CS5008 HW 08: Graphs

The purpose of this assignment is to give you experience with implementing a graph data structure and implementing graph functions. This assignment has a total of 100 possible points.

Objectives

Upon completion of the assignment exercise, you will be able to do the following:

- use the adjacency matrix representation of a graph
- calculate degrees for vertices
- determine neighbors of vertices
- implement depth-first search
- determine the path from node x to y using the results of depth-first- search

Pull the Homework Starter code from Github. Try to compile them:

```
gcc -Wall graph.c  graph.h graph_main.c
./a.out
```

Part 1: Understand the existing code

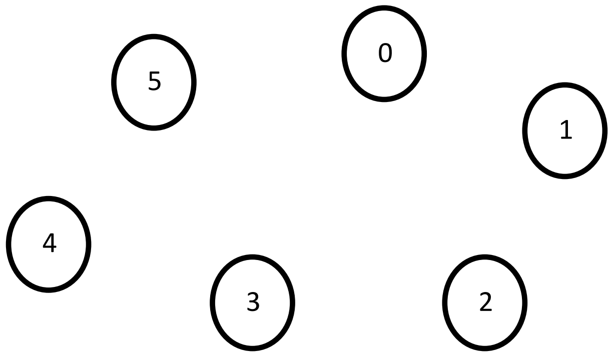
Open `graph.c`. In this assignment, the graph is an **unweighted**, **directed** graph that is implemented using the adjacency matrix representation. The nodes are labeled 0 to |V|-1 where V is the set of vertices in the graph. For example, if the graph has 4 nodes, then the node labels are {0, 1, 2, 3}.

1. What are the members of the Graph struct? ___________________________________

2. What are the members of the DFS struct? _____________________________________

3. Look at the `main` function. How many vertices does `g` have after it is created? _________

4. You'll see calls to `addEdge`. Some of these calls will not add an edge (if the src is out of range, if the dest is out of range, if an edge already exists). Draw the edges in the graph g below:



5. Scroll down to the function `createEmptyGraph`. Read through this. This function creates a graph with no edges of numVertices nodes. `freeGraph` frees all memory associated with a graph.

Part 2: Implementing functions (30 points0

As you work through the function definitions, you may want to compile often:

1. Go to the `addEdge` function definition. Complete this function so that it does the following:
   - if graph is null, prints an error message about not adding an edge and return 0
   - if src or dest is out of range for the graph (< 0 or >= g->V), print "src or dest vertex invalid" and return 0
   - if graph already has an edge from src to dest, print "edge already exists between 0 and 3" if the src is 0 and dest is 3; return 0
   - set g->M[src][dest] to 1 and return 1

After you get addEdge working, test it on the graph in main. Which of the `okX` variables are set to 0?

____

2. Go to the `outDegree` function definition. Complete this function so that it does the following:
   - if graph is null, returns 0
   - if v is out of range, returns 0
   - count the number of edges from v to other nodes and return this count (think about what row or column you need to look at in G->M)

3. Go to the `inDegree` function definition. Complete this function so that it does the following:
   - if graph is null, returns 0
   - if v is out of range, returns 0
   - count the number of edges from other nodes to v and return this count (think about what row or column you need to look at in G->M)

4. Go to the `isNeighbor` function definition. Complete this function so that it does the following:
   - if g is NULL, return 0
   - if x or y are out of range, return 0
   - if has an edge from x to y, return 1; else, return 0

Part 3: Implementing DFS and path finder (25 points)

1. Read through the `dfsInit` function. This initializes the array of DFS objects prior to calling `dfs`. Complete the `dfs` function so that it recursively performs depth-first search. It should print "visited 0" when it visits node 0. Note that the `time` variable is global, so the recursive function calls have access to the same `time` variable. Use `isNeighbor` to determine the vertices adjacent to the node you are currently processing.

2. Complete the `printReversePath` function so that it prints the path from dest to source. Note that a more user-friendly version would reverse the path to print from src to dest, which could easily be done with a stack. To implement this, you start with current node as the dest node. You find its parent and set this to the current node. Print the current node. Keep doing this while the current node is not -1 and current node is not the src.

An example printout for a path from 0 to 1 would look like:

```
PATH: 1 <-3 <-0 <-
```

Part 4: Choose your own graph function (20 points)

1. Choose something you want to implement for the graph. Here are some options:
    a. breadth-first search (will also need a queue; can use queue code from earlier lab)
    b. distance-away from source to dest (# hops, based on breadth-first search), for example if node 5 is 3 hops away from 0, then it should return 3. If there is no path from the source to the dest, return -1.
    c. print a topological sort of the graph (based on depth-first search)
    d. re-do the graph representation as an adjacency list; update addEdge, isNeighbor, printGraph, inDegree, outDegree
    e. minimum spanning tree (Prim); note: graph needs to be undirected, so update addEdge to add edges between (src, dest) and (dest, src)
    f. minimum spanning tree (Kruskal); note: graph needs to be undirected, so update addEdge to add edges between (src, dest) and (dest, src)
    g. is-cyclic, check to see if there is a cycle from any node back to itself
    h. is-connected, check to see if there is a path from every src node to all other nodes
    i. is-subgraph, create new graph and check it against another graph. A graph G' is a subgraph of G if V' is a subset of V and E' is a subset of E.
    j. something of your own desire (ask Tammy first if it is not on the above list); note that you are implementing dijstra's for homework, so **do not do dijkstra** for the lab

2. What feature did you add to the code? ____

Part 5: Finish up (25 points)

Push your files to your Github code repository but first ensure that the code runs in the Khoury Server.

For this Homework, note that you will need to present the work to TA and Professor for code walk through. This component and uploading to Github is 25 points.