 graph_visited

- Inputs: pointer to a graph and the index to the vertex that were checking if its been visited
- Output: true/false value
- returnes true if vertex v is visited and false otherwise

0.0.12 stack_create

- Inputs: 32 bit int for capacity
- Output: an an initialized stack ADT
- initializes memory for the stack and sets its custom data fields for capacity, top, and items

0.0.13 stack_free

- Inputs: double ptr to a stack data type
- Output: all memory freed and ptr set to null
- frees all the allocated memory of the stack and sets the original pointer to null
- free the memory allocated to items array

0.0.14 stack_push

- Inputs: pointer to a graph, and value to push to the top
- Output: updated stack
- adds the value to the satck at the index of top, then increment top

0.0.15 stack_pop

- Inputs: pointer to a stack and pointer to the value to receive
- Output: success state and pointer updated to the value popped
- updates the pointer to the value at the top of the stack, then decrement top.

0.0.16 stack_peek

- Inputs: pointer to a stack and pointer to the value to receive
- Output: success state and pointer updated to the value popped
- updates the pointer to the value at the top of the stack.

0.0.17 stack_empty

- Inputs: pointer to a graph
- Output: true/false
- returns true if the stack is empty false otherwise

0.0.18 stack_full

Inputs: pointer to a graph
Output: true/false
returns true if the stack is full false otherwise

0.0.19 stack_size

- Inputs: pointer to a graph
- Output: int representing the size of the stack
- returns the size of the stack

0.0.20 stack_copy

- Inputs: pointer to a destination stack, pointer to a source stack
- Output: updated destinantion stack, unmodified source stack
- iteravvly copies items from one stack to the next

0.0.21 stack_print

- Inputs: pointer to a stack, pointer to a file, pointera an array of strings
- Output: output file filled
- iterates through the array and print them to the outfile

0.0.22 path_create

- Inputs: 32 bit int for capacity
- Output: an initialized path ADT
- initializes memory for the path and calls stack_creates initializes distance to 0

0.0.23 path_free

- Inputs: double ptr to a path data type
- Output: all memory freed and ptr set to null
- frees all the allocated memory of the path and then call stack_free sets the original pointer to null

0.0.24 path_vertices

- Inputs: pointer to a path
- Output: returns the number of vertices
- accesses the stack and calls the stack_size funtions

0.0.25 path_distance

- Inputs: pointer to a path
- Output: int representing the total distance of the path
- returns the value stored in total_distance

0.0.26 path_add

- Inputs: pointer to a path, pointer to a graph, vertex
- Output: updated path
- adds reads vertex v from the graph and push it to the stack, then update total distance

0.0.27 path_remove

- Inputs: pointer to a path, pointer to a graph
- Output: updated path
- pops the first value from the stack and updates the total distance

0.0.28 path_copy

- Inputs: pointer to a destination path, pointer to a source path
- Output: updated destinantion path, unmodified source path
- iteravly copies items from one path to the next

0.0.29 stack_print

- Inputs: pointer to a path pointer to a file, pointer to a graph
- Output: output file filled
- prints path to the outfile and then call stack_print

0.0.30 tsp

- Inputs: pointer to a graph, int for a vertex, pointer to a current path, pointer to a best path
- Output: updated best path
- implements a DFS algorithm exploring every path, and then after each path is fully explored check if its shorter then the best path found so far and update the shortest path.