Assignment 2 – Hangman Report Template

Your Name

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.

This program makes use of the depth first search algorithm and graph theory to find the shortest path between a set of beaches. Any given beach will only be visited once, and all will be visited. This path will start and end at a given location in the graph. If such a path cannot be found it will state "Alyssa is lost!". If such a path can be found it prints the shortest path to the outfile.

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 have the perks of not storing non existent paths between two locations, which saves on memory. Adjacency matrices have the benefit of being more search friendly, meaning that is does not take as long to find a particular item. Also good if you just want to draw it out.
- Which one will you use. Why did we chose that (hint: you use both) My program strictly used the adjacency matrix, just because it made more sense to me, and the type of data structure I wanted to use. It would have been more difficult for me to program an adjacency list, and of course when it comes to these assignments, time is everything. I understood the matrix more, so I went with it.
- If we have found a valid path, do we have to keep looking? Why or why not? Yes indeed we need to keep looking. Not only are we trying to find A path, but we are trying to find the most efficient one. The first path you find is not necessarily the best one.
- If we find 2 paths with the same weights, which one do we choose? Depends on the situation. If you still need to complete your path search, you explore both of them. If the entire path has been figure out, and both only visit each beach once and end in Santa Cruz, you pick the 1st one.
- Is the path that is chosen deterministic? Why or why not? Yes, because the search for the path is structured in a very sequential specific way, there are no random elements to this process. Each node has a list of connecting nodes, and so on. Each node on the list is tested, before it backs out to the previous node, and so on, and the program does not stop until every path has been inspected.
- What type of graph does this assignment use? Describe it as best as you can Well it can use either a directed or un directed graph. An undirected graph allows you to travel between 2 nodes both ways with the same weight. In a directed graph, you might not be able to travel back at the same weight, or at all. This assignment defaults to undirected.

- 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? Edge weights make traveling different directions in our graphs more or less efficient, especially if we are dealing with directed graphs. Automatically eliminating repeating nodes in a path is a good start, since it costs extra to go back and forth. Then Eliminating cycles will also be efficiant (besides the main hameltonian cycle), because if you have a cycle, you have a set of repeating nodes, which as discussed earlier are non efficiant.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Remember that you will not be getting a reference binary [2].

First I will start by testing the data structures, since everything depends on those functions working properly. Just writing simple c tests checking if the various functions return the correct values in specific situations should be fine enough. Then, since we were given a few example graph files to work with, using those as inputs and seeing how the program responds could be helpfully as well, since they are relatively small and you can deduce what the correct answer should be on your own. Perhaps I can also make my own graph file and test that too.

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 or inputs that your program uses, and what they do.

First you will need a file to input. This file must consist of various sections. The first section is just the first line, which states the number of vertices ex) 3. The next section are a number of lines, stating the vertice names, each seperated with a newline. The next section is one line stating the number of edges. The final section constains lines that are structured like start end weight. Each of these must be numbers seperated by spaces. The order the vertices were inputted indicate their numerical value. After each weight you should but a newline. Then there is the matter of command line options.You can just input ./tsp and tsp will read in from stdin and write to stdout. BUT, if you would like to write to a specific file, do -o file.name. If you would like to read in from a specific file do -i file.name. If you want the graph you make to be directed, do -d. And finally, if you need help with anything, do -h to print the help message.

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].

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.

(not a section??)

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.

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

[2] The output of your binary is not the only thing you should be testing!

The keystone of this program is the stack data structure. The stacks function is to store values in a list like structure, and manipulate the items on command. All stack related functions and structs can be found in stack.c and stack.h. From there, the path data structure relies on the stack structure, and anything involved with path can be found in path.c and path.h. Path is intended to store graph nodes, and the sum of the weights between them. Then the most complicated data structure of them all is graph. Of course any and all graph related functions, structures, or problems can be found in either graph.c or graph.h. The graph structure contains all of the nodes, and the weights between them, in am adjacency matrix. An additional file that I added to this assignment is bonus-functions.c and .h. This is where the dfs and all visited function can be found. DFS is by far the most complex function, and it depends on all visited, in addition to many graph functions. Finally, there is tsp, which contains the main program. First it established the command line options. Depending on which ones were picked, this will alter the read in and out files, but no matter what the file pointer will be stored in fi and fo. Then the read in section begins. If there are any issues with the read, the program will exit with a value of 1. To note, most of the major sections in tsp are broken up by large comment lines for readability. During the read in section, the graph is also created from the inputs from the file. Num vertices, names of vertices, num edges, and start end weight lines are all read in and assigned to their respective graph functions. Then the file is closed and we move on to the dfs section. dfs is called and the shortest path is assigned to the winner path pointer. If a path was actually found, the statement "Alyssa starts at: // winning path vertices in order visited// Total distance: (the distance)" is printed to the designated outfile. If a suitable path was not found, ""No path found! Alissa is lost!" is printed to the outfile instead. Then everything that was allocated during the program is freed and the program ends with a return 0.

Function Descriptions

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

- graph create: Takes a unit32-t in the form of vertices, and a boolean called directed. Returns a graph. Makes a new graph struct, and sets all of the items in visited to false. Returns a pointer to the graph.
- graph free: Takes a pointer pointer to a graph, frees all of the memory tied to the graph, and returns nothing.
- graph vertices: takes a pointer to a graph, finds all of the vertices in the graph, returns this number as a unit32-t.
- graph add vertex: takes a pointer to a graph, a character pointer called name, and a unit-32 called v. Gives the city at the given vertex (v) the name passed in . Basically naming a vertex. Also copies the name and stores it in the graph. Returns nothing.
- graph get vertex name: Takes a pointer to a graph and a unit32 t vertices. figures out the name of the vertex inputted from the names list. Returns a pointer to the name.
- graph get names: takes a pointer to a graph and returns a char pointer pointer. Returns the pointer to the array of all the names. An array of strings is a double pointer.
- graph add edge: takes a pointer to a graph, and 3 unit32-t values (start end weight). creates an edges between the two points, with the given weight, to the matrix. Returns nothing
- graph get weight: takes a pointer to a graph and two unit32-t values (start and end). Figures out the weight between the two
- graph visit vertex: takes a pointer to a graph and a unit32-t value (v). Adds vertex v to the visited list, returns nothing :3
- graph un-visit vertex: takes a graph pointer and a unit32-t value (the vertex). Takes the vertex off of the visited list, returns nothing

- graph visited: takes a graph pointer and a unit32-t value (the vertex). Checks if the vertex has been visited in the graph. Returns a bool true if it has been, and false if not.
- stack create: Takes a unit32-t value (capacity). Allocates the memory for the stack size, and returns a pointer to the new stack
- stack free: Takes a pointer to a pointer to a stack, frees all of the memory associates with it, sets the pointer to NULL, and returns nothing.
- stack push: takes a stack pointer and a unit32-t value. Adds this value to the top of the stack, and updates the top pointer variable. Returns true if it worked and false if it did not.
- stack pop: takes a stack pointer and a unit32-t value pointer. Changes the value pointer to whatever is on the top of the stack and decrements the top pointer. Returns true if it worked and false if it didn't.
- stack peek: takes a stack pointer and a unit32-t value pointer. Sets the value pointer to the top of the stack and does not decrement it. Returns true if it worked and false if it didn't.
- stack empty: takes a stack pointer and checks if the stack is empty. Returns true if it is and false if it isn't.
- stack full: takes a stack pointer and checks if the stack is full. Returns true if it is and false if it isn't.
- stack copy: takes two stack pointers and returns nothing. Copies the exact verticies from src and writes it into dst. Also copies over the path weight of src.
- stack print: Takes a stack pointer, a file pointer, and a char pointer to an array. Prints the name of every vertice in stack to the designated file. Returns nothing.
- capacity finder: takes a stack pointer and return a uint32-t value. Intended to return the capacity of the stack
- path create: Takes a unit32-t value (capacity). Allocates the memory for the path size, and returns a pointer to the new stack. Also sets path weight to 0.
- path free: takes a double pointer to a path and frees all associated memory with it. Returns void.
- path vertices: takes a path pointer and returns a uint32-t value. Returns how many vertices there are.
- path distance: takes a path pointer and returns a uint32-t value. returns the total weight of the path.
- path add: takes a path pointer, a graph pointer, and a uint32-t value. returns void. Adds a vertice to the path, and adds its weight cost to total weight.
- path copy: Takes two path pointers (dst and src), returns void. copies all vertices and weight from one path to another.
- path remove: takes a path pointer, a graph pointer. Returns a uint32-t value. removes the last vertice from the path and subtracts its weight cost from total weight. returns the vertice removed.
- all visited: takes a graph pointer and returns a boolean. Checks if every vertice in a graph has been visited.
- dfs: RECURSIVE function. takes a uint32-t value (target), a graph pointer, and two path pointers. Target is the vertice that is currently being analysed, and the two paths are for storing the winning path, and the current path. Essentially, the target vertice is marked as have been visited. Then it checks all of its adjacent vertices. If the path between a vertice and our target is 0, it is not adjacent. Then it checks if it has already visited the adjacent vertice. if it has, it skips it, if it has not, it calls dfs with that vertice being the target. There is one exception to this. If the starting vertice appears as an adjacent vertice, and all other vertices have been visited, it till add it to the path, and compair it to winning paths. If it is shorter than the current winning path, it will be copied into winner. Then it will go back a node and the cycle will continue :).

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.