SCHOOL OF COMPUTER SCIENCES

SESSION 2019/2020 SEMESTER II

CPT 212

Assignment II: Graph Algorithms

SUBMISSION DATE: 7TH JUNE 2020

GROUP 30

Name	Matric number	Division of Task
Nurul Murshida binti Omar Bakri	144184	Strong Connectivity
Farah Mursyidah Binti Fuahaidi	144395	Cycle Detection
Nuraina Zafirah binti Abdul Rahim	144635	Shortest Path

1. Description of the chosen data structure

There are essentially three types of graph representations which are adjacency list, adjacency matrix and incident list. In this paper, we choose to implement an adjacency list to represent and initialize our graph. Adjacency list is one of the graph representations that combines adjacency matrices with edge lists. In other words, it represents all edges as a list where an array of lists is used. In this approach, each Node is holding a list of Nodes, which are directly connected with that vertex. If it reaches the end of the list, each node will be connected with null values to denote the end of the list (Adjacency lists in Data Structures) .

The main alternative for adjacency list will be adjacency matrix. Both of the graph representation types have their own trade offs in terms of Access Time and Space Needed. One drawback of adjacency list is that it does not provide a quick way to determine whether a particular edge (u,v) is present in the graph. In fact, it does not provide a place to store data such as the lengths or costs of the edges. Adjacency matrix takes only O(1) time to check the edges since it can simply be looked up using an array. Adjacency list, in contrast, provides only one way to search for an edge which is by traversing the list Adj[u] to look for v.

Despite the weakness of adjacency list in access time, it is the best graph representation compared to adjacency matrix for sparse graphs. Adjacency list representation consumes less memory space, as it does not require to store edges that do not exist. This is the main reason why the adjacency list is used for this paper. In terms of space requirement, adjacency list eliminates adjacency matrix where it will consume the array space with storing either true/false (if unweighted), or the weight of the edge. Using a naive array implementation of adjacency lists on a 32-bit computer, an adjacency list for an undirected graph requires about 8e bytes of storage, where e is the number of edges: each edge gives rise to entries in the two adjacency lists and uses four bytes in each.(Jobs 2009) On the other hand, because each entry in an adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only n2/8 bytes of contiguous space, where n is the number of vertices.(Jobs 2009)

Apart from space requirement, it is easier to find all vertices adjacent to a given vertice by using adjacency list. The list just simply has to be traversed which will take O(n) time. Below is a table to compare between both of the graph representations.

Adjacency List	Adjacency matrix
The number of edges will determine the memory usage (save a lot of memory if the graph is sparse)	Utilise O(n^2) memory
Consume a longer time to determine the presence or absence of a particular edge	Able to determine the presence or absence of a particular edge in a short amount of time
Fast to iterate over all edges	Slow to iterate over all edges
Fast to add/delete node	Slow to add/delete node
Able to add edge in a short amount of time	Able to add edge in a short amount of time

2. Description of functions

Function 1: Strongly Connected (Check Strong Connectivity)

This function is to check if the current graph has strong connectivity or not. We say that a vertex a is strongly connected to b if there exist two paths, one from a to b and another from b to a. Hence, for a graph to have strong connectivity, each vertex must be able to reach all other vertices and vice versa. Therefore, to solve this problem we will first check whether the current graph has strong connectivity or not by using the function bool isSC() . This boolean function will return true or false regarding the strong connectivity of the current graph. The main procedure to detect if a current graph is strongly connected or not is by implementing a Depth First Search algorithm, which are function DFS(v, already visited[]) and function Transpose() to reverse the edges in the graph. Depth-first search is one of the algorithms for traversing graph data structures. The algorithm starts at the root node (in this case ,vertex 0) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. In the main function, if the function isSC() returns a false value that the graph does not have strong connectivity, then we will generate a random edge between two random vertices and check again for the strong connectivity of the graph.This process will be repeated until the graph has strong connectivity.

The source code for this function is taken from https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/

Function 2: Cycle Detection

This function is to check whether the given graph is acyclic or contains cycles. A graph is said to be cyclic if it is a directed graph that contains a path from at least one vertex back to itself. Whereas an acyclic graph is a graph without any paths that connect back to itself. In this function, several procedures will be involved in determining graph cycles. Through this function, the default graph that has been initialized will be determined whether it is cyclic or acyclic. If it is cyclic, it will return true and print out 'This graph has at least a cycle'. Otherwise, at least one edge will be randomly generated to turn the graph cyclic. The procedures involved to detect cyclicity will be bool isCyclic() and bool isCyclicUtil(int v, bool visited[], bool *recStack) where depth first search (DFS) is implemented. Both of these functions are interrelated as recursive calls are utilized to detect cycles in different DFS trees as well as to recur for all the vertices adjacent to the current vertex. This will be further elaborated below using pseudocode.

The source code for this function is taken from https://www.geeksforgeeks.org/detect-cycle-in-a-graph/

Function 3: Shortest Path

This function is to calculate the shortest path between two vertices. Users are allowed to select any two vertices based on the graph to calculate a possible shortest path between the vertices by using the shortestPath(int from, int to, int V) function**.** If there is no path between the selected vertices, the function will create random edges to the graph until there is a path between the selected vertices. In this case, the algorithm used to find the shortest path is Dijkstra's algorithm. Dijkstra's algorithm is a greedy method to find the shortest path from a single source node or vertex, by building a set of nodes that have a minimum distance between the source. The algorithm will stop once the shortest path to the destination node has been determined. The only drawback for using this algorithm is that this algorithm cannot handle negative edges. Basically, Dijkstra's algorithm works by identifying all of the vertices that are connected to the current vertex with an edge. The distance to the destination vertex is calculated by adding the weight of the edge to the mark on the current vertex. All the corresponding distances of each vertex will be marked but the mark will only change if it is less than the previous mark. Therefore, the one with the smallest mark will be the current vertex and the steps will repeat again until the destination vertex is reached. Hence, the shortest distance between the vertices will be the last distance that has been marked.

The source code for this function is taken from
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/ and
https://www.tutorialspoint.com/Dijkstra-s-Algorithm-for-Adjacency-List-Representation

3. Pseudocodes of the functions

FUNCTION 1 : STRONG CONNECTIVITY

Under this function, the user can use the main feature which is **Check Strong Connectivity** of the graph as well as three other features which are **Add New Edge, Delete Edge** and **Back**. Three main pseudocodes of **Check Strong Connectivity** function will be **isSC()**, **DFS (v, already visited [])** and **TransposeGraph().**Firstly,we will check whether the graph is strongly connected or not using function **isSC()**.If it is not a strongly connected graph,we will generating a random edge between the vertices until the graph is strongly connected.In the **isSC()** function, it will check if a vertex can reach all other vertices and vice versa.This can be perform by using implemented Depth First Search algorithm , **DFS(v,already visited []).**Then,if a vertex can reach all other vertices,we will use a function **TransposeGraph()** where it will reversed all the edges inside the graph and perform a depth first search again to make sure the current vertex can reach all other vertices.The additional feature **Add New Edge** and **Delete Edge** is to allow the user to manually add and delete edges in the graph. These functions will perform some validation and checking to check if the two vertices entered exist or not by using the function **Traverse(from,to).**Lastly, additional function **Back** is to allow the user to go back to the main menu.

Procedure isSC()
Begin

Boolean already visited[V] - store list of vertices

Firstly, we initialize all the vertices as not visited (for first Depth First Search)
V - Number of vertices in the graph
For i ← 0 to i < V
already visited [i] ← false

Secondly, we perform Depth First Search traversal starting from the first vertex which is 0 by calling function DFS(), If Depth First Search traversal does not visit all the vertices, then return false as for not strongly connected
DFS(0,already visited)
For i ← 0 to i < V
If already visited [i] is equal to false
return false

Next, if Depth First Search traversal does visit all the vertices then we create a reversed or transpose graph by calling function TransposeGraph(). This function will reverse all edges of the graph.
StrongConnect gr - A new reversed graph
StrongConnect gr = TransposeGraph()

Lastly, we mark all the vertices as not visited (for second Depth First Search)
For i ← 0 to i < V
already visited [i] ← false

Again we perform Depth First Search traversal for reversed graph starting from the first vertex, same vertex in step 2 (the original graph). If Depth First Search traversal does not visit all the vertices, then return false as for not strongly connected otherwise, return true as for the graph is strongly connected.
gr.DFS(0,already visited)
For i ← 0 to i < V
If already visited [i] is equal to false
return false
return true

End

DFS() is a recursive function to print Depth First Search traversal starting for vertex v

Procedure DFS (v, already visited [])
Begin
v - number of vertices
already visited - list array of vertices

First, we will mark the current node as visited and print it
already visited [v] ← true

Next, recur for all the vertices that are adjacent to this current vertex by using a pointer
list<int>::iterator i
For i ← adj [v].begin to i not adj [v].end
If already visited [i] is equal to false
DFS(i,already visited)

End

TransposeGraph() is a function that returns a reverse or transpose of the original graph

Procedure TransposeGraph()
Begin
StrongConnect gg(V) - create a new graph with (V) vertices
StrongConnect gg(V)
For v ← 0 to v < V

Next, recur for all the vertices that are adjacent to this current vertex by using a pointer
list<int>::iterator i

By using the concept link list and stack, first, we point the pointer at the first vertex, then while there are still adjacent vertex, then it will push back the current vertex(v) to the vertex (i)
For i ← adj [v].begin to i not adj [v].end
gg.adj [i].push back (v)
End of the inner loop
End of the outer loop

Lastly, we will return the reversed graph
return gg
End

Traverse(from,to) is a function that will check whether the edge exists or not between the two vertices.

Procedure Traverse(from, to)
Begin
We will first set the boolean found as false
found ← false

Then, we will create a pointer to traverse all the adjacent nodes of the "from" node. If the " to" node is found, then boolean found is set to true, otherwise false
list<pair<int,int>::iterator i
For i ← adj [from].begin to i not adj [from].end
If (*i).first is equal to to
found ← true

break from the loop

Else

found ← false

End of loop

return found

End

In the main function

bool m - boolean function to return if the graph is strongly connected or not
g1 - the object of the graph
from - the "from" vertex
to - the "to" vertex
wt - wight of the edge
bool n - boolean function to return if the edge already exists or not

We check if the current graph is strongly connected or not by using function isSC() and store the boolean value in variable m.
m ← g1.isSC()
If m is equal to false
do

Create Strong Connectivity on the graph by generating random edges between two vertices by using the function rand. This function allows it to generate a self loop edge.
from = rand() mod 5 as for there are 5 vertices in the graph
to = rand() mod 5 as for there are 5 vertices in the graph
wt = rand() mod 15

Check whether the generated edge already exists or not by using function traverse(from, to)
n ← traverse(from, to)
If n is equal to false
Add the edge between the vertices
g1.addEdge(from,to,wt)
Check again if the graph is strongly connected or not
m ← g1.isSC()
While m is equal to false

FUNCTION 2 : CYCLE DETECTION

Under this function, the user will have the main feature which is **Check Cycle** and other two additional features which are **Add New Edge, Delete Edge** and **View Graph** which have been mentioned above. The pseudocodes below are the process in determining and printing cycles in the graph. From the main function, it will go to **isCyclic** function to determine the cyclicity of the graph. A recursive function **isCylicUtil** will be called in **isCyclic** method to detect cycles by using Depth First Search Algorithm. The results will be returned to **isCyclic** method and it will be printed out. Method **isCyclic** will return true or false to main which will indicate the cyclicity of the graph. If the graph is acyclic, random edges will be added to the graph that will automatically make the graph cyclic. Before the random edges are added, the edges in the graph will be traversed first to ensure not to produce existing edges, this process will happen in procedure **Traverse** which is an added feature.Other Added features have also been included such as addNewEdge and deleteEdge as what mentioned in the Strong Connectivity explanation.

Procedure isCyclic( )
Begin

int Num - integer array to mark node that has been visited
bool recStack - boolean arrays as part of recursion stack
int cycle - integer array to mark cycle node

Initialize every element of array to 0 and false

For m← 0 to m< v
Num[m] ← 0
recStack[m] ← false
For v← 0 to v < V
Call recursive helper to detect cycle in different Depth First Search Tree
If isCyclicUtil(v, num,i,recStack,cycle) is equal to true

If the result returned from the recursive function is true, the below will be executed to print cycle
For n← 0 to n<V

If the value in the array is not equal to zero
If cycle[n] is not equal to zero

If the value in the array is equal to 5, change the value to zero
If cycle[n] is equal to 5
Cycle[n] is 0

The value of the cycle which is less than 0 or more than 4 (gibberish value - vertex that does not involve in the detected cycle) will not be printed out

If it is in the valid range, it will pushed in a vector
If cycle[n] is bigger than -1 and less than 4
Push back in vector1

The vector will then be printed, this vector is the list of vertex that involve in the detected cycle
For auto it is equal to v1.begin and not equal to v1.end
Print *it

It will return true to main
Return true

If the result for the returned function is false, it will return false to the main which means cycle is not detected
Return false

Procedure isCyclicUtil(v, num,i,recStack,cycle)
Declare an iterator to traverse list
list<int>::iterator u

If array Num[v] (array to mark nodes that will be visited) is equal to zero - means it is still unvisited
If Num[v] is equal to 0

Update value of current node so that it will not equal to 0 - means it has already been visited
Num[v] ← i++ ***//update the value of nodes visited so that it will not equal to 0***

Update recStack so that it will not equal to false
recStack[v] is equal to true

Traverse to the vertex that is adjacent to the current vertice by using iterator
For all vertices adjacent to this vertex

If the value of the place where the iterator is pointing (the adjacent of the current vertex) is equal to zero -Means it has not been visited yet
If *u is not visited

Push the adjacent of the current vertex into array cycle

Cycle[z] is equal to *u

If the value of vertex that is adjacent to the current vertice is equal to zero - not visited yet, it will enter recursive caller

If (num[*u] is equal to zero and isCyclicUtil (*u,num,pred,i, recStack,cycle))

Return true

If the value of the place where the iterator is pointing (the adjacent of the current vertex) is not equal to zero - means it has been visited and it is placed in the recStack is equal to true- it denotes that the cycle is already found

Else if (recStack[*u])

Push the vertex to array cycle

The below code is for printing purposes

If v is equal to zero

Cycle[z] is equal to 5

Z++

Else

Cycle[z] is equal to *u;

Z++

It will return true to function caller which indicates the cycle is found

Return true

If array Num[v] (array to mark nodes that will be visited) is not equal to zero - means every nodes are visited and no cycle is found, return false

recStack[v] is equal to false

Return false

In the main function

bool p- boolean function to return if the graph is strongly connected or not
g1 - the object of the graph
from - the "from" vertex
to - the "to" vertex
wt - wight of the edge
bool q - boolean function to return if the edge already exists or not

The graph is checked whether it is acyclic or not

p ← g1.isCyclic()

If it return 0, means the graph is acyclic

If p is equal to 0

It will enter do while loop

do

Create Cycle on the graph by generating random edges between two vertices by using the function rand. This function allows it to generate a self loop edge.

from = rand() mod 5 *there are 5 vertices in the graph*

to = rand() mod 5 as for *there are 5 vertices in the graph*

wt = rand() mod 15 *limit the weight to value 15*

Check whether the generated edge already exists or not by using function traverse(from, to)

q← traverse(from, to)

If q is equal to false

Add the edge between the vertices using function addEdge that has been mentioned before

g1.addEdge(from,to,wt)

Check again if the graph Cyclic or not
p← g1.isCylic()

Do while loop will continue to loop if m is equal to false
While p is equal to false

FUNCTION 3 : SHORTEST PATH

For this function, the user will have several options included in the menu provided. The main feature includes **Calculate Shortest Path,** while **Add Edge, Delete Edge and Back** are the additional features in this function. This function is mainly focused on **Calculate Shortest Path** where Dijkstra algorithm is implemented in **shortestPath** function. When the user chooses **Calculate Shortest Path**, the default graph will be displayed to the user. The user will be required to select two vertices they want to find the shortest path for. After selecting the vertices, the function will check whether the vertices are reachable through **isReachable** function. If both vertices are reachable which means that the path already exists, this function will simply display the shortest path using the **printPath** function. However, if it returns false, it means that the path from the source vertex to the destination vertex does not exist. Therefore, random edges will be generated until the path exists. To avoid any duplicate edges, **traverse** function is used to check whether the edges already exist and it will continue generating random edges until the edges created are distinct and the path exists. After that, the new graph will be updated. Lastly, it will perform the **shortestPath** function again to compute the shortest path between the selected vertices. Below are the pseudocodes and the explanation for the code used in this function.

Procedure shortestPath(from, to, V)
Begin

An empty priority queue is created where every element stored in pq is in pairs(weight, vertex). Weight refers to the distance, and is used as the first element of a pair as the first element by default used to compare two pairs.
priority_queue< iPair, vector <iPair> , greater<iPair> > pq

Then, it will initialize distances of all vertices as infinite.
vector<int> dist(V, INF)

*Source vertex, **from** is inserted into the pq and the distance is made as 0.*
pq.push(make_pair(0, from)
dist[from] ← 0

*At first, **done[from]** is initialized as true indicating that the source vertex has been visited while **prev** indicating the previous node is initialized as undefined because there is no previous vertex yet.*
vector<bool> done(V, false)
done[from] ← true
vector<int> prev(V, -1)

*For the while loop, while either **pq** does not become empty, the minimum distance vertex is extracted from the **pq**. This is because in the priority queue, the element having higher priority is popped first. '**u**' indicates the extracted vertex.*
while pq is not empty do
u := pq.top().second
pq.pop()

*Next, iterator **i** is declared to traverse all the adjacent vertex.*
list< pair<int, int> >::iterator i;

*Then it will loop through all adjacent of **u***
For i ← adj [u].begin to i not adj [u].end do

Pointer i points to the first and second element in the pair which are the destination vertex and the weight of the edges.
to := (*i).first
weight := (*i).second;

*If there is a shorter path to **to** through **u***
if not done[to] and dist[u]+weight < dist[to] then

*Distance of **to** will be updated*
dist[to] := dist[u]+weight

Now, u will be the previous node of the current vertex
prev[to] := u

*While, **to** is inserted into the pq even if **to** is already there*
pq.push(make_pair(dist[to], to))

end for loop
done[u] ← true
end while loop

Lastly, the shortest path between the two selected vertices will be printed out along with the route taken starting from the source vertex to the destination vertex.
For i ← 0 to i<V do
if i equal to to
Print dist[i]
printPath(prev,i)
end for loop
End

Procedure isReachable(from, to)
Begin
This procedure of detecting the reachability between the source vertex and the destination vertex is implemented using Breadth First Search (BFS). BFS is used to find the path between two vertices. At first, if the vertex is self looping, the function will return true indicating that it is already reachable.
if from == to then
return true

Secondly, the boolean visited is declared for every vertex that will be visited.
visited ← new bool[V]

For every vertex, it is marked as not visited at first.
For i ← 0 to i<V do
visited[i]← false

Then, a queue for BFS is created to store all the vertices that have been marked as visited.
list<int> queue;

Once a vertex is marked as visited, it will be enqueued in the queue starting from the start node.
visited[from]← true
queue.push_back(from)

```
    Next, iterator i is declared to traverse the adjacent vertex.
    list<pair<int, int> > :: iterator i;

    Then, while the queue is not empty, the front element of the queue is being dequeued and all its adjacent elements are iterated.
    while queue is not empty do
        from := queue.front()
        queue.pop_front()

        It will loop through all adjacent of from
        For i ← adj [from].begin to i not adj [from].end  do

            and it will return true if any of the adjacent elements is the destination node.
            if (*i).first == to then
                return true

            All the adjacent and unvisited vertices are then pushed in the queue and are marked as visited.
             if not visited[(*i).first] then
                visited[(*i).first] ← true
                queue.push_back((*i).first)

        end for loop
    end while loop
        However, if the destination is not reached in BFS, it will return false indicating that the vertices are unreachable.
        return false
End
```

Procedure printPath(&prev, i)

```
Begin
    If i < 0
        return
    printPath(prev, prev[i])
    Print i
End
```

This procedure is for displaying the route taken starting from the source vertex to the destination vertex provided that the previous node is always defined. The value of **i** indicating the visited vertex is printed out. If the index **i** of the previous node is less than zero, it will return back to the previous function.

GENERAL FUNCTIONS

Add New Edges

This is an additional feature in function 1, function 2 and function 3. It allows the user to add an edge between two vertices. It will attempt the user to key in the two vertices (from and to) and the weight of the edge. Then it will check whether the key in edge is already existing or not by using function Traverse() (refer to Function Check Strong Connectivity). If it does not exist, then it will add the edge otherwise it will not and print an error message.

Procedure addEdge(from,to,weight)

```
Begin
    To all respective class, add the "to" node to "from" node
    adj[from].push_back(make_pair(to,weight)
```

End

Delete Edge

This is an additional feature in function 1, function 2 and function 3. It allows the user to delete an existing edge between two vertices. It will attempt the user to key in the two vertices (from and to) to be deleted. Then it will check whether the key in edge is already existing or not by using function Traverse() (refer to Function Check Strong Connectivity). If it does exist, then it will delete the edge otherwise it will not and print an error message.

Procedure deleteEdge (from, to)

```
Begin
        boolean found -  boolean function if the edge is exists or not
        stack < int > s - stack of the adjacency nodes
        p - counter of how many times we copy the nodes
        x - value of the top of the stack s
        I - pointer i
        u - pointer u

        We create two pointers to traverse the adjacent nodes
        list< pair<int,int> >::iterator i;
        list< pair<int,int> >::iterator u;

        For i ← adj [from].begin to i not adj [from].end  do
                If (*i).first is equal to to
                        found ← true
                        break from the loop
                Else
                        found ← false
        End of loop

        If found is equal to true

                Pointer u will point to the same as pointer i
                u is equal to i
                p ← 0
                u ← u+1

Then, we copy the adjacent nodes of the "from" vertex starting the next from the vertex that we want to delete. We copy to the stack s.
                While u is not equal to the last node
                        s.push(make_pair((*u).first , (*u).second))
                        u ← u+1
                        p ← p+1

                Then, we delete the adjacent nodes of the "from" vertex starting from the vertex that we want to delete
                For m ← 0 to m <= p
                        adj [ from ].pop back()

                Then, we insert back the vertices in the stack s to the vertex " from ".
                For m ← 0 to m < p
                        x ← s.top().first
                        y ← s.top().second
                        adj[from].push_back(make_pair(x,y))
                        s.pop()
End
```

View Graph

This is an additional feature in the main function. It allows the user to view the graph.

Procedure display_AdjList()

Begin

v - current number of the vertices
V - number of the vertices in the graph (5)
For v ← 0 to v < V

We create a pointer to traverse the adjacent nodes , then we display by using switch (v) and switch (i) as v and i are the markup for the vertex names.

list<pair<int, int> > :: iterator i
For i ← adj [v].begin to i not adj [v].end
Switch (v)
case 0 : print from HK
Switch (i)
Case 0 : print to HK
Case 1 : print to AU
Case 2 : print to EG
Case 3 : print to DK
Case 4 : print to BE

End

Reset Graph

This function enables the user to reset the graph to the default values.First, we will delete all the edges in the graph. Then, we initialize back the graph to the default values.

The outer for loop is for the "from" node and the inner for loop is for "to" node.Hence, it will delete all the adjacent vertices from the current vertex.

For i ← 0 to i < 5
For z ← 0 to z < 5
g1.deleteEdge(i,z)
End of inner for loop
End of outer for loop

Then, we initialize the graph to the default values by adding the edge one by one.

g1.addEdge(0, 4, 9)
g1.addEdge(1, 0, 6)
g1.addEdge(1, 2, 12)
g1.addEdge(3, 2, 4)
g1.addEdge(3, 4, 1)

d. Results:

```
                    Assignment II: Graph Algorithms
        ********************************************************
         [1] Strong Connectivity
         [2] Cycle Detection
         [3] Shortest Path
         [4] View graph
         [5] Reset graph
         [6] Exit

  Your choice :
```

1)Main Menu
This is the main menu where users can choose the functions

```
                      STRONG CONNECTIVITY
        ***********************************************
         [1] Check Strong Connectivity
         [2] Add New Edge
         [3] Delete Edge
         [4] Back

  Your choice :
```

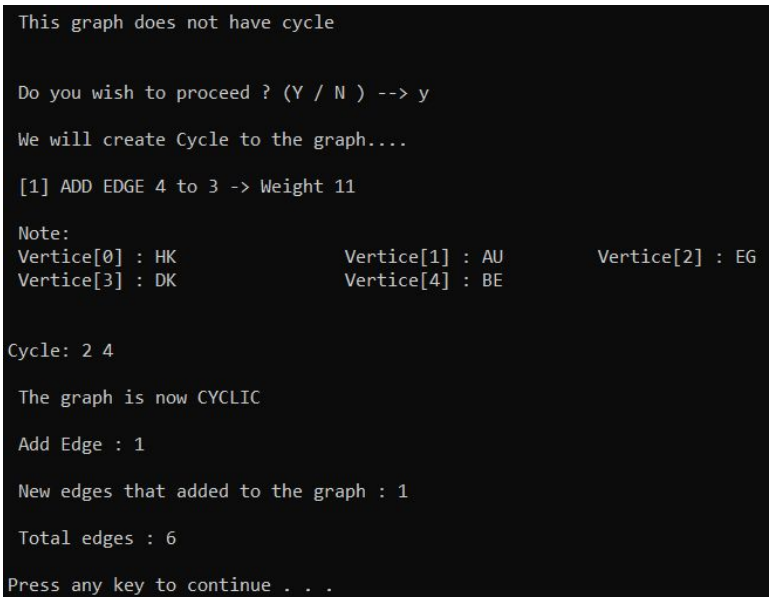
2) Function 1 : Strong Connectivity
There are four features in this function which are check strong connectivity,add new edge,delete edge and back to the main menu.

```
This graph does not have Strong Connectivity


Do you wish to proceed ? (Y / N ) --> y

We will create Strong Connectivity to the graph....


Note:
Vertice[0] : HK                Vertice[1] : AU         Vertice[2] : EG
Vertice[3] : DK                Vertice[4] : BE

[1] ADD EDGE 3 to 3 -> Weight 7
[2] ADD EDGE 4 to 0 -> Weight 5
[3] ADD EDGE 2 to 1 -> Weight 6
[4] ADD EDGE 4 to 2 -> Weight 3
[5] ADD EDGE 2 to 2 -> Weight 6
[6] ADD EDGE 1 to 3 -> Weight 5

The graph is now STRONGLY CONNECTED

Add Edge : 6

New edges that added to the graph : 6

Total edges : 11

Press any key to continue . . .
```

3) Check Strong Connectivity
When the user chooses 1 as for checking strong connectivity, the system displays whether the graph has strong connectivity or not. Then, it will prompt the user to proceed to the next process which is creating strong connectivity to the graph by generating random edges. After the graph finally has strong connectivity, it will then show the number of new edges that have been added and total edges of the graph.

```
                      CYCLE DETECTION
        ***********************************************
         [1] Check Cycle
         [2] Add New Edge
         [3] Delete Edge
         [4] Back

Your choice :
```

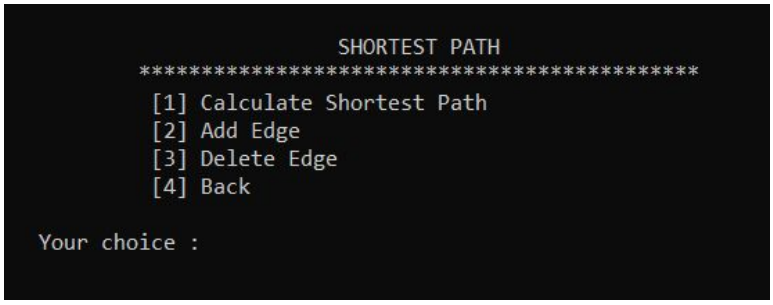
4) Function 2 :Check Detection
There are four features in this function which are check cycle,add new edge,delete edge and back to the main menu



5) Check Cycle

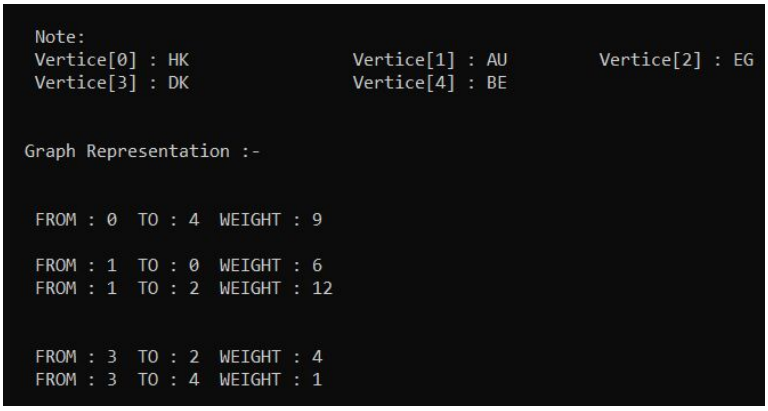
This function will check whether a cycle exists or not in the graph. If it does not exist, the user will be prompted with a question whether to proceed with generating random edges or not. Enter Y/y for yes or N/n for no.

Random edges will be generated until cyclicity is achieved. Cycle will then be printed out and a message '*The graph is now cyclic'* will be printed.



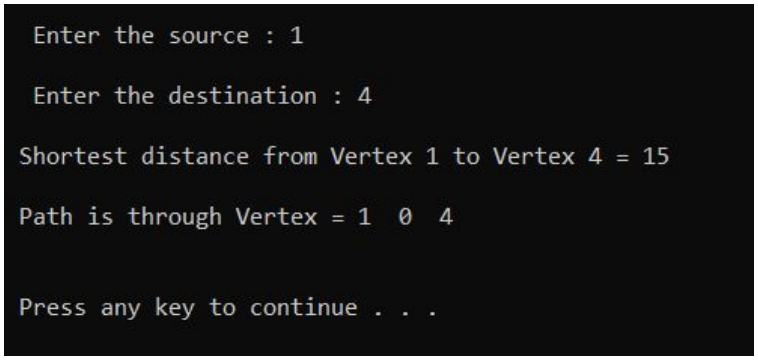
6) Function 3 : Shortest Path

The menu displays the options provided for this function. The options include calculate shortest path, add new edge, delete edge and back to main menu.



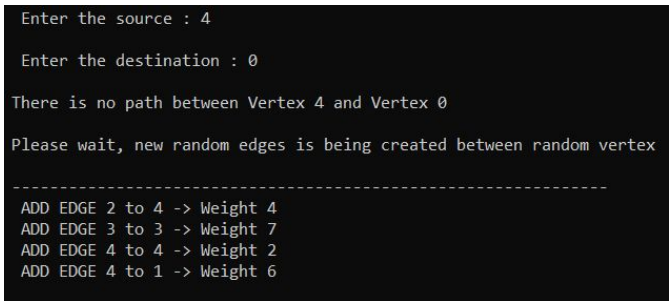
7) Calculate shortest path

The graph will be displayed to the user so that the user can refer to the vertices that exist. Hence, they can choose any vertices to compute the shortest path.



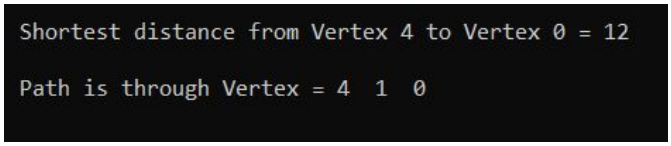
8) Select vertices (has path)

The user is required to enter any vertex as the source and the destination for the shortest path to be computed. This picture displays the result when the path of the selected vertices already existed. The shortest distance computed from vertex 1 to vertex 4 is 15.



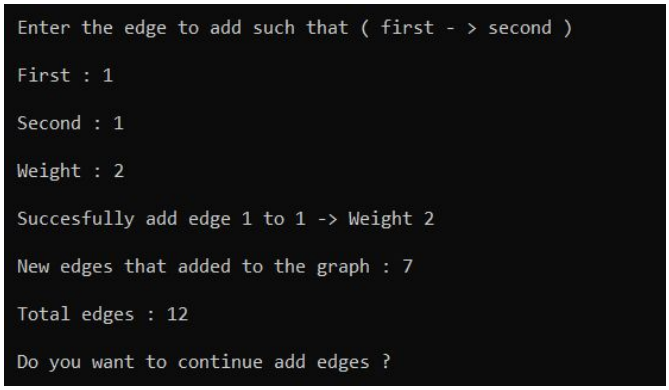
9) Select vertices (has no path)

If the user happened to select two vertices where the path does not exist between the vertices, random edges will be generated until there is a path and the new edges will be updated to the graph. Based on the picture, vertex 4 and vertex 0 has no path between them.



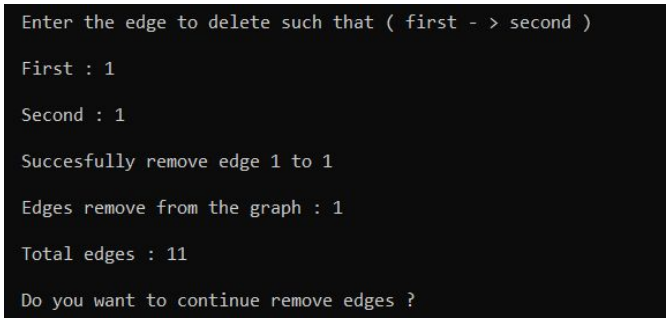
10) Print shortest distance

Once the shortest path between two selected vertices is computed, the result will be displayed along with the route taken to the destination.



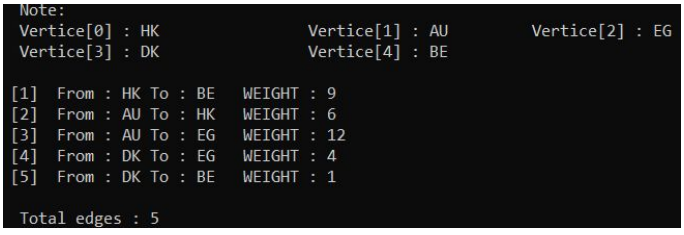
11) Add New Edge

This is an additional feature that exists in function 1, function 2 and function 3.It allows the user to manually add an edge between two existing vertices. After the user enter the vertices, the program will check whether the edge exists or not. If it exists, the program then will display an error message (Failed add edge . The edge from “ “ to “ “ already exists!)otherwise it will inform the user the edges have been added and display total edges in the graph.



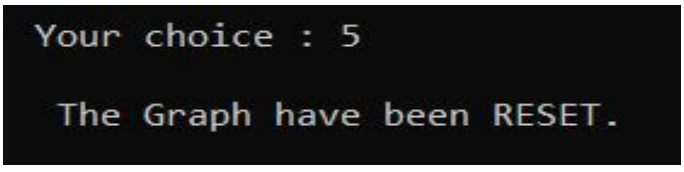
12) Delete Edge

This is an additional feature where it allows the user to delete an existing edge. It will prompt the user to enter two vertices where the edge between them will be deleted. Then it will check for existing edges. If the edge between two vertices entered exists, then it will delete the edge and display a message to the user .Otherwise, the program will display an error message (Failed add edge . The edge from “ “ to “ “ does not exist!)



13) View Graph

This additional feature allows the user to view the graph. It will print all the edges and display the total of edges that exist in the graph.



14) Reset Graph

This function will allow the user to reset the graph to the default values after the user has done modification on the graph.