Assignment 4: The Perambulations of Denver Long

Kelly Liu

October 21, 2021

1 Description

In this assignment, we are making a program to find the most optimal path for Denver Long to return home. We will be using constructors, destructors, accessors and manipulator functions in our graph, path and stack interfaces to do so.

2 Pseudocode

2.1 graph.c

First, I need to include all files and define certain variables.

2.1.1 Graph

```
1 struct Graph {
2     uint32_t vertices;                    // Number of vertices.
3     bool undirected;                      // Undirected graph?
4     bool visited[VERTICES];               // Where have we gone?
5     uint32_t matrix[VERTICES][VERTICES]; // Adjacency matrix.
6 };
```

This implements our structure declaration.
Set up a vertices, undirected and visited[VERTICES], matrix[VERTICES][VERTICES] variables

2.1.2 graph_create

This is our constructor for our graph.
Initialize our graph by allocating memory
Initialize our vertices and undirected variables to our arguments passed in
Check with for loop to see if the matrix is set to 0
Set our visited array to false

return graph

2.1.3 graph_delete

Free all memory by freeing our graph and setting it to null

2.1.4 graph_vertices

Return vertices

2.1.5 graph_add_edge

Use an if statement to check if vertices i and j are within bounds of the matrix
Use an if statement to check if the graph is undirected
If so, add a weight k from vertex j to vertex i
Else, return false
return true

2.1.6 graph_edge_weight

Use an if statement to check if vertices i and j are within bounds of the matrix
If so, return edge weight at i, j
Else, return 0

2.1.7 graph_has_weight

Use an if statement to check if vertices i and j are within bounds of the matrix
If so, use an if statement to check if that edgeweight at i,j is bigger than 0
Return true if so
Exit both if statement
Return false

2.1.8 graph_visited

Use an if statement to check if visited[argument] is true
If so, return true
Else, return false

2.1.9 graph_mark_visited

Use an if statement to check if vertex v is in bounds
If so, set visited[v] to true

2.1.10 graph_mark_unvisited

Use an if statement to check if vertex v is in bounds
If so, set visited[v] to false

2.1.11 graph_print

Debug function where I can test if all my functions work properly in my graph.c file

2.2 path.c

First, I need to include all files and define certain variables.

2.2.1 Path

```
1 struct Path {
2     Stack *vertices; // The vertices comprising the path.
3     uint32_t length; // The total length of the path.
4 };
```

This implements our structure declaration.
Set up the vertices and length variables

2.2.2 path_create

This is our constructor for our path.
Set up our path by creating a stack with VERTICES number of vertices
Set length to 0

2.2.3 path_delete

Free up our path and set it to null

2.2.4 path_push_vertex

Use an if statement to check if our stack is full by using stack_full()
If so, return false
Push vertex v onto our path
Update length to itself plus the edge weight of the vertex pushed on
Return true

2.2.5 path_pop_vertex

Use an if statement to check if the stack is empty
If so, return false
Set the pointer v to the popped vertex in our stack using stack_pop()
Update the length to itself minus the edge weight of the vertex at the top of the stack and v Return true

2.2.6 path_vertices

Return length of our vertices variable

2.2.7 path_length

Return length

2.2.8 path_copy

Call on stack_copy the vertices
Copy the length variables as well

2.2.9 path_print

print stack_print() to outfile

2.3 stack.c

First, I need to include all files and define certain variables.

2.3.1 Stack

```c
struct Stack {
    uint32_t top;       // Index of the next empty slot.
    uint32_t capacity;  // Number of items that can be pushed.
    uint32_t *items;    // Array of items, each with type uint32_t.
};
```

This implements our structure declaration.
Set up the top,capacity and items variables

2.3.2 stack_create

Create our stack by allocating memory
Initialize top to 0
Initialize capacity to argument passed in
Initialize our items array by allocating memory
Use an if statement to check if our stack is not equal to our items array
If so, free up our stack and set it equal to null
Return stack

2.3.3 stack_create

```c
void stack_delete(Stack **s) {
    if (*s && (*s)->items) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}
```

Psuedocode is provided above on the assignment doc, but I'm not quite sure how this works and will figure it out later

2.3.4 stack_empty

Use an if statement to check if top equals 0
If so, return true
Else, return false

2.3.5 stack_full

Use an if statement to check if the top-1 is equal to the capacity
If so, return true
Else, return false

2.3.6 stack_size

Return top

2.3.7 stack_push

Run stack_full() to see if it's true
If it's true, return false
Else
Push argument x to the top of our stack
Update top to itself plus one
Return true

2.3.8 stack_pop

Run stack_empty() to see if it's true
If it's true, return false
Else
Update top to itself minus one
Pop an element of our stack into pointer x
Return true

2.3.9 stack_peep

Run stack_empty() to see if it's true
If it's true, return false
Else
set a temp variable
pop the pointer of the temp variable
set the pointer x variable to temp
push the temp variable back into our stack
return true

2.3.10 stack_copy

Free up dst items stack memory
allocate memory into dst items stack
use memcpy to copy dst src items stack into dst items stack
copy capacity and top variables as well

2.3.11 stack_print

Print out the stack to my outpile from the bottom Pseudocode is provided

```
void stack_print(Stack *s, FILE *outfile, char *cities[]) {
    for (uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, "%s", cities[s->items[i]]);
        if (i + 1 != s->top) {
            fprintf(outfile, " -> ");
        }
    }
    fprintf(outfile, "\n");
}
```

2.4 tsp.c

2.4.1 main

I will be using gept opt to check which cases are ran
Using a readbuffer, I will be reading the amount of cities I have, then iterating through the
cities names to save them to an array
Then I will be reading the given edges and adding it to the graph by splitting the string by spaces
Then I will create my two paths, and then recursively call on a search to find the most optimal path
Have a bunch of error handling for specific test cases
Freeing up the memory after I am done

2.4.2 isNumber

checks if the argument passed in is a number by iterating through each character

2.4.3 dfs

increment my recursive count
mark v as visited in my graph
create a temp variable
push my vertex onto my path
use an if statement to check if the current path is longer or equal to the path
If so, pop the vertex and mark ait as unvisited and then exit
use an if statement to check if the current path has every vertex and if it connects to the beginning (outter if statement)
check if it's verbose (if statement 1)
If so, print out the path. Exit if statement 1
Use if statement to check if the path is the first path found or if it's shorter than the shortest path (if statement 2)

If so, copy the current path to shortest path. Exit if statement 2
pop the start vertex, exit outter if statement
Use a for loop to loop through every vertex
use an if statement to check if all vertices that the current vertices connects to and hasnt been visited
If so, recursively call on DFS for that vertex. Exit if statement
Exit for loop
pop the vertex
mark the vertex as unvisited

3 Files

graph.c- A source file for implementing the graph of ADT.
graph.h- A header file that specifies the interface for graph.c.
path.c- A source file for implementing the path ADT.
path.h- A header file that specifies the interface for path.c.
stack.c- A source file for implementing my stack ADT.
stack.h- A header file that specifies the interface for stack.c.
tsp.c- A source file that has my main function and every other function needed for my program.
vertices.h- A header file that contains macros regarding vertices.
Makefile- This allows us to use clang and compile our program.
README.md- In markdown format, it tells us how to run the program and how the program was made.
DESIGN.pdf- This is how I started thinking about how to code the program.

4 Error Handling

1. For my stack pop function, I realize that I needed to decrement my top before saving it to the x pointer.
2. I had forgot to write an function needed in my graph.c function.
3. My Makefile had alot of issues with it that was easily debugged when ran.
4. My DFS function didn't work for a long time, it took alot of trying and retrying and then I realized that I was using the wrong logic. I had to completely scrap it and try again. It worked after hours of looking and trying different implementations.
5. I had a segmenetation error because I was doing top-1 in my stack function in certain areas that I shouldn't have been.
6. A lot of syntax errors, albeit alot of it was because my a key on my keyboard doesn't work some of the time. The syntax errors were easily fixed as well when I ran it to see what was wrong.
7. There were some test cases that I didn't account for when I was writing my

DESIGN PDF, I had to add all those into my code.
8. I had commented out a part of my stack.c file when I was debugging, and I forgot that I had it commented out when I was running it through the pipeline.

5 Credit

1. Professor Long has provided pseudocode in the Assignment 4 description PDF for our graph.c, path.c, and stack.c file as well as describing specific things that we should be doing.
2. Professor Long has provided us with a few files in the resources folder in Assignment 4.
3. I used https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm to figured out how to copy my stacks.