CS 513

DS Lab

Assignment 4

Graph

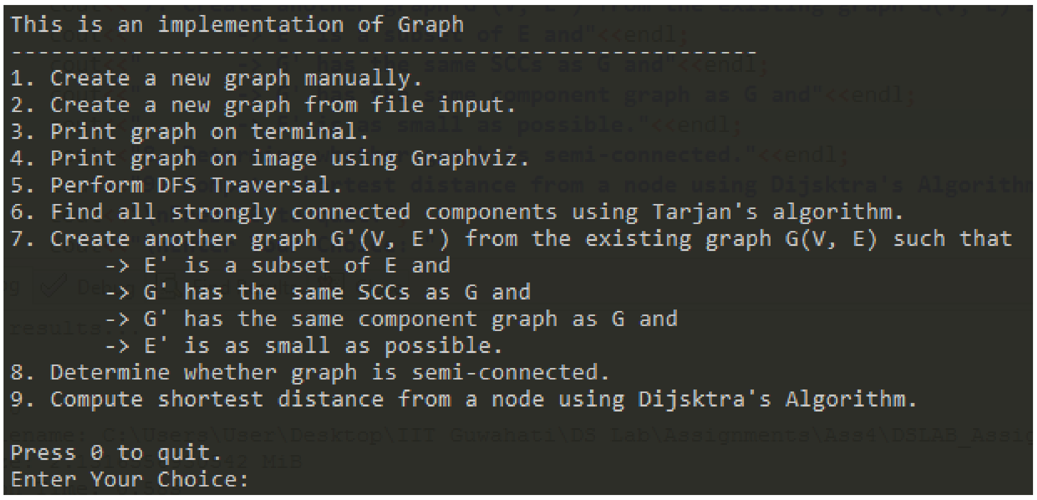
Abhijeet Padhy

Roll: 214101001

Index

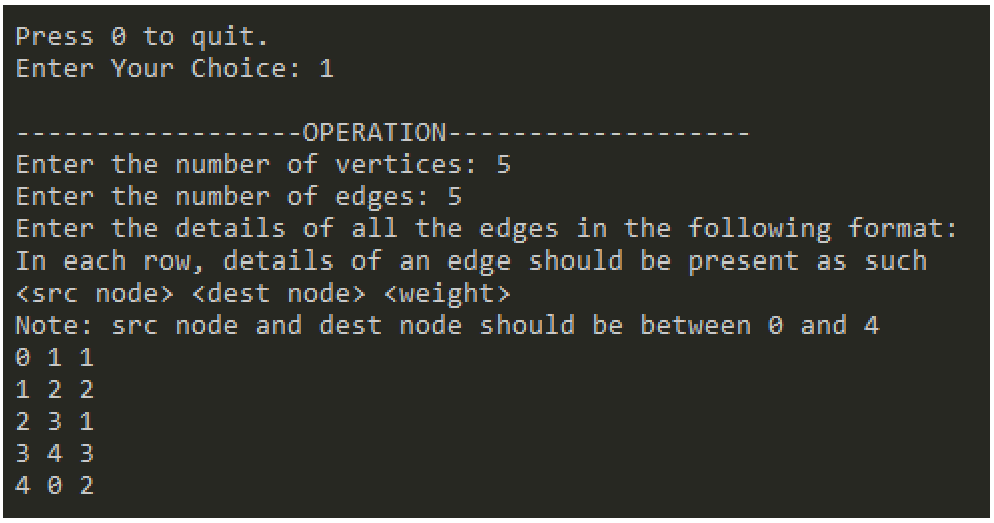
Basic Information

The program starts with the following menu:



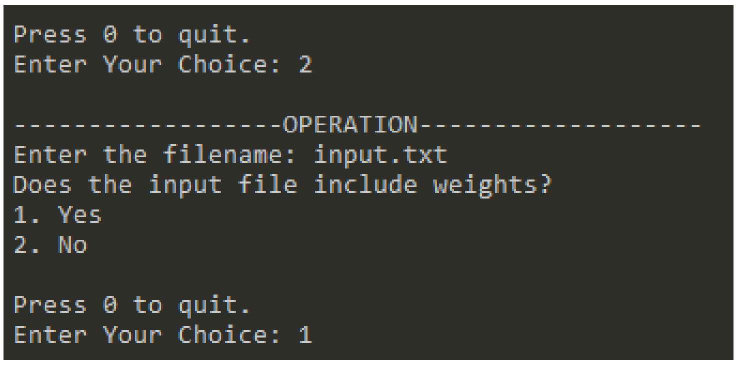
```
This is an implementation of Graph
-------------------------------------------------------
1. Create a new graph manually.
2. Create a new graph from file input.
3. Print graph on terminal.
4. Print graph on image using Graphviz.
5. Perform DFS Traversal.
6. Find all strongly connected components using Tarjan's algorithm.
7. Create another graph G'(V, E') from the existing graph G(V, E) such that
       -> E' is a subset of E and
       -> G' has the same SCCs as G and
       -> G' has the same component graph as G and
       -> E' is as small as possible.
8. Determine whether graph is semi-connected.
9. Compute shortest distance from a node using Dijsktra's Algorithm.

Press 0 to quit.
Enter Your Choice:
```

Option 1: Use this to enter the edges of the graph manually.



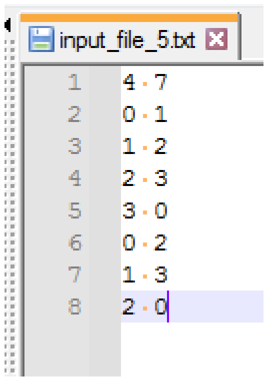
```
Press 0 to quit.
Enter Your Choice: 1

------------------OPERATION-------------------
Enter the number of vertices: 5
Enter the number of edges: 5
Enter the details of all the edges in the following format:
In each row, details of an edge should be present as such
<src node> <dest node> <weight>
Note: src node and dest node should be between 0 and 4
0 1 1
1 2 2
2 3 1
3 4 3
4 0 2
```

Option 2: Use this to take file input to initialise the edges of the graph.



```
Press 0 to quit.
Enter Your Choice: 2

------------------OPERATION-------------------
Enter the filename: input.txt
Does the input file include weights?
1. Yes
2. No

Press 0 to quit.
Enter Your Choice: 1
```

Enter the name of the input file. It then asks if the input file also contains weights. Press 1 if input file contains weights. Press 0 if input file does not contain weights. The format of the input file is shown below:

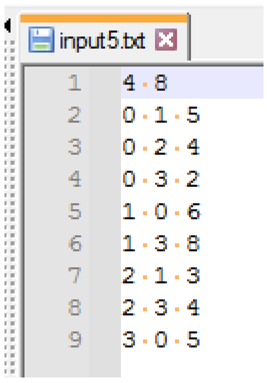


```
4 7
0 1
1 2
2 3
3 0
0 2
1 3
2 0
```

Input file without weights

The first line contains two numbers (4 and 7 in this example). The first number of first line is the number of vertices in the graph. The second number is the number of edges. The lines following the first line contain the edge information. Each row consists of two numbers, source vertex and destination vertex of the edge. In the above example, there are 4 vertices and 7 edges. Since edge weight is not specified, for each edge, the weight is 1.

However, if we choose to provide the edge weights, the option 2 should be used in the menu and the input file format corresponding to this is:



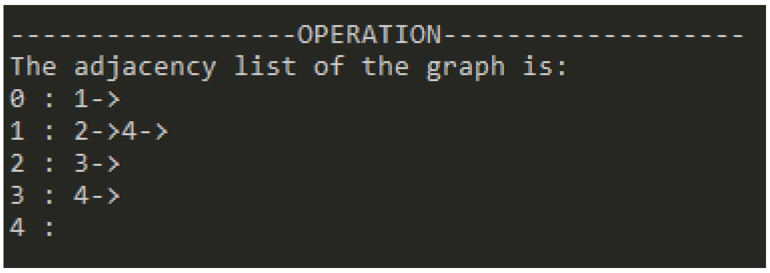
```
4 8
0 1 5
0 2 4
0 3 2
1 0 6
1 3 8
2 1 3
2 3 4
3 0 5
```

Input file with weights.

Here everything is same as the previous case except the fact that each row now contains a weight value. Thus each row consists of <src vertex> <dest vertex> <weight of the edge>.

Once file input is completed, the graph is initialised. We can then perform various operations on it.

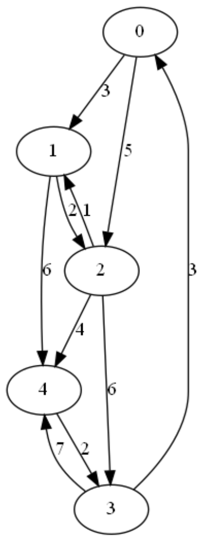
Option 3: If we want to print the adjacency list representation of the graph, use this option. The output produced is as such:



```
------------------OPERATION-------------------
The adjacency list of the graph is:
0 : 1->
1 : 2->4->
2 : 3->
3 : 4->
4 :
```

Each row corresponds to a vertex. The first number of a row is the source vertex number to all the edges going out from it. The numbers following the first number are the destination vertices of the edges going out from the source vertex. Thus vertex 1 has an edge to vertex 0, vertex 2 and vertex 4.

Option 4: Use this option to print an ordinary image of the graph. An image looks like this:



Vertex names are written inside the nodes. The edge weights are written as labels next to the edges.

Options 5 to 9 will be explained in detail in the following sections.

Note: If number of vertices in the graph is N, vertices can only have names from 0 to N-1. No other vertex number is allowed.

<u>Class Implementation</u>

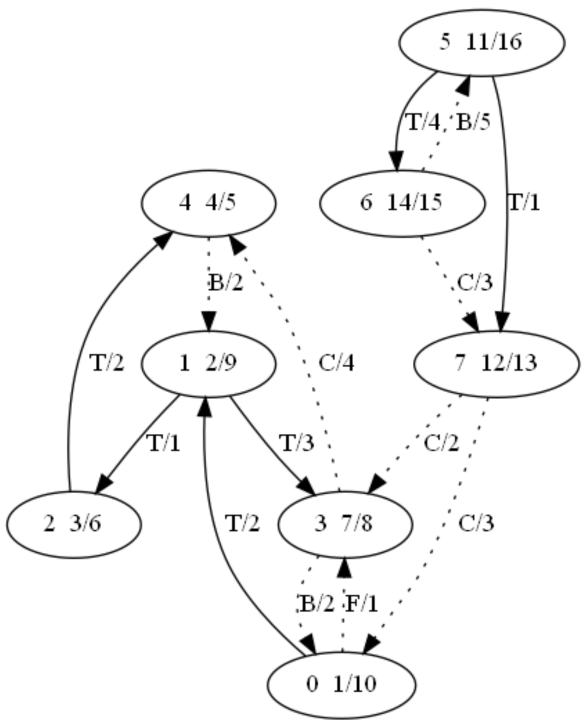
The Graph Data Structure is implemented in the form of a class. It has two member variables:

1. **int V; // stores number of vertices**
2. **vector<pair<int,int> >* adj; // stores the adjacency list**

DFS Traversal

Q1. Perform DFS traversal on the graph and classify all the edges. After DFS show the annotated graph where each vertex is labelled with the DFS start and end time and each edge is labelled with its edge type (tree/forward/back/cross).

Use option 5 to run DFS Traversal on the graph. Specify the vertex from where DFS traversal should start. The output produced by DFS Traversal from vertex 0 is as shown below:



Each node consists of three values:
<Vertex Number> <Start Time>/ <End Time>
Vertex number starts from 0 to N-1 where N is the number of nodes.
Time starts from 1 and then increases at each visit of a vertex. The time spent on a vertex is 1.

Each edge has a label of this form:
<Type of edge>/<Edge Weight>
An edge can be of the following four types:

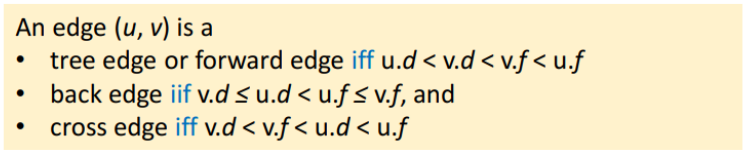
Edge Type	**Denoted By**
Tree	Black, solid, T
Forward	Black, dotted, F
Back	Black, dotted, B
Cross	Black, dotted, C

The vertex from which we start DFS Traversal will have start time as 1. All the tree edges are denoted by solid edges. Thus we can understand the DFS Tree just by following the solid tree edges.

Function Prototypes:

void dfs_traversal(bool[], int[], int[], int, FILE *);
void dfs_traversal(int);

Implementation:

1. visited array is used to keep track of nodes that are already visited.
2. discovery_time array is used to store the time when the node was visited for the first time.
3. finish_time array is used to store the time when the traversal leaves the node.
4. First call dfs traversal on the vertex provided by user.
5. In case the graph has disconnected components, we need to call traversal on them too. So we make use of a loop to run dfs traversal on those nodes which are not visited.
6. Inside a particular call to DFS Traversal, we first mark the node visited.
7. Update the discovery_time.
8. We call further dfs traversal on nodes adjacent to the current node provided the adjacent node is not already visited.
9. Before leaving the current vertex, at the end of the function call, we update the finish_time .
10. Based on few conditions, the edges are classified:
11. 

An edge (u, v) is a
- tree edge or forward edge iff u.d < v.d < v.f < u.f
- back edge iif v.d ≤ u.d < u.f ≤ v.f, and
- cross edge iff v.d < v.f < u.d < u.f

<u>**Output:**</u>

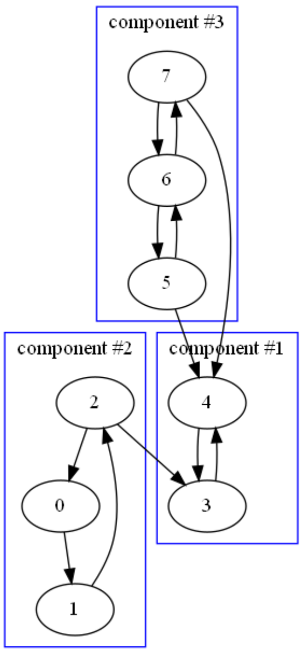
Please run the below command to create the image:
step 1: dot -Tpng dfs_traversal.gv -o dfs_traversal.png
step 2: Open the file dfs_traversal.png to view the output.

All the required information regarding DFS Traversal can be seen from the image.

Strongly connected components using Tarjan’s Algorithm

Q2. Find all strongly connected components using Tarjan’s algorithm.

Use option 6 to find out the strongly connected components using Tarjan’s Algorithm. The output of this method produces an image of the form shown in right. Each strongly connected component is shown enclosed in a rectangle.



Function Prototypes:

1. **vector<vector<int>> *find_scc();**

2. **void find_scc(int, bool[], int[], bool[], stack<int>*, int[], vector<vector<int>>*);**

3. **void output_scc(vector<vector<int>> *);**

Implementation:

Low(v) is the lowest numbered vertex that is reachable from v
by taking zero or more tree edges and then possibly one back edge
(in that order).

Low(v) is the minimum of

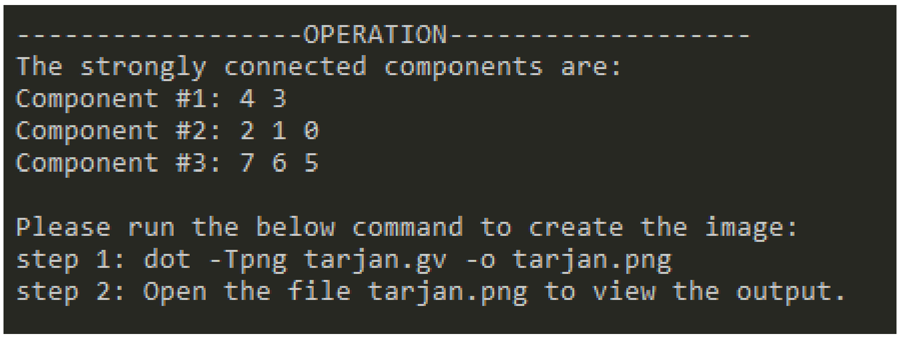
1. DFS number of v
2. The lowest DFS number of all w’s such that (v,w) is a back edge
3. The lowest Low(v) among all tree edges(v, w).

visited array stores whether a vertex is visited or not. Initially all the vertices are unvisited.
disc array stores DFS number of a vertex.
low array stores low number of a vertex.
stk is a stack which stores the elements of a particular component.

Steps:

1. For all the vertices v that are not visited during DFS call we perform DFS on it.
2. Inside a DFS call
   a. Update visited and DFS number. Set low value as dfs number when it is visited for the first time. Add the node to stack.
   b. For all the vertices v adjacent to the current vertex u:
      i. If it is a tree edge, perform DFS traversal on v. Then update low value of u as the minimum of low value of u and low value of v.
      ii. If it is a back edge, update low value of u as the minimum of low value of u and dfs number of v.
   c. If low value of u is equal to dfs number of u, start popping all the vertices present in the stack and add them to a vector. This vector contains all the nodes of the component. This vector is then returned.
3. Likewise all the SCCs are found out.

<u>Output:</u>



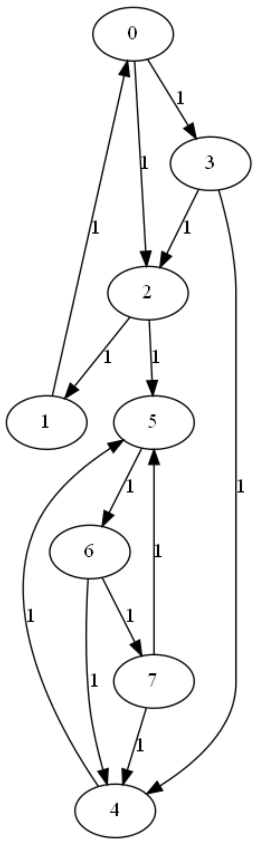
```
------------------OPERATION-------------------
The strongly connected components are:
Component #1: 4 3
Component #2: 2 1 0
Component #3: 7 6 5

Please run the below command to create the image:
step 1: dot -Tpng tarjan.gv -o tarjan.png
step 2: Open the file tarjan.png to view the output.
```

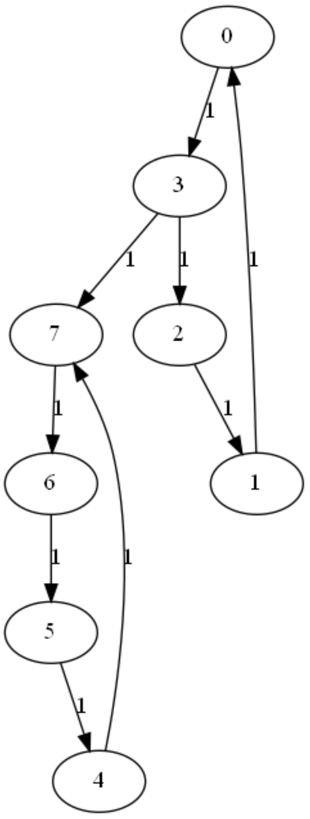
The vertices present in each component are displayed on the terminal.
Please run the below command to create the image:
step 1: dot -Tpng tarjan.gv -o tarjan.png
step 2: Open the file tarjan.png to view the output.

Removal of Edges from a graph

Q3. Given a directed graph G = (V, E), create another graph G'= (V, E'), where E' is a subset of E, such that (a) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E' is as small as possible.

To understand how this problem works, we have to look at the outputs shown below. These outputs are produced by the program. The graph on left is the original graph. In the program, use option 7 to get output to this problem. The output of the operation is shown on the right.

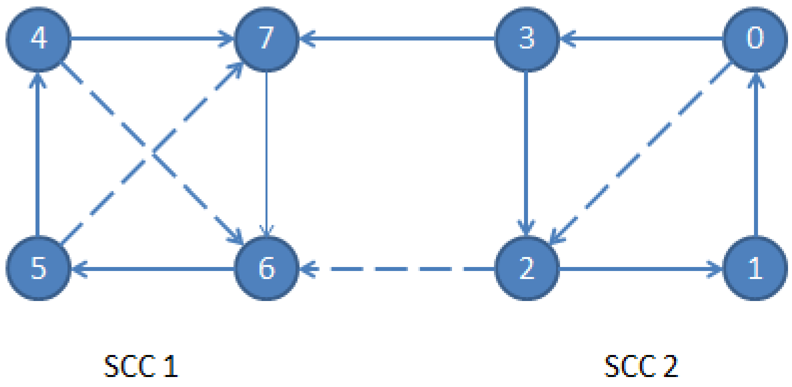




Graph G (V, E)

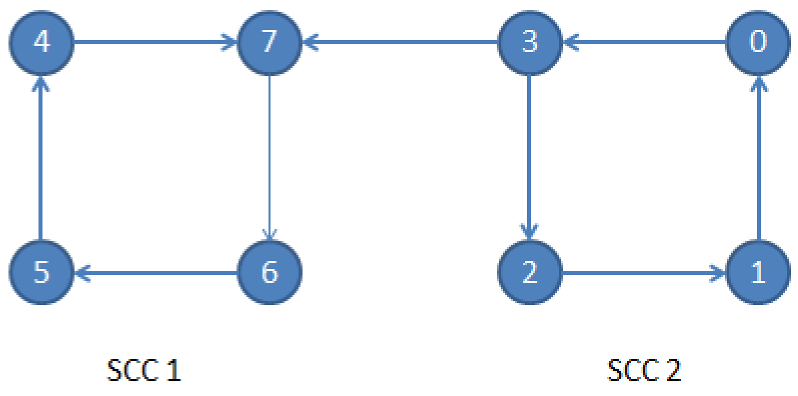
Graph G' (V, E')

The simplified view of Graph G is



This graph has two strongly connected components shown as above. The dotted edges are the edges which can be removed in the graph G' according to the problem statement.

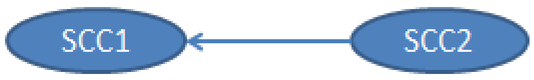
The simplified view of Graph G' is



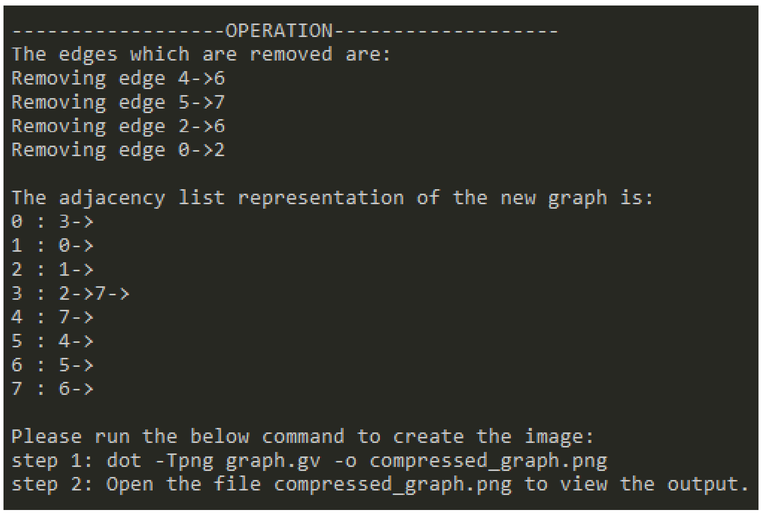
As can be seen, all the dotted edges have been removed in G'.
To find out which edges of G are to be excluded in G', we have to take care of following ideas:

1. Inside an SCC, exclude those edges whose removal will not create any new SCC. For example we can safely remove edge (0, 2), (4, 6) and (5, 7) as their removal will not create any new SCC. Suppose there is an edge (u, v). We can exclude it only if there is a path from u to v and there is a path from v to u without including the edge (u, v).
2. Remove the parallel edges connecting two different SCCs. For example edge (3, 7) and edge (2, 6) are parallel edges going from SCC2 to SCC1. Thus we can safely remove edge (2, 6) without disturbing the component graph. Suppose there is an edge (u, v) such that u and v belong to different components. We can exclude this edge only if there is already an edge (u', v') such that u and u' belong to same component and v and v' belong to same component. Thus (u, v) and (u', v') are parallel edges connecting one component to another.

The component graph in both the cases is same and can be visualised as follows:



Output:



```
------------------OPERATION-------------------
The edges which are removed are:
Removing edge 4->6
Removing edge 5->7
Removing edge 2->6
Removing edge 0->2

The adjacency list representation of the new graph is:
0 : 3->
1 : 0->
2 : 1->
3 : 2->7->
4 : 7->
5 : 4->
6 : 5->
7 : 6->

Please run the below command to create the image:
step 1: dot -Tpng graph.gv -o compressed_graph.png
step 2: Open the file compressed_graph.png to view the output.
```

The edges which are removed in the new Graph G' are shown. After that, the adjacency list representation of the new graph is shown. Also graph.gv file is produced as the output which can be converted to image file using the command:
dot -Tpng graph.gv -o compressed_graph.png
After this step, open the image file compressed_graph.png to view the output.

Functions used:

1. **Graph *compress_graph();**
2. **void remove_edges_from_components(Graph *, vector<vector<int>> *, int , int *);**
3. **Graph *clone_graph();**
4. **vector<vector<int>> *find_scc();**
5. **bool path_possible(Graph *, int, int, int, int);**
6. **void remove_edge(int, int);**

Implementation:

At first compress_graph method is called. This creates a clone of the existing graph using clone_graph method. Strongly connected components are found out. All the vertices are then classified on the basis of components; this information is stored in scc_of_nodes array. For each component we remove those edges from the new graph whose removal will not create any new SCC by calling remove_edges_from_components method. This method will also remove the parallel edges connecting two different components.

Inside remove_edges_from_components, we iterate over all components. For a single component, we check for all the edges (connecting two vertices of the same component) if by excluding them we do not lose a path from u to v or v to u. This is checked using path_possible method. If u to v and v to u paths are possible excluding the edge, we imply remove the edge by calling remove_edge method. Suppose the edge connects two different components, we check if already there is an edge connecting the two components in the same direction. If there is already an edge, we simply remove the edge under consideration as this is a case of a parallel edge.

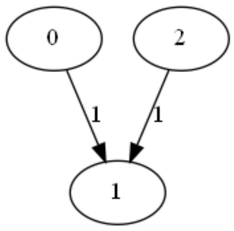
At the end, we return the newly created graph.

Semi connected Component

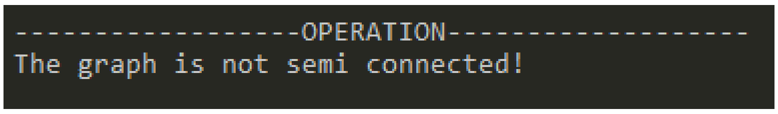
Q4. A directed graph G = (V, E) is semi connected if, for all pairs of vertices u, v in V, we have either a path u ---> v or a path v ---> u. Design an efficient algorithm to determine whether or not G is semi connected. Implement your algorithm and analyse its running time.

To check whether a graph is semi connected, we use option 8 in the menu. The output is a single line printed in the terminal which says whether the graph is semi connected or not.

Example 1:



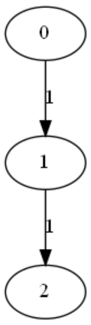
The output produced on the terminal for option 8 is:



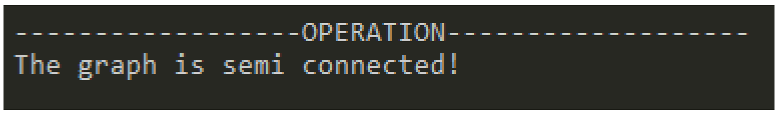
```
------------------OPERATION-------------------
The graph is not semi connected!
```

There is a path from 0 to 1 and 2 to 1. But there is no path between 0 and 2. Thus this graph is not semi connected.

Example 2:



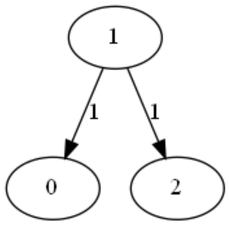
The output produced on the terminal for option 8 is:



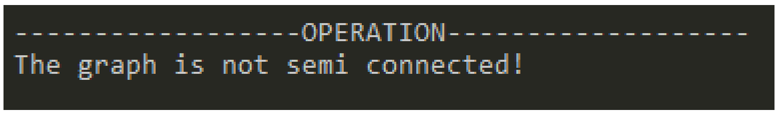
```
------------------OPERATION-------------------
The graph is semi connected!
```

There is a path from 0 to 1, 0 to 2 and 1 to 2. Thus this graph is semi connected.

Example 3:



The output produced on the terminal for option 8 is:



```
------------------OPERATION-------------------
The graph is not semi connected!
```

There is a path from 1 to 0 and 1 to 2. But there is no path between 0 and 2. Thus the graph is not semi connected.

Logic and Time Complexity Analysis:

Suppose we have a DAG and we have a topological ordering of it. It is semi connected if there is an edge (v_i , v_{i+1}) for each i. If there is no edge (v_i , v_{i+1}), then there cannot be a path from v_i to v_{i+1} or v_{i+1} to v_i as in a topological order, back edges are not present. Hence it cannot be a semi connected graph. If for every I there is an edge (v_i , v_{i+1}), then for each i, j (i<j) there is a path $v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_j$, and the graph is semi connected.

Suppose G has strongly connected components $C_1, C_2, \ldots , C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$ and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(i, j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$.

Having thus defined component graph, we state that component graph is a DAG. Thus we can apply the above concepts of DAG and semi connected graphs to find out if the graph is semi connected.

The Algorithm:

1. Find SCCs in the graph.
2. Build the component graph. Each node in the component graph corresponds to a component in the original graph and it represents the set of vertices belonging to the component.
3. Perform Topological Sort on the component graph and store the order of components.
4. Check if for every i, there is edge (v_i , v_{i+1}). If for any i, the edge (v_i , v_{i+1}) does not exist, declare the graph to be not semi connected. Otherwise it is semi connected.

Running Time Analysis:

Step 1 can be done using Tarjan's Algorithm for finding SCCs. This step takes O (V+E) time.

Step 2 can be done in O (V+E) time as we are just traversing all the edges of the original graph and adding the edge in the component graph if required.

Step 3 is Topological Sort which consists of a DFS Traversal and it takes O (V+E).

Step 4 takes O (V+E) because for each vertex v_i in component graph, we check if there is an edge to v_{i+1}. Since we iterate for each vertex of component graph and check for all the edges, the time can be at max O (V+E).

Since all the steps take O (V+E), we can hence conclude that the overall time complexity of this algorithm is O (V+E).

Functions Used:

1. **bool is_semi_connected();**
2. **vector<vector<int>> *find_scc();**
3. **int *topo_sort(Graph *graph);**
4. **void topoSortUtil(int, Graph *, stack<int> *, bool[]);**

Implementation:

The method **is_semi_connected** implements this problem. First Tarjan's SCC algorithm (via find_scc method) is used to find the list of all connected components. Then the component graph is created. There is an edge in the component graph if in the original graph there is an edge (u, v) such that u and v belong to different components. In this way all the edges of the original graph are traversed through and required edges are added to the component graph.

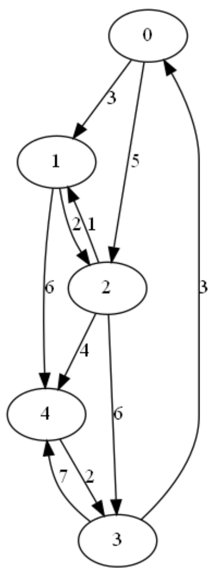
Then topological order of the component graph is found out using **topo_sort** method.

In the order found in topo sort, we iterate through all the vertices and check if there is an edge between two vertices v_i and v_{i+1}. If in any case we find that such an edge does not exist, we declare the graph to be not semi connected. Otherwise we declare it to be semi connected.

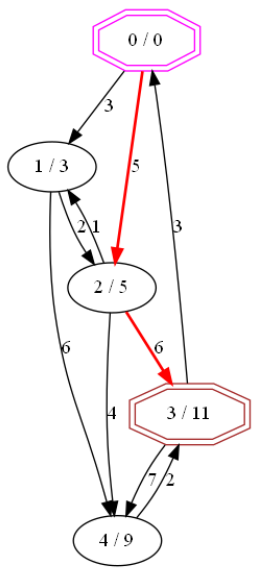
Dijsktra’s shortest path algorithm

Q5. Implement Dijkstra’s shortest path algorithm (in O (E log V) using a priority queue) to find a shortest cost path between a source vertex s to any target vertex t.

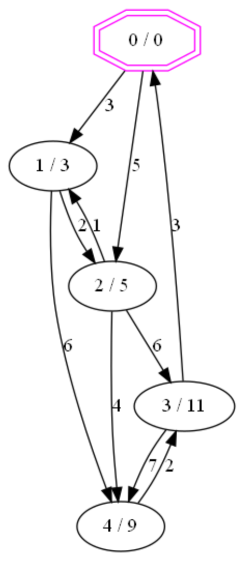
Use option 9 to compute shortest cost path from a source node to all other nodes or a particular node using Dijsktra’s algorithm.



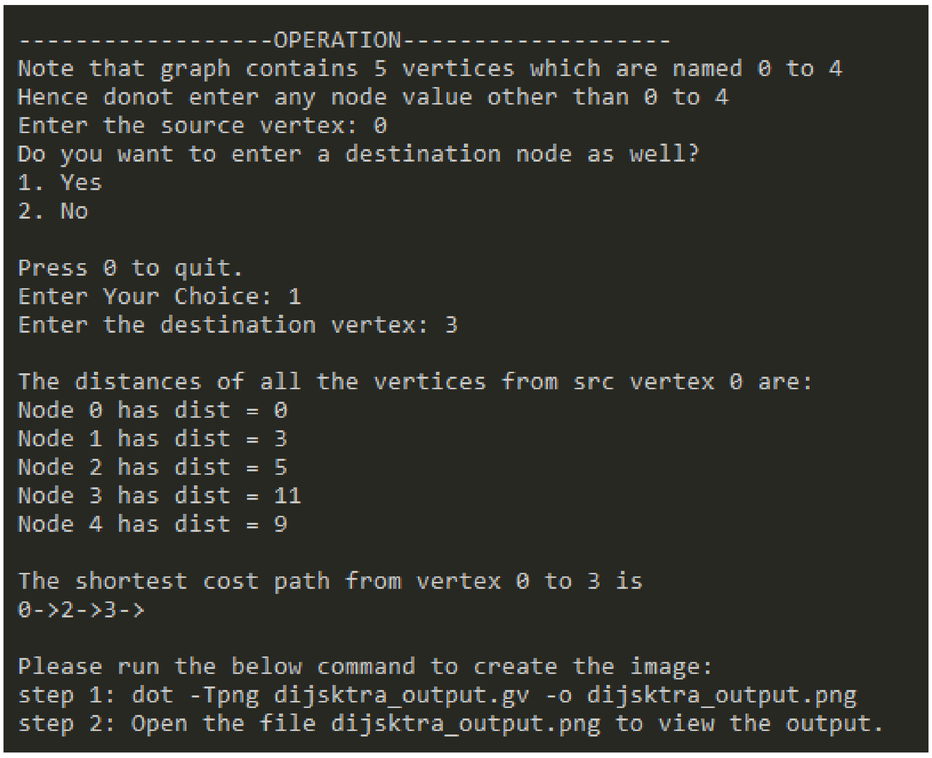
Normal Graph Output



Output produced by Dijkstra’s Algorithm with src = 0, dest = 3



Output produced by Dijkstra’s Algorithm with src = 0 and no dest



```
------------------OPERATION-------------------
Note that graph contains 5 vertices which are named 0 to 4
Hence donot enter any node value other than 0 to 4
Enter the source vertex: 0
Do you want to enter a destination node as well?
1. Yes
2. No

Press 0 to quit.
Enter Your Choice: 1
Enter the destination vertex: 3

The distances of all the vertices from src vertex 0 are:
Node 0 has dist = 0
Node 1 has dist = 3
Node 2 has dist = 5
Node 3 has dist = 11
Node 4 has dist = 9

The shortest cost path from vertex 0 to 3 is
0->2->3->

Please run the below command to create the image:
step 1: dot -Tpng dijsktra_output.gv -o dijsktra_output.png
step 2: Open the file dijsktra_output.png to view the output.
```

When we select option 9, we have to provide the source vertex. After that we have to choose whether or not we want to provide a destination vertex. If we do not provide a destination vertex, shortest distance is calculated for all the nodes as usual and is written inside the node in the graph output. However if we choose to provide a destination vertex, apart from the work shortest distance calculation for all the vertices, the shortest cost path from source vertex to destination vertex is also displayed using a red line. Note that source vertex is represented using a double octagon coloured red. The destination vertex is represented using a double octagon coloured brown.

3/11 inside the node means that 3 is the node's name and 11 is the shortest path from source vertex. Labels present on the edges represent the edge weights.

The terminal also displays the distance of all vertices from source vertex. The source vertex always has a distance of 0.

Functions used:

1. **void dijsktras_shortest_path_algorithm(int);**
2. **void dijsktras_shortest_path_algorithm(int, int);**
3. **void output_dijsktra(int[], int[], int, int);**

min_heap class is used to maintain the min heap data structure.

1. **min_heap(int V); //constructor**
2. **int parent(int);**
3. **pair<int, int> pop(); // takes O(log V) time**
4. **void push(pair<int,int>); // takes O(log V) time**
5. **void decrease_key(int, int); // takes O(log V) time**
6. **void swap(int, int);**
7. **void min_heapify(int); // takes O(log V) time**

Implementation:

First we create a min heap. The min heap is used to store the vertices for which the shortest distance has not been found out yet. So initially the min heap will contain all the vertices present in the graph. The distance value of a vertex is also stored which is the key value used to do the comparisons between the vertices in a heap. Initially all vertices have INT_MAX value as the key except the source vertex which has 0 value as the key.

The **dist** array stores the distance of each vertex from source vertex. Initially all vertices except the source vertex have INT_MAX stored in **dist** array. Source vertex has value 0 stored in **dist** array. Value of **parent**[v] for a vertex v stores parent vertex of v in shortest path. Initially push the source vertex to the min heap.

Pop an element from the min heap till the heap is not empty. After popping the element, we try to relax all out going edges from the vertex. If for an edge (u, v), the sum of distance of u and weight of the edge is less than distance of v, we relax the edge, there by performing a decrease key operation on vertex v present in the heap. Note that if minimum distance to a particular vertex v has been finalised, we do not relax an edge pointing towards v. We also update the **dist** array and the **parent** after relaxation.

To perform the decrease key operation in log v time, we follow an approach. We have the name of the node for which we want to decrease the distance value (or key value in a min heap). We need to find the location of the node present in the heap in O (1) time. This can be done using an array called **location_of_node** present inside the **min_heap** class. Thus we the location of the node for which we want to update the key value. We then decrease the key value of the vertex and keep on swapping with the parent till we find a parent having lower key value than the vertex under consideration. This step can take place height number of times in a heap which is O (log v). Thus this decrease key step takes O (log v) time.

At the end, there will not be any vertex present in the min heap. This is the time we quit the algorithm. The output_dijsktra method is then called to produce the output.

Time Complexity Analysis:

Since the purpose of the algorithm is to relax all the edges, the loops run for O (E + V) time. The inner loop has decrease key operation which takes O (log V) time. So overall time complexity is

O ((E + V) x log V) = O (E log V)

In the beginning, we inserted V elements into heap. Each insert took no more than O(log V) time. Thus for all insert operations, time taken is O (V log V).

Thus final time complexity is

O (E log V) + O (V log V) = **O (E log V)**

<u>Output:</u>

The shortest distances of all the nodes from source vertex are displayed on the terminal. If a destination node was provided, the path is also printed. Apart from that dijsktra_output.gv is also produced which must be converted to an image file manually by running the following command in another terminal:

Please run the below command to create the image:
step 1: dot -Tpng dijsktra_output.gv -o dijsktra_output.png
step 2: Open the file dijsktra_output.png to view the output.