Assignment10

A10: Graphs

Due Date: Tuesday, July 31th 2018 at 11:55 pm.

Assignment Files:

1- Create an eclipse C Project with any relevant title like: "cp264_A10_*studentID*".
2- Download the file: "`cp264_A10_files.zip`"
3- Extract the following files and import them into the project. The files are:
   1. "`graph_components.h`"
   2. "`graph_components.c`"
   3. "`graphs_AM.h`"
   4. "`graphs_AM.c`"
   5. "`main.c`" `//main`

You only need to change two files: *"`graphs_AM.c`"* and *"`main.c`"*.

Submission Instructions:

Submit two files: *"`graphs_AM.c`"* and *"`main.c`"*. Make sure you edit/add the following comment on top of the two files:

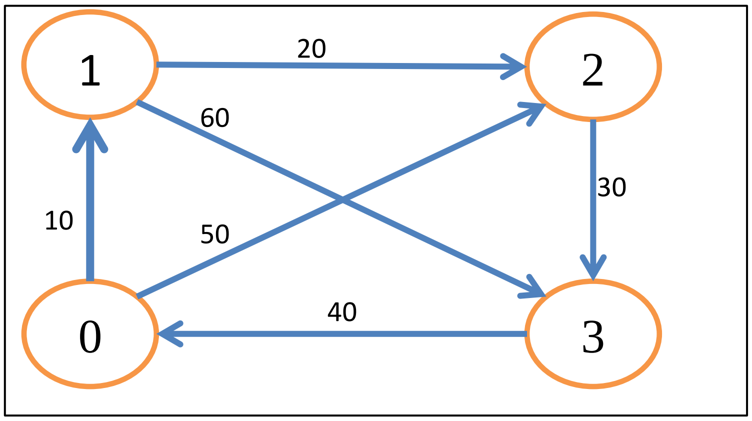
```
/*
-------------------------------------------------------
Author:  Your Name
ID:      Your ID
Course:  CP264 Spring 2018
Category: Assignment 10
-------------------------------------------------------
*/
```

Task 1: Create Graphs

In *"`main.c`"* file, create the following function:

```
GraphAM* create_graph1(int directed, int weighted);
```

The function creates the graph depicted in the following diagram and returns a pointer.



If the input parameter *directed* is set to 0, it means the graph should be undirected (no arrows – just edges). If it is set to 1, then the graph should display the arrows.

If the input parameter *weighted* is set to 0, then all edges are assumed to have a weight of 0 (i.e. it is non-weighted graph). If it is set to 1, then the graph should display the weights displayed above.

Create an array of Vertex pointers and array of Edge pointers and use loops whenever possible.

Running the *test_create_graph1( )* would yield:

```
undirected and non-weighted graph:
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1    0    0   -1
   -1   -1    0    0
   -1   -1   -1    0
    0   -1   -1   -1
directed and non-weighted graph:
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1    0    0   -1
   -1   -1    0    0
   -1   -1   -1    0
    0   -1   -1   -1
undirected and weighted graph:
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1   10   50   -1
   -1   -1   20   60
   -1   -1   -1   30
   40   -1   -1   -1
```

```
directed and weighted graph:
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1   10   50   -1
   -1   -1   20   60
   -1   -1   -1   30
   40   -1   -1   -1
```

Note that in the above implementation of adjacency matrix, we would need to rely on previous knowledge to recognize if the graph is directed or undirected.

Task 2: Remove Edge

In *"graphs_AM.c"* file, create the following function:

```
int removeEdge_graph_AM(GraphAM* g, Edge* e);
```

The function removes an edge from the given graph.

If the edge does not exist (i.e. the vertex numbers does not exist), then the function should print an error message and return 0. Similarly, if the graph is empty, an error message should be printed and the function returns 0.

Note that according to the given implementation of the adjacency matrix, that if there is an edge between v1---v2, then the edge stored at the following locations depending on the value of direction.

Direction value	Location of edge
direction = 0 (undirected)	matrix[v1][v2]
direction = 1 (directed v1→v2)	matrix[v1][v2]
direction = -1 (directed v1←v2)	matrix[v2][v1]
direction = 2 (directed v1→v2 , v2←v1)	matrix[v1][v2] & matrix[v2][v1]

Running *test_removeEdge()* should give the following output:

```
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1   10   50   -1
   -1   -1   20   60
   -1   -1   -1   30
   40   -1   -1   -1

Removing Edge: [(0)-10->(1)]
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 5
Adjacency Matrix:
   -1   -1   50   -1
   -1   -1   20   60
   -1   -1   -1   30
   40   -1   -1   -1

Removing Edge: [(1)-20->(2)]
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 4
Adjacency Matrix:
   -1   -1   50   -1
   -1   -1   -1   60
   -1   -1   -1   30
   40   -1   -1   -1

Removing Edge: [(2)-30->(3)]
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 3
Adjacency Matrix:
   -1   -1   50   -1
   -1   -1   -1   60
   -1   -1   -1   -1
   40   -1   -1   -1

Removing Edge: [(4)-70->(1)]
Error (removeEdge_graph_AM): Edge does not exist
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 3
Adjacency Matrix:
   -1   -1   50   -1
   -1   -1   -1   60
   -1   -1   -1   -1
   40   -1   -1   -1

Removing Edge: [(3)-40->(0)]
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 2
Adjacency Matrix:
   -1   -1   50   -1
   -1   -1   -1   60
   -1   -1   -1   -1
   -1   -1   -1   -1

Removing Edge: [(0)-50->(2)]
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 1
Adjacency Matrix:
   -1   -1   -1   -1
   -1   -1   -1   60
   -1   -1   -1   -1
   -1   -1   -1   -1

Removing Edge: [(1)-60->(3)]
```

```
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 0
Adjacency Matrix:
   -1   -1   -1   -1
   -1   -1   -1   -1
   -1   -1   -1   -1
   -1   -1   -1   -1

Removing Edge: [(1)-60->(3)]
Error (removeEdge_graph_AM): Can not remove from an empty graph
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 0
Adjacency Matrix:
   -1   -1   -1   -1
   -1   -1   -1   -1
   -1   -1   -1   -1
   -1   -1   -1   -1
```

Task 3: Remove Vertex

In *"graphs_AM.c"* file, create the following function:

```
int removeVertex_graph_AM(GraphAM* g, Vertex* e);
```

Below is the strategy to remove a vertex from a graph:

1- If the vertex does not exist (check the vertex *num* value over *vertexCount*) , print error message and return 0
2- Identify the row and column locations referring to the vertex.
3- Copy the last row and last column to the arrows
4- Change the value of *num* for the recently moved vertex to be the same as the vertex to be removed.
5- If there are edges associated with that vertex, then they also have to be removed, and the *edgeCount* value should be adjusted.
6- Use *realloc* to re-adjust the size of the adjacency matrix.
7- Remove vertex from vertex list, decrement the *vertexCount*, replace the removed one by the last vertex in the list after changing its *num* value.

Running *test_removeVertex()* should give the following output:

```
(Adjacency Matrix Representation): #Vertices = 4, #Edges = 6
Adjacency Matrix:
   -1   10   50   -1
   -1   -1   20   60
   -1   -1   -1   30
   40   -1   -1   -1
```

```
Removing Vertex 1:
(Adjacency Matrix Representation): #Vertices = 3, #Edges = 3
Adjacency Matrix:
   -1   -1   50
   40   -1   -1
   -1   30   -1

Removing Vertex 4:
Error (removeVertex_graph_AM): Vertex does not exist
(Adjacency Matrix Representation): #Vertices = 3, #Edges = 3
Adjacency Matrix:
   -1   -1   50
   40   -1   -1
   -1   30   -1

Removing Vertex 2:
(Adjacency Matrix Representation): #Vertices = 2, #Edges = 1
Adjacency Matrix:
   -1   -1
   40   -1

Removing Vertex 0:
(Adjacency Matrix Representation): #Vertices = 1, #Edges = 0
Single Vertex graph:
-1

Removing Vertex 0:
(Adjacency Matrix Representation): #Vertices = 0, #Edges = 0
<null graph>

Removing Vertex 1:
Error (removeVertex_graph_AM): graph is null
(Adjacency Matrix Representation): #Vertices = 0, #Edges = 0
<null graph>
```

Task 4: Finding a path

In *"graphs_AM.c"* file, create the following function:

```
int findpath_graph_AM(GraphAM* g, Vertex* v1, Vertex* v2)
```

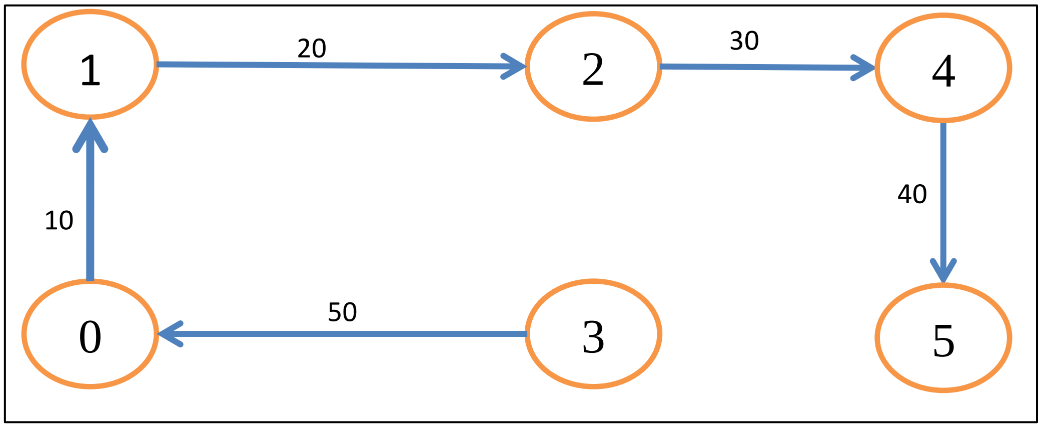
The function finds the path, if it exists, between vertex v1 and vertex v2. The function returns the cost of the path (sum of all edge weights).

Assume that you are dealing with directed weighted graphs.

For simplicity, we will assume that for every vertex, there is at most one edge coming in, and at most one edge coming in.

Also, for simplicity, we will assume that a graph will not have more than 10 vertices.

An example, of such a graph is provided below.



The function create_graph2() creates the above graph, and is provided for you.

Your function should start at v1 and then search for destination, if it is found it stops, otherwise, it gets into its (only) neighbor and it keeps searching. It should stop, when a vertex has no out neighbors (e.g. Vertex 5 in the above diagram).

If a path is found, the function should print the path from v1 to v2 before returning the cost.

Running *test_findpath()* should give the following output:

```
(Adjacency Matrix Representation): #Vertices = 6, #Edges = 5
Adjacency Matrix:
   -1   10   -1   -1   -1   -1
   -1   -1   20   -1   -1   -1
   -1   -1   -1   -1   30   -1
   50   -1   -1   -1   -1   -1
   -1   -1   -1   -1   -1   40
   -1   -1   -1   -1   -1   -1

Path from Vertex (0) to Vertex (1):
(0)-->(1)
Cost = 10

Path from Vertex (1) to Vertex (0):
no path was found
```

```
Cost = -1

Path from Vertex (0) to Vertex (2):
(0)-->(1)-->(2)
Cost = 30

Path from Vertex (3) to Vertex (2):
(3)-->(0)-->(1)-->(2)
Cost = 80

Path from Vertex (3) to Vertex (5):
(3)-->(0)-->(1)-->(2)-->(4)-->(5)
Cost = 150

Path from Vertex (5) to Vertex (3):
no path was found
Cost = -1

Path from Vertex (1) to Vertex (4):
(1)-->(2)-->(4)
Cost = 50
```