Assignment 4

The Perambulations of Denver Long

Background

In this assignment, we are basically given a list of cities, (with given distances between each one), and we need to figure out what is the most *optimal* route for Denver Long to travel. The most optimal route can be found by finding the shortest Hamiltonian path. We need to use a Hamiltonian path due to the fact that we need to visit ***all*** the cities at least once.

Graphs

So how will we represent this problem? For this assignment, we will be representing our situation with a graph. More specifically, we will be representing this as an adjacency matrix. With that being said, we will have two different kinds of graphs: a directed graph and an *undirected* graph. A *directed* graph is used to represent one-way roads that Denver can travel on. An *undirected* graph represents a two way road that Denver can travel on.

In our matrix, we will be using the form <i, j, k> for our coordinate-like system. This is called an edge and it is filled with vertices and weights. In our problem, a vertice $\{i, j\}$ is the list of cities Denver will be passing by while our weight $\{k\}$ would be the distance between two different vertices.

Let's look at the assignment pdf for an example:

	0	1	2	3	4	5	⋯	25
0	0	10	0	0	0	0	0	0
1	0	0	2	5	0	0	0	0
2	0	0	0	0	0	3	0	5
3	0	0	0	0	21	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0

Here, we can see that the distance from vertex 0 to vertex 1 has a distance of 10.

Functions and Pseudocode for the Graphs struct

- Graph *graph_create(uint32_t vertices, bool undirected)
    - This is a constructor function that creates an “instance” of the Graph struct. The undirected boolean tells us if the Graph is supposed to be inverted or not. We do not need any pseudocode for this since Prof. Long has already given us the C code for making the function
- Void graph_delete(Graph **G)
    - This is a destructor for the Graph struct. We would need this due to the fact that we are calling the calloc() function in graph_create(). Everytime we call calloc(), we would need to free up memory or else we would get a memory leak. In addition, we would need to set our pointer to NULL that way it’s contents would be completely wiped.
    - We do not need any pseudo code for this function since Prof. Long has already given us the C code for this function.
- uint32_t graph_vertices(Graph *G)
    - This returns the vertices variable of the Graph struct.
    - No pseudocode is needed here as well since this is just a simple return statement
- graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)
    - This is a function that manipulates the matrix variable of the Graph struct.
    - In short, we are basically saying => matrix[i][j] = k. However, this is not the full extent of the function since we need to check other conditions as well. Below is some pseudocode of how I would implement the function:
    -

```
def graph_add_edge(G, i, j, k):
    if i or j are not within vertices bounds:
        return False

    if undirected == True:
        matrix[i][j] = k
        matrix[j][i] = k
    else:
        matrix[i][j]=k
    return True
```

    - NOTE: the pseudocode is somewhat in python language
- graph_has_edge(Graph *G, uint32_t i, uint32_t j)
    - This is a boolean function that returns true if the vertices <i,j> have an edge or weight that is a positive non-zero.
    - The pseudocode I made can be found below:
    -
```
def graph_has_edge(Graph, i, j):
    if i or j are not in bounds:
        return False

    if matrix[i][j] > 0:
        return True
    return False
```
- graph_edge_weight(Graph *G, uint32_t i, uint32_t j)
    - This function is exactly the same as the previous function however, it returns the actual value of the weight and returns 0 if the edge doesn't exist.
    - The pseudocode for this function is not necessarily needed since it is the same as graph_has_edge however, instead of returning True or False, I can just return matrix[i][j]
- graph_visited(Graph *G, uint32_t v)
    - This returns true of vertex v has been visited and false otherwise
    - Since the visited array in the Graph struct is already an array of booleans, we don't need any if else statements. Rather we would just return the array with the specified index which is v
- void graph_mark_visited(Graph *G, uint32_t v)
    - This just manipulated a certain element in the visited array as true
    - No pseudocode is needed since I will just call the visited array with index v and change it to true
- void graph_mark_unvisited(Graph *G, uint32_t v)
    - This is the same as mark_visited however, I will change the value to false

Depth-first Search

In order to find the shortest hamiltonian path, we would need to use the **Depth-first Search** (DSP) method. The DSP method is a recursive algorithm that finds all combinations of paths that would pass through all the vertices at least once.

The solution to our problem would then be the shortest Hamiltonian path that we found.

Below is the pseudocode front he assignment pdf that tells us how to use the DFS:

```
1 procedure DFS(G,v):
2     label v as visited
3     for all edges from v to w in G.adjacentEdges(v) do
4         if vertex w is not labeled as visited then
5             recursively call DFS(G,w)
6     label v as unvisited
```

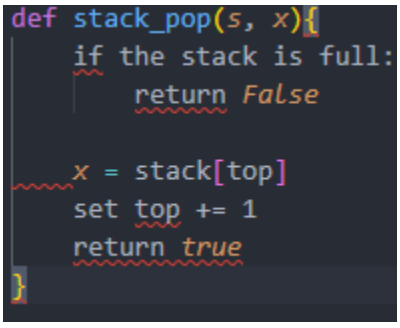
Stacks

A stack uses the first item in, last item out idea. Think of it as a stack of pancakes. You can only add pancakes on top of the plate and you can only eat pancakes at the top of the stack. It is not possible to eat the pancakes at the bottom of the plate unless you have eaten all the pancakes above it.

In this assignment, we will be using stacks in order to keep track of all the hamiltonian paths we have made. So let's get into the specifics of this abstract data type.

Functions and pseudocode for Stacks

- Stack *stack_create(uint32_t capacity)
    - This is a constructor function that creates an "instance" of the Stack struct. In the function, all variables will be initialized and the items array will be dynamically initialized with calloc().
    - We do not need pseudo code for this since we are already given the constructor code in C from Prof. Long
- stack_delete(Stack **s)
    - This function deletes the instance of Stack since we are calling both malloc and calloc. Without this function, we would be having a lot of memory leaks.

    - We do not need any pseudocode for this due to the fact that we are already given working C code that can deal with this already.
- stack_empty(Stack *s)
    - This returns true if the stack is empty or not false otherwise.
    - We don't really need pseudocode for this due to the fact that you can just check if the top is 0. If it is then return true else false.
- stack_full(Stack *s)
    - This returns true if the stack is full else false.
    - We also don't need to write pseudo code for this due to the fact that we can just simply check if the top is the same as the capacity. If it is then you would return true.
- stack_size(Stack *s)
    - This just returns the number of items on the stack.
    - No pseudocode is needed for this as well since we can just return what the current value of top is.
- stack_push(Stack *s, uint32_t x)
    - This function pushes the item on top of the stack.
    - Implementing this function is simple due to the fact that we are just adding the passed in value to the top index of the stack. The following pseudocode shows us how I would go about implementing the function:
    - 

    - NOTE: this is somewhat written in python
    - ANOTHER NOTE: I made a typo and the function is supposed to say stack_push
- stack_pop(Stack *s, uint32_t *x)

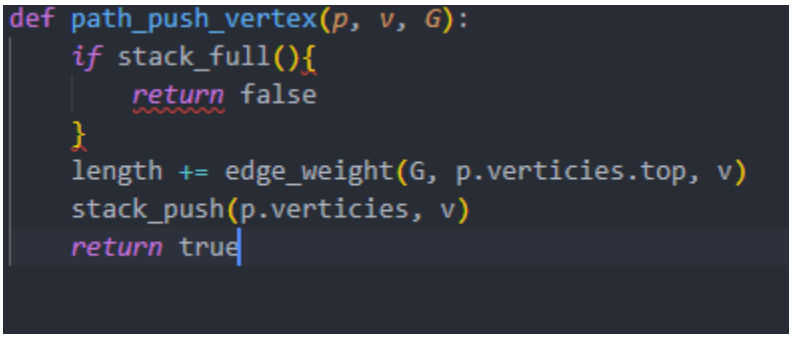
- This function pops the element from the top of the stack. This would point to a value x and “returns” what element that was popped and decrement the top count
- Implementing this is simple as well since we are just decrementing x
-
```
def stack_pop(s, x){
    if the stack is empty:
        return False

    pointer x = stack[top]
    set top -= 1
    return true
}
```
- This is was also written somewhat in python form

- stack_peek(Stack *s, uint32_t *x)
  - This function is exactly like stack_pop() except it doesn’t modify the top variable.
  - We don't need any pseudocode for this due to the fact that we can just use the pseudocode for stack_pop() except we would just need to remove the top decrement line.
- stack_copy(Stack *dst, Stack *src)
  - This function copies one Stack “instance” to another Stack “instance. This is like a copy and paste function.
  - In order to implement this we would need to copy the items array of *src to the items of *dst. In addition, we would need to copy the value of *src->top to *dst->top.
  - Below is a python-like pseudocode as to how I would implement the copy function:
  -
```
def stack_copy(stack1, stack3):
    stack1.top = stack2.top
    for x in stack1.items:
        index i = 0
        stack2.item[i] = x
        i += 1
```

Paths

In order to keep track of the paths that we have taken in our program, we should use a struct that handles all the stack related operations.

- *path_create(void)
    - This is a constructor function that initializes the vertices stack and the length of the path.
    - Implementing this is pretty simple. It is identical to initializing the Graph struct where we call malloc and initialize the stack. Then we would initialize the fields of Path. This is pretty simple to implement and we would not need any pseudocode for this.
- path_delete(Path **p)
    - This is the destructor function where we would need to free() the Path struct we initialized.
    - All we need to do here is free the Path and set that Path to NULL.
- path_push_vertex(Path *p, uint32_t v, Graph *G)
    - This pushes the current vertex v onto the stack. In addition, you would increase the length of path based on what's on top of the stack and what the current vertex is. You then push the vertex onto the stack.
    - Below is some small pseudocode that shows us how it can be done.
    - 

    - NOTE: in this case, the stack is full when it equals the number of vertices in G
- path_pop_vertex(Path *p, uint32_t *v, Graph *G)

    - This does the complete opposite of path_push_vertex(). Instead of increasing the length, you would decrease the length depending on what's on being vertex is being popped and the top of the vertex
    - If the stack is empty then you would return false.
- path_vertices(Path *p)
    - This is similar to stack_size() where you would need to return the number of elements in the stack. In this case you could just return whatever the value of stack_size() is.
- path_length(Path *p)
    - This just returns the length field of the Path struct
- path_copy(Path *dst, Path *src)
    - This just utilizes the stack_copy() and copies the stack in *src to *dst.
    - Also this copies length field of *src to *dst

TSP and main()

Let's go through how we will run this this program now:

- First, we will have to handle a variety of command line options:
    - All command line options are specified in the assignment pdf
    - We don't need to know to how to handle command-line options since we already have covered this in the past few assignments
- Then we will parse a given graph file
    - The first line of file tells us that there are *n* vertices
    - The the next *n* line tells us the names of the cities or places as a string
    - After *n* lines, we have the coordinates and weights of our graph. We would need to initialize our Graph G and add the edges of the coordinates
        - The coordinates are listed all the way until we hit the end of the file
    - For each coordinate, we should utilize the add_edge function.
    - Then we would need to create two Path instances. One Path is used for keeping track of the current path while the other one is used to keep track of the shortest path

    - Now, we will call our dfs function.
        - We need to start at the origin vertex which is given by the macro START_VERTEX in vertices.h.
        - While the function is running remember to keep updating the 2 Path variables
    - Once we found the path, we need to print out a message that displays our results
    - If the verbose option is enabled, we would then need to print all Hamiltonian paths.

Credits

- The graph and the DSF algorithm pseudocode was all taken from the assignment pdf
- Euegene's 10/19/21 section helped inspire me with approaching this assignment