Jessie Zhu jelzhu@ucsc.edu CSE 13s, Spring 2021 Monday, April 26, 2021

Assignment 4: The Circumnavigation of Denver Long

Design Document

- In this assignmnet, we are writing a program to find an optimal route on Denver Long's trip to all of the cities along his route and return him home to Clearlake.
- The purpose of this assignment is to familiarize us with different data structures, specifically we are using graphs and stacks. We will also be using recursion as our method for searching through the graph and keeping track of visited cities.
- I learned a lot about how dynamically allocated data differs from static data and how important it is to free dynamically allocated data to prevent memory leaks.

Design Process:

- Originally I was super confused with how the adjacency matrix was supposed to translate into a path map. (I had originally thought the matrix was the map of cities and that each non-zero item in the map was a city (vertex), silly me). After Eugene's section, it made more sense to me and so I made a lot of changes to my design doc and how I used the matrix.
- I followed the stack code from previous assignments so building the graph, stack, path based on the brief explanation in the document wasn't challenging. Header files with function parameters specified were extremely helpful. (Stack was from the stack code that was provided directly in previous lesson slides or previous assignment docs)
- Getting the recursion to work went smoother than expected, I followed the pseudocode and added the necessary if statements for the base case of recursion. The first time I got it to work, the obtained shortest path was correct, however my recursion counts were off and verbose printing was printing too many sequences. I added more if conditions to reduce the amount of unnecessary looping happening.
- Many errors and memory leaks were found in valgrind tests, so I had to add deletes and frees everywhere there was a return. This fixed the problem but it was repetitive so I put it in a function (cough cough missed points for being repetitive in asgn2). After this there were still many errors, saying not initialized values were being used. I ended up fixing this by initializing values an actual value (rather than just leaving it as: int x).

Graphs:

We will be using graphs to representing the possible paths of the locations. The graph will be represented using an adjacency matrix M, to show paths going from vertex i to j. Each edge is represented in <i, j, k> where k = M [i][j]. Valid edges have k integer values greater than or equal to 1. If the graph is undirected, then the graph should be symmetrical down the diagonal.

Our graph in this assignment is going to have properties:

- vertices: number of vertices
- Undirected: boolean of undirected? (aka do we mirror about the diagonal)
- visited[VERTICES]: keeps track of the places we have gone, array of bool values for each vertex
- matrix[VERTICES][VERTICES]: the adjacency matrix with dimensions of VERTICES by VERTICES

** START_VERTEX and VERTICES are all given to us**

With functions:

- graph_create (vertices, undirected) : The constructor for a graph. Undirected? All cells of matrix is all zeroed, visited array is all false. Vertices field reflects number of vertices in the graph
- graph_delete: destructor when done frees pointer
- graph_vertices: returns number of vertices in the graph
- graph_add_edge (i, j, k): Adds edge at (i, j) with weight k in matrix. If graph is undirected, also add edge at (j, i). Returns true if both vertices are within bounds and added successfully
- graph_has edge(i,j): returns true if vertices I and j are within bounds and if there is an edge at (i,j). Check if i and j are less than graph vertices and if the edge is non-zero. Otherwise return false
- graph_edge_weight(i, j): returns length of path from vertex i to j. If graph does not have this edge, return 0
- graph_visited (v): return bool from visited array at vertex v

- graph_mark_visited(v): if vertex is in bounds, mark visited at vertex v true.
- graph_print: will print the graph

File text -> Graph/Map:

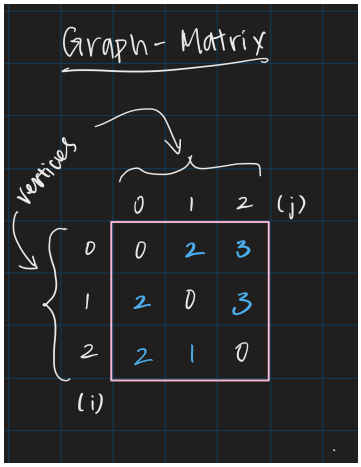
Our graph will be made by taking the input (file), such as the one provided below. In the example input file (left), annotated in red is what each line of the file means and what its purpose serves. Graph will put the <i, j, k> values into a matrix (right).

ucsc.graph 63 Bytes

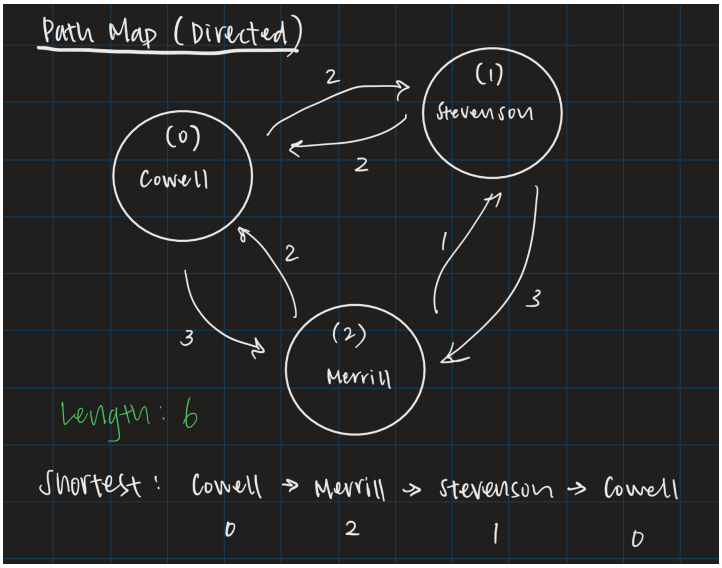
```
3 → # of cities (aka # of vertices)
Cowell → city (vertex 0)
Stevenson → city (vertex 1)
Merrill → city (vertex 2)
0 1 2
0 2 3
1 0 2
1 2 3
2 0 2
2 1 1
```

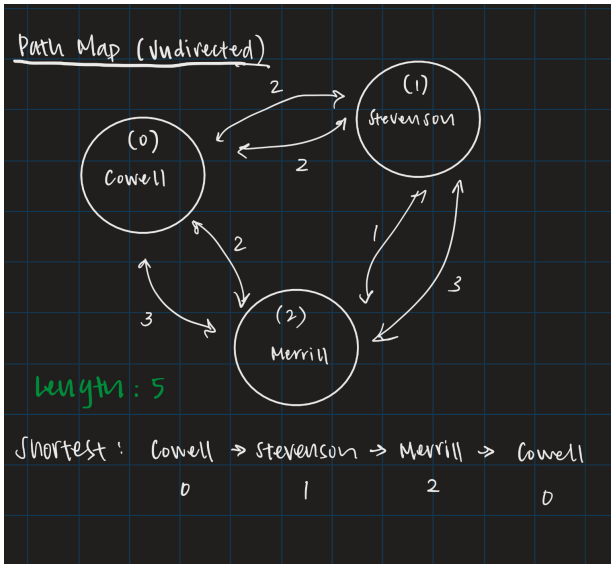
→ i j k

vertex i to vertex j edge with weight k.



Visually, the map of possible paths this example creates looks like this:





(left is directed map, right is undirected map)

Paths:

When we iterate through the the traveled path, vertices are going to be stored in a stack, defined below. Stack will hold all the vertices in a traveled path and length will keep track of the length of a path.

Attributes for paths: stack for vertices, length.

Methods:

- path_create: constructor for the path. Stack for vertices will have size of VERTICES. Length at start is 0.
- path_delete: destructor for a path. Sets pointer to null
- path_push_vertex (v, *G): adds v to stack, increases length of the path by the edge that is being pushed
- path_pop_vertext(*v, *G): removes item from stack, decreases length of path by the vertex that was just removed.
- path_length: return length of the path
- path_copy(*dst *src): makes a dst copy of the src path. Call stack_copy to take a copy of the vertices stack, copy length of the source path

- path print (*outfile, char *cities) : prints out path to outfield using fprintf(). Calls stack_print to print out the contents of the vertices stack

Stack:

We will be using the stack inside the paths to keep track of the current vertices in a path.

All same stack functions as previous assignment except:

- stack_peek (*x) - points pointer to the top item of stack, returns false if stack is empty
- stack_copy(*dst, *src) - will copy the stack's items from *src to the *dst items. All struct variables should match as well in the copy
- stack_print (outfile, cities) - print function is already given to us in asgn doc.

Jessie Zhu jelzhu@ucsc.edu CSE 13s, Spring 2021 Monday, April 26, 2021

DFS/Main:

We will be be using Depth-first Search to iterate through all of the edge.

This recursive call will search through the graph without revisiting a vertex and we will take the generated paths, and find the one that is the shortest. If a found path is shorter than our previously found shortest path, make a copy of the current path into the shortest path holder. Then at the end, our shortest path will be in our shortest path holder.

```
Global var: recursion
DFS (curr path, shortest path, verbose, cities, outfile):
        increment recursion counter
        mark v as visited, add v to current path
        if current path is done (and has not gone back to origin):
                add origin to path
                if current path is shorter than previously shorter path:
                        copy current path into shorter, print path if verbose
                pop last vertex from path
        for each edge adjacent to v:
                if vertex is not visited and current path is shorter than shortest path:
                        recursively call DFS(G, w)
        mark v as unvisited, remove from current path
        return
```

Helper function “end” was created so that there was less repetition in my code, it frees and closes dynamically allocated data to prevent memory leaks.

```
end (graph, curr path, shortest path, cities, num_cities, infile, outfile):
        delete graph
        delete both paths
        free all cities
        close infile and outfile
```

Main handles user input using getopt, mark the correct bool flag markers so that recursion happens for the path with the correct bool markers. Then the infile will be read, storing number of cities, city names, and edges, which are all verified to be valid. Recursion is called starting on vertex 0, and a shortest path will be either found or not found. If found, it will be printed out and success is returned.

IMPORTANT: At any point a success or error is returned, the function “end” will be called to delete any used memory

Main:

bools help, verbose, undirected, not_an_option

default infile and outfile is stdin, stdout

(use getopt switch for cases to set each bool and in/outfile)

if help or not_an_option:

print help stuff, return

scan infile for number of cities, and validate (return error if not valid)

create graph, curr path, shortest path, city name array (return error if not valid)

scan next number of cities times for city name and validate (return error if not valid)

scan next many lines until EOF for edges between cities and add each edge to graph

call DFS

//results:

if number of cities equals 0 or 1, “There’s no where to go” (return error)

if path of shortest path has no vertices, then no path was found

otherwise, print shortest path

print number of recursive calls