DESIGN: Assignment 4 – The Perambulations of Denver Long

Nelly Sin

October 2021

1 Functionality

In this assignment, we will be using matrixes and fake ADT (Abstract Data Type). We will be creating paths in Hamiltonian and by using each command line.
-h (printing the help menu)
-v (prints all Hamiltonian paths and the total recursive calls)
-u (making the graph to be undirected)
-i (inputting file path containing the cities and edges)
-o (creating a new file and input the results)

Graphs and stacks will have their own ADT or their own interface that comes with a constructor, destructor, accessor, and manipulator functions.

2 Graphs

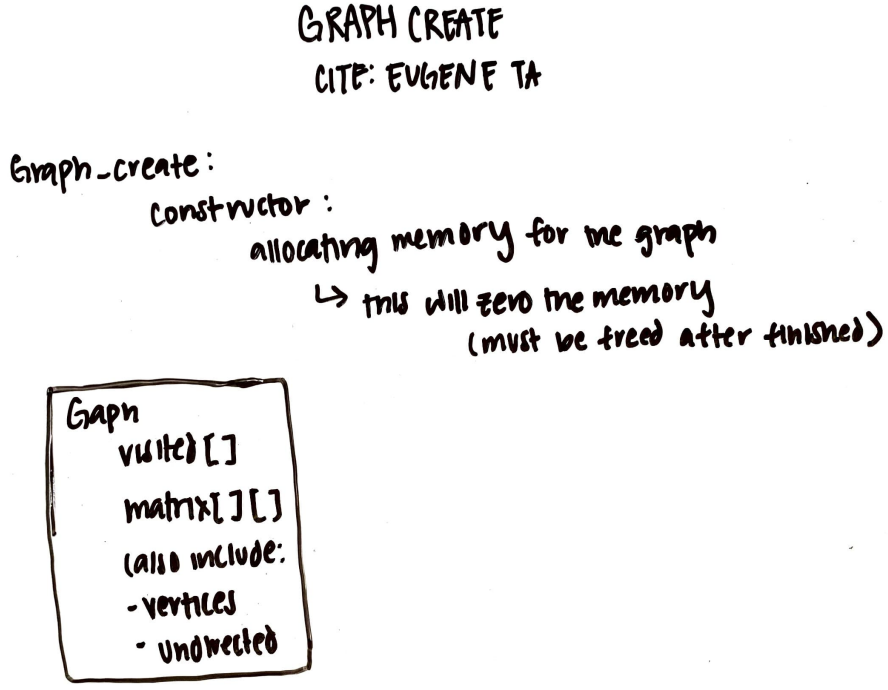
Graphs can be indirected and directed. By having a matrix that points both ways.

G = <V, E>
V = vertices V0, V1, V2, ..., Vn
E = E <V0,V1>, <V0, V2>, ...
(Edges will be a tuple of vertices)
Such as: $< 0, 1, 10 >, < 1, 2, 2 >, < 1, 3, 5 >, < 2, 5, 3 >$

ADT: abstract data type interface.
This checks if the vertices have been visited and setting the adjacent matrix.

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

A visual representation of undirected graphs; differences between undirected and directed graphs



This was my thought process of creating the graph. After watching Eugene's lecture

2.1 Graph create

The ADT is the first to be implemented because it will have the number of vertices, undirected graph, if we have visited the vertices, and the matrix. This constructor is a function that returns the graph pointer and two in two inputs:

1. whether or not it's undirected.
2. how many vertices to take in.
3. the visited array
4. the matrix array

Graph create creates a graph with vertices. This will have an array of booleans, and set the values to false. This function is only for initializing. By using dynamic memory allocation to set matrix to 0.

The vertices capacity will be defined as 26 and the startvertex is 0.

2.2 Graph delete

Because we used dynamic memory allocation, we must free it once we're done. When doing this, we can prevent memory leaks.

2.3 Graph Vertices

This function keep tracks of the vertices. So this function will return the graph vertices.

2.4 Adding graph edge

Adding "k", a weight, from vertex i to vertex j. Return a boolean true if the vertices are within the bound and edges but return false if they have not been successfully added.

2.5 Graph edge weight

Returns the weight of the edge from vertex i to vertex j. The edge will be k.

2.6 Graph visited

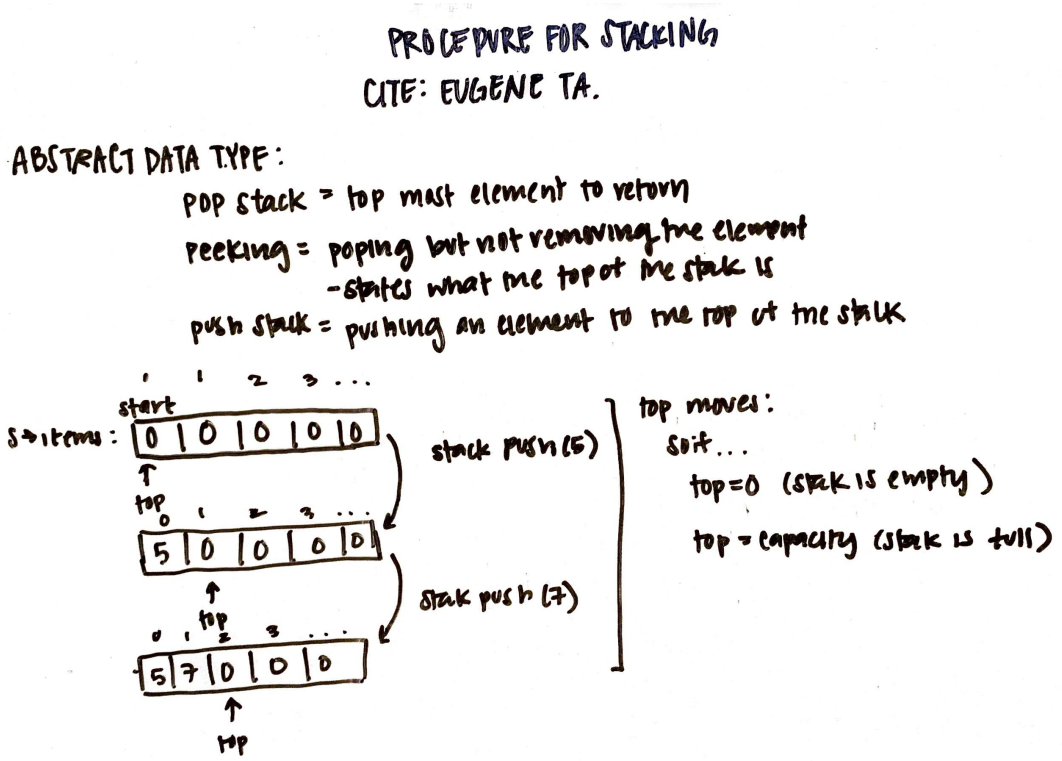
Returns true if the vertex has been visited.

2.7 Graph marking

If vertex is within the edge, and mark the vertex as true for visited, else mark the vertex as false.

3 Stack

Stack will be formatted like a stack of pancakes. Last in, first to return.



After watching Eugene's section video, this is what I have thought of the stack in this assignment.

3.1 Peeking:

What is on top of the stack without modifications. (This will be similar to stack pop, but with no modifications to top)

If stack is empty: we can peek return true else: we cannot and return false

3.2 Pushing:

Pushing is pushing to the stack, this is will putting the item where the top is. This is be increment to the stack and increment the top.

If stack is empty: we can pop return true else: return false

3.3 Popping:

Pop is the inverse operation of pushing. Decremented top instead of increment.

If stack is full: we can push and return true else: return false

3.4 Stack empty:

When there is no items in the stack. The top will equal to 0, thus returning the stack is empty is true.

3.5 Stack full:

When the there are maximum capacity on the stack pile. The top will equal the capacity and the stack full will return true.

3.6 Stack copy:

Take source stack and apply to destination stack. In the end, both stacks will be identical. This will be used in the path abstract data type (ADT).

3.7 Stack print:

Printing all items in the stack.

4 Path

Path contains the stack. So we will have to call stack.

4.1 Path create

Creates a path (using stack create) and initialize path vertices and length.

4.2 path delete

Deletes a path after using it. So there is no memory leak.

4.3 Path vertex

Path will always start in the origin (0), and add the weight of the edge. The length of the path is decreased by the edge weight connecting the vertex at the top of the stack and popped vertex.

Peek each time you move in the stack.

We can see if it's a Hamiltonian path if and only if it goes back to the origin. The size of the stack is equal to number of vertices in the graph AND up back to as the origin.

All vertices are unique, and therefore, if you have gone to all of them, you would know when you have visited everything and to the origin.

– If the edge is connecting to the origin then it's a Hamiltonian path.

I will have to use the function with graphhasedge, stackpeek, stackpop, and stackpush.

4.4 Path vertices

Returns the number of vertices in the path (stack size).

4.5 Path length

Returns length in the path. So how is length increments and decrements in the path vertexes.

4.6 Path copy

Necessary for when I'm doing the DFS. Save it somewhere to find the shortest path.

It takes the path destination that is initialized and makes a copy of the source path. Requiring to make a copy of the vertices stack and the length of the source path.

dest = src

4.7 Path print

prints path to "outfile". A separate file for the result.

5 Command Lines

5.1 -v

printing out all the Hamiltonian paths it created and found. Including the number of recursive calls (DFS())

*path function calls the stack function

5.2 -u

This can be a boolean since it sets Graph to be undirected.

5.3 -h

Print help menu with all the command lines and a summary of each.

5.4 -i

Input the path containing the cities edges. This will have a default.

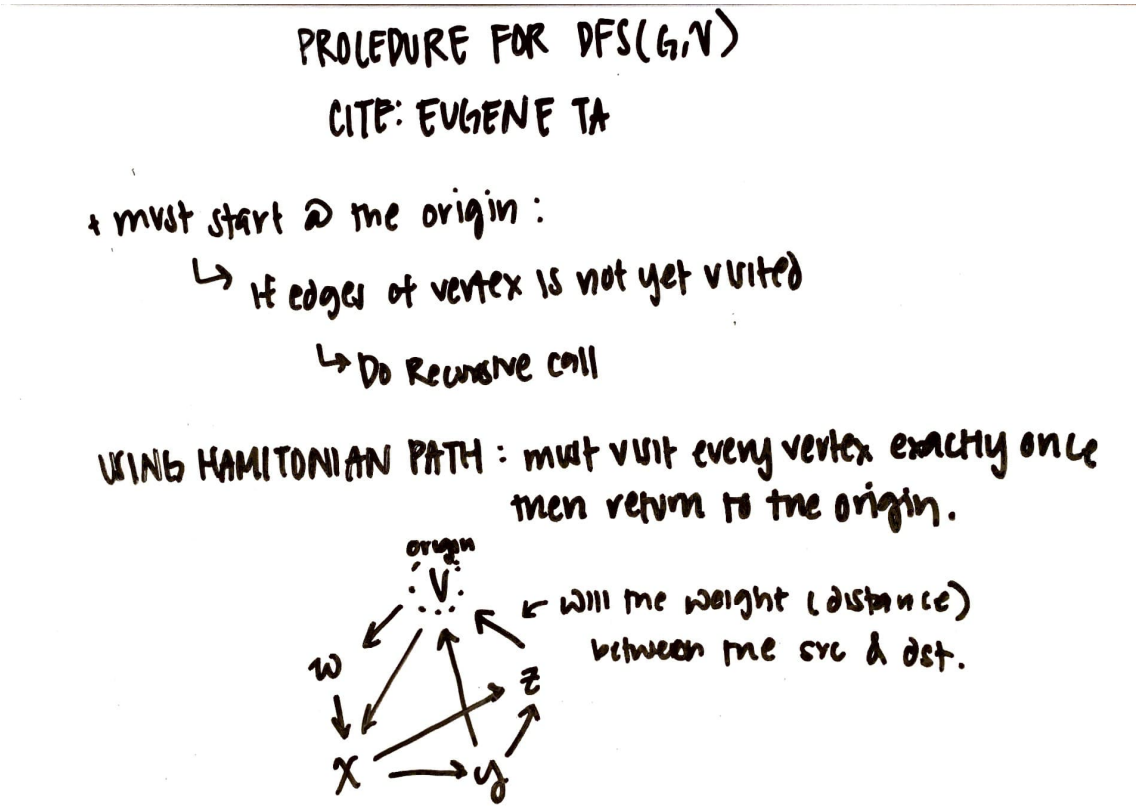
5.5 -o

Copying result to a new file. By printing out the result and storing it to an outfile.

6 Putting it all together

Similar to the previous assignment, we will be linking the files together.

In addition, we will be using DFS to find paths and that passes through the vertices. (Again, using Hamiltonian path)



After watching Eugene's section video, this is what I have thought of implementing the DFS in this assignment.

```
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
```

Provided by Professor Long as a template of a base case and the recursive call for the DFS

Depth for search functionality: Base case: Every vertex has been visited and is reachable (This means it is a Hamiltonian path) Recursive case (else): for every vertex accessible from the current vertex.

However, we will always have to end up from the beginning. And when the path is almost done,if the first city is reachable, then it completes the Hamiltonian path. So, if it is not reachable we must backtrack to find a different path to take.

Depth for search in sudo:
v ¸ takes in the vertices
Initialize visited to false
for(i=0;i¡v;i++) ¸ loop for checking
visited[i] = false;

while the stack is not empty:
stackpop(stack, v)
visited = graphvisited(visited stack, v);

for variable u in graphvertices, increment u:
if has visited and there is a graph edge
that is reachable:
stackpush on to the visited stack
mark to be visited
else if it is not reachable then:
stackpop on the visited
mark to be not visited
do a recursive call of DFS

Implement the command lines as well for -h, -i, -u, -v, and -o.

We will also have to include the the functions we implemented in our fake graph ADT. Such as creating the graph and adding the edges to the graph. This must be done when reading the file of the graph.

Create a current path and shortest path using the path create from the path ADT.

Call DFS, depth for search, to start the recursion.

Lastly, we should be printing the results using the path print and stack print for the paths.

6.1 Main

CITE: James Tennant
Parse command lines first with -u -v -h -i (infile) -o (outfile)

This part is reading the infile
One section will be getting the number of vertices and print an error if the vertices is larger than 26.

Next is calling the creategraph by using the undirected boolean and the vertices

After, we can create an array and store the next couple of lines for the city names. We can tell how many city names to read by using the vertices we have just received.

Continuing along reading the file, we will need to add i,j, and k to the graphaddedge and printing out each.

After reading all lines:
Create two paths by calling pathcreate, this will be current and shortest.

Calling DFS will be the last and printing out results by an outfile.