Assignment 6 – Surfin' U.S.A.

William Zhou

CSE 13S – Winter 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.

The purpose of this assignment is to implement a graph that represents the beach locations that Alissa wants to visit. DFS will be used to determine the fastest route to start from Santa Cruz, go through each city, and arrive back at Santa Cruz.

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?

Adjacency lists are more space efficient as unlike adjacency matrices, there is no redundant information. They are more dynamic as edges can be added and removed without having to alter anything else.

Adjacency matrices have the benefit of being more visually accessible. If you wanted to see if an edge existed between 2 edges, a quick look at the table will tell you the answer. If you were to try to do the same thing in a list, you would have to look through every entry in the list to make sure.

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

The adjacency list is used to store the graph. The adjacency list is used internally inside the graph to store linked lists which contain the weights.

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

We definitely have to keep looking. We are not trying to just find a route that works, we want to find the route with the least weight to minimize gas usage.

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

It should choose the first path of the lowest length that our program finds.

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

If there is only one optimal answer the path that is chosen is deterministic. However, if there are multiple optimal paths, then the path chosen will depend on how the program chooses between tie breakers. In this case, it is not deterministic and will vary from implementation to implementation.

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

This assignment uses an undirected graph. The points will represent cities, and the edges will represent the path to traverse the corresponding cities. In this assignment will use this graph to determine the Hamiltonian cycle with the smallest weight that starts from Santa Cruz.

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?

We only want to visit each city once, meaning that we will never traverse an edge more than once. We could optimize our dfs further by making it so once an edge has been traversed we can't traverse it again. This constraint should reduce the number of possibilities the program will test.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags that your program uses, and what they do.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[1]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

To use the program, please compile the code with the makefile by typing in "make" in the console. To execute the code, type "./tsp" plus some parameters.

Here are the parameters available to you:

-i : Makes it so you can provide a filename to read from.

-o: Makes it so you can provide a filename to write out to.

-d: Treats graphs as directed. Will treat graphs as undirected if not specified.

-h: Prints a help message to stdio.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

[1] This is my footnote.

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.

My main function and my primary program functionalities will be in tsp.c. Memory will be freed once program finishes running.

Graph.c will implement the graph datatype defined in graph.h. Included are functions to create, remove, adn edit graphs.

Stack.c will implement the stack datatype defined in stack.h. Included are functions to create, remove, and edit stacks.

Path.c will implement the stack datatype defined in path.h. Included are functions to create, remove, and edit paths.

Function Descriptions

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

- The inputs of every function (even if it's not a parameter)
- The outputs of every function (even if it's not the return value)
- The purpose of each function, a brief description about a sentence long.
- For more complicated functions, include pseudocode that describes how the function works
- For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge.

Functions in graph.c:

Graph *graph_create(uint32_t vertices, bool directed):

Takes in a uint32_t which represents the number of vertices the graph will have. Also takes in a boolean which determines whether the graph is directed or undirected. Function graphs a graph, and then returns a pointer to it. All itmes in the visited array are initialized to false. The purpose of this function is to create an graph that we can use to solve the Hamiltonian cycle.

void graph_free(graph **gp):

Takes in a double pointer to a graph. Returns nothing. The purpose of this function is to free up all memory used by the graph once it is done being used. It also sets the graph pointer to null.

uint32_t graph_vertices(const Graph *g):

Takes in a pointer to a graph. Outputs the number of vertices in the graph. Purpose is pretty simple, to inform the user of the number of vertices in the graph they enter.

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

Takes in a pointer to a graph, as well as a pointer to a name, and a uint32_t. Returns nothing. The function's purpose is to copy the char given and store it inside the graph.

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

Takes in graph pointer and uint32_t. Returns the name of the city from the uint32_t vertex. Graph is not modified.

char *graph_get_names(const Graph *g):

Takes in graph pointer. Returns a double pointer to an array of strings. The purpose of this function is to take in a graph and return a double pointer to an array with the names of every city in an array.

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

Takes in graph and takes in 3 ints. Returns nothing. Purpose of this function is to place an edge whose starting and ending point corresponds to the first 2 ints. The 3rd int will become the weight of the edge.

uint32_t graph_get_weight(const Graph *g, uint32_t start, utin32_t end):

Takes in a graph pointer with 2 ints. Returns an int. Looks at the graph and returns the weight of the edge whose edges correspond to the 2 ints taken in.

void graph_visit_vertex(Graph *g, uint32_t v):

Takes in a graph pointer and an int. Returns nothing. Adds vertex of int parameter to the list of visited vertices. This is so we can keep track of which vertexes have been visited.

void graph_unvisit_vertex(Graph *g, uint32_t v):

Does the same thing as the previous function, but removes from list instead of adding.

bool graph_visited(const Graph *g, uint32_t v):

Takes in a graph pointer and int. Returns a boolean. Purpose of the function is to check if the corresponding vertex inside the graph has been visited. Returns true if this is the case, and returns false otherwise.

Functions in stack.c:

Stack *stack_create(uint32_t capacity):

Takes in a int. Returns a stack pointer. Creates a stack object, dynamically allocates space for the stack object and returns a pointer to it.

void stack_free(Stack **sp):

Takes in a double pointer to a stack. Returns nothing. Purpose is to free up all the memory being allocated by the stack. Also sets the pointer to NULL.

bool stack_push(Stack *s, uint32_t val):

Takes a stack pointer and an int. Returns a boolean. Purpose is to add the int value onto the stack and increments the counter. If this process is successful it returns true, otherwise it returns false.

bool stack_pop(Stack *s, uint32_t *val):

Takes in stack pointer and int. Returns a boolean. Sets the integer val is pointing to to the top item on the stack and removes it. If this process is successful it returns true, otherwise it returns false.

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

Takes in stack pointer and int. Returns a boolean. Sets the integer val is pointing to to the last item on the stack. Does not make any modifications to the stack. If this process is successful it returns true, otherwise it returns false.

bool stack_empty(const Stack *s):

Takes in a stack pointer. Returns a boolean. Returns true if stack is empty and false otherwise.

bool stack_full(const Stack *s):

Takes in a stack pointer. Returns a boolean. Returns true if stack is full and false otherwise.

uint32_t stack_size(const Stack *s):

Takes in a stack pointer and returns an int. Returns the number of elements in the stack.

void stack_copy(Stack *dst, const Stack *src):

Takes in 2 stack pointers. Returns nothing. Overwrites the stack pointed to by *dst with the items from the stack pointed to by *src. Also updates dst-¿ top.

void stack_print(const Stack* s FILE *outfile, char *cities[]):

Takes in a stack pointer, a file, and a pointer to an array of characters. Returns nothing. Purpose of the function is to print in a list format all the elements in the stack alongside the city names provided.

Functions in path.c:

Path *path_create(uint32_t capacity):

Takes in an int. Returns a path object. Purpose is to create a path with a stack and a weight of zero.

void path_free(Path **pp):

Takes in a double pointer to a path. Returns nothing. Frees the memory allocated by path.

uint32_t path_vertices(const Path *p):

Takes in a path pointer. Returns an int. Purpose is to return the number of vertices in the path provided.

uint32_t path_distance(const Path *p):

Takes in a path pointer. Returns an int. Purpose is to return the distance covered by the path provided.

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

Takes in a path pointer, an int, and a graph pointer. Purpose is to add vertex of the int provided from graph g onto the path provided. It will also update the distance and length of the path. Distance will remain zero when the first vertex is added. Subsequent vertexes will increase distance and total weight.

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

Takes in a path pointer and graph pointer. Returns an int. Removes the most recent vertex from the path. Distance and length of the paths are updated accordingly.

void path_copy(Path *dst, count Path *src):

Takes in 2 path pointers. Returns nothing. Copies the path from src to dst.

void path_print(const Path *p, FILE *outfile, const Graph *g):

Takes in a path pointer, file pointer, and graph pointer. Returns nothing. Purpose is to Get the path stored and print them to outfile using the vertex names from the graph.

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.