

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.

$$G = \langle V, E \rangle$$

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

GRAPH CREATE

CITE: EUGENE TA

Graph-create:

constructor :

allocating memory for the graph

↳ this will zero the memory
(must be freed after finished)

Graph

visited[]

matrix[][]

(also include:

- vertices
- undirected

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.

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.

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.

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.

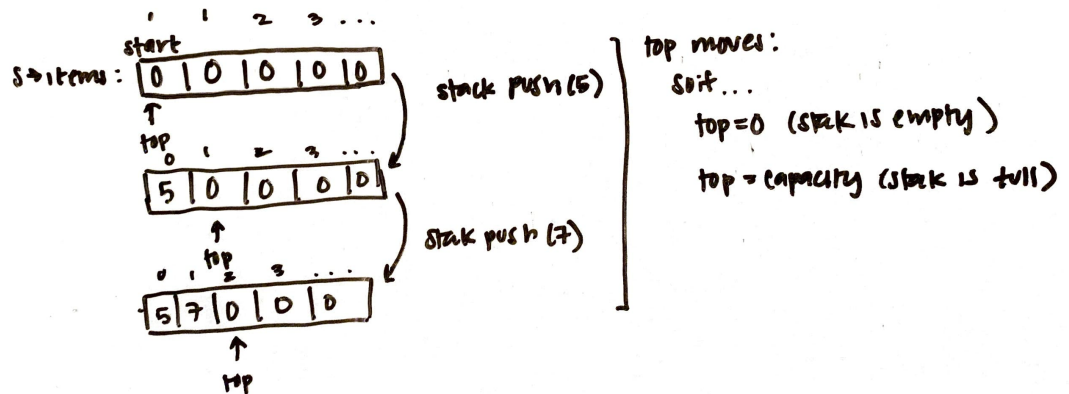
PROCEDURE FOR STACKING
CITE: EUGENE TA.

ABSTRACT DATA TYPE:

POP stack = top most element to return

PEEKING = popping but not removing the element
- states what the top of the stack is

PUSH stack = pushing an element to the top of the stack



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.

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.

3.3 Popping:

Pop is the inverse operation of pushing. Decrement instead of increment.

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.

4.1 path create

creates a path

4.2 path delete

deletes a path

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.

4.4 Path vertices

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

4.5 Path length

returns length in the path. so what's in path copy.

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.

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 Graph to be undirected

5.3 -h

Print help menu

5.4 -i

Input the path containing the cities edges. Default this to stdin.

5.5 -o

Copying result to a new file.

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)

PROCEDURE FOR DFS(G, V)

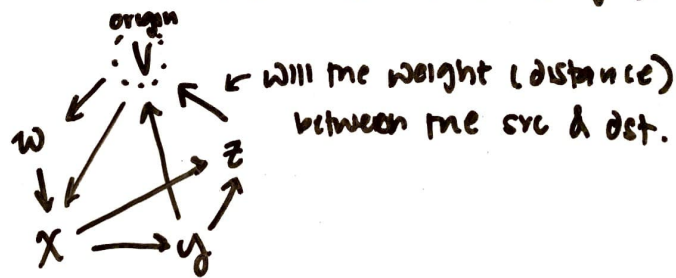
CITE: EUGENE TA

* must start @ the origin :

↳ If edges of vertex is not yet visited

↳ Do Recursive call

USING HAMILTONIAN PATH : must visit every vertex exactly once then return to the origin.



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