Data Structures Lab

Assignment - 04

Sanjeev Kumar (214101048)

Graph Data Structure Implementation

Introduction

- A graph is a group of vertices and edges that are used to connect these vertices. G = (V,E) , where V is the set of nodes and E is the set of edges.

- It is a non-linear data structure .

- Here the graph has been stored as an adjacency list and an array of size equal to the number of nodes in the graph is used to store the head of the linked list of adjacent edges for each node. Here the graphs are weighted and directed.

Implementation Overview

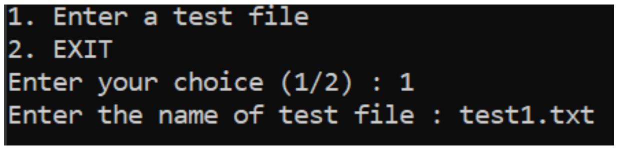
- Two classes have been used here - one class for nodes of linked list which is the list of edges outgoing from a node and the other class for Graph which contains a vector of pointers to list of edges.

- Class definition for two classes is as follows -

```
class Edges
{
        int id;
        int weight;
        char type_of_edge;
        Edges *next_edge;
public:
Edges(int node_id , int node_weight);
friend class Graph;
};
```

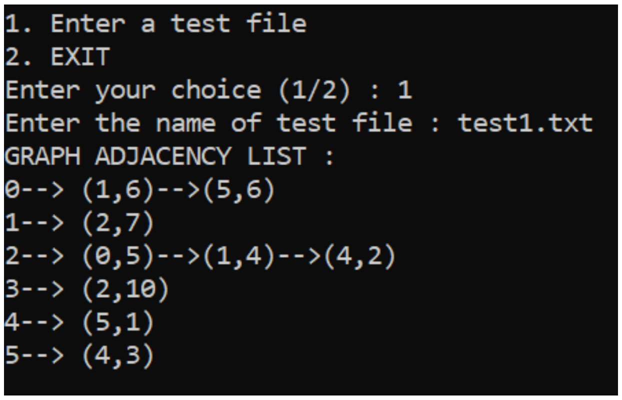
```
class Graph
{
        int nodes;
        vector<Edges*> edgeList;
        int num_scc;
        vector<vector<int> > strongly_connected_components;
        vector<int> component_indegree;
        vector<int> start_time;
        vector<int> end_time;

        //Member functions
        public:
        Graph(int n);
        void CreateEdge(int u , int v , int w);
        void PrintAdjacencyList();
        void DFS();
        void NodeVisitDFS(int v,vector<bool> &visited,vector<int> &d,vector<int> &f,vector<int> &pi,int
&time ,vector<int> &dfs_order);
        void findEdgeType(vector<int> &d , vector<int> &f , vector<int> &pi);
        void PrintEdgeClassification();
        void PrintTime();
        void GraphPrint(const char* file);
        void NodePrint(int v , FILE* f , vector<bool> &printed);
        void FindSCC();
        void TraverseSCC(int i,vector<bool> &visited,vector<bool> &present_in_stack,stack<int>
&scc_stack,vector<int> &low,int &disc,vector<int> &dv);
        void PrintSCC();
        void AssignComponentsToNodes();
        void Dijkstra(int source_node);
        void PrintDistance(vector<int> distance , int source);
        void ComponentGraph();
        bool CheckSemiConnected();
        };
```

- On running the program, the following menu appears on the screen. Enter the file name with .txt extension.

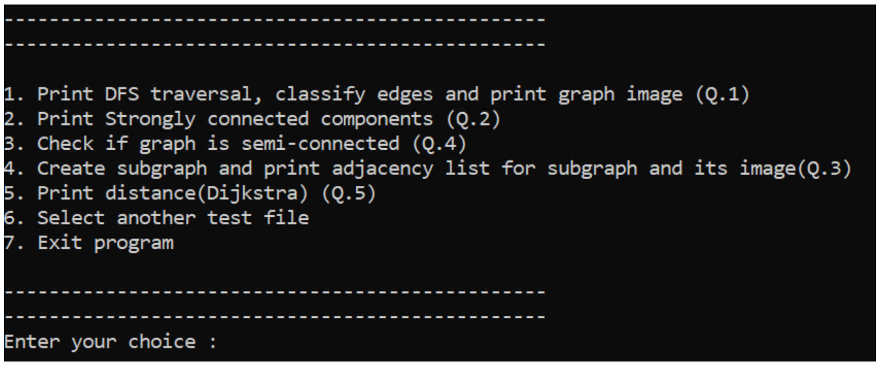


```
1. Enter a test file
2. EXIT
Enter your choice (1/2) : 1
Enter the name of test file : test1.txt
```

- Then after entering the name of the test file, the program reads data from the file and creates a graph.It then prints the adjacency list on the output screen and then a menu appears on the screen with a list of operations to choose from.



```
1. Enter a test file
2. EXIT
Enter your choice (1/2) : 1
Enter the name of test file : test1.txt
GRAPH ADJACENCY LIST :
0--> (1,6)-->(5,6)
1--> (2,7)
2--> (0,5)-->(1,4)-->(4,2)
3--> (2,10)
4--> (5,1)
5--> (4,3)
```



```
-----------------------------------------------
-----------------------------------------------

1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

-----------------------------------------------
-----------------------------------------------
Enter your choice :
```

- Enter the operation of your choice (a number from between 1 and 7, both inclusive).

<u>Operation Implementation</u>

1. DFS traversal
2. TARJAN’s algorithm
3. Checking Semi-connectedness
4. Subgraph with same strongly connected components
5. Dijkstra’s single source shortest path

<u>DFS traversal</u>

- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and marks the vertices as visited until no unvisited node is found and then backtracks.

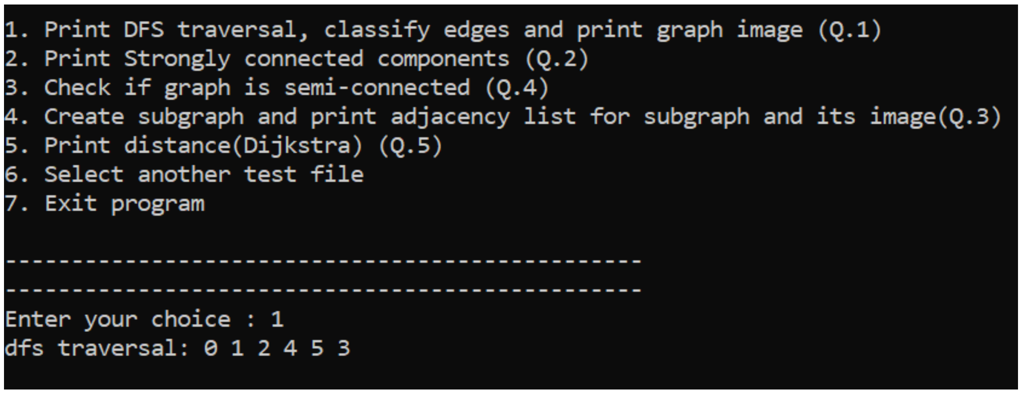
- While visiting the nodes, the start time and the finish time of dfs of the node and the parent node of each node(from which that the dfs for a particular nodes was called) is are also stored in vectors named as start_time and end_time and pi with their sizes equal to the number of nodes in the graph.

- These are used to find the type of edges by comparing the start and end time of the source and destination nodes of an edge.

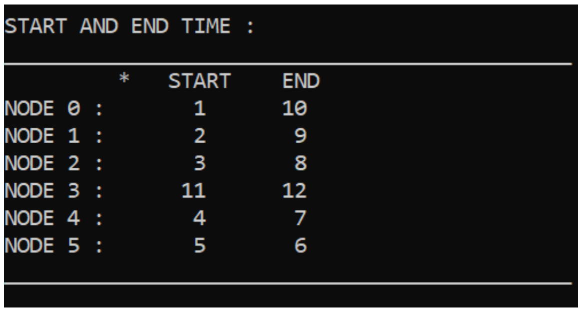
- For an edge u➡v , if **pi[v] = u** then it is a tree edge
- Else if **start_time[u]<start_time[v]<end_time[v]<end_time[u]** then it's a forward edge.
- For back edge, **start_time[v]<start_time[u]<end_time[u]<end_time[v].**
- For cross edge ,**start_time[v]<end_time[v]<start_time[u]<end_time[u]**

- Time Complexity of DFS traversal - since an adjacency list is being used here to store the graph , the time complexity is **O(V+E)**.

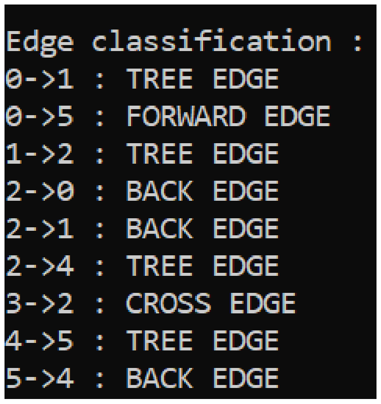
- Sample output of program for dfs traversal -
  After selecting 1, the program prints the dfs traversal, start and finish dfs time and classifies edges on the output screen. Then it asks the user to input a name for the image file and prints the graph image using graphviz.



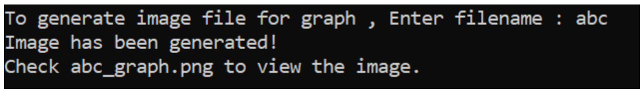
```
1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

------------------------------------------------
------------------------------------------------
Enter your choice : 1
dfs traversal: 0 1 2 4 5 3
```



```
START AND END TIME :
__________________________________________
        *   START    END
NODE 0 :       1      10
NODE 1 :       2       9
NODE 2 :       3       8
NODE 3 :      11      12
NODE 4 :       4       7
NODE 5 :       5       6
__________________________________________
```

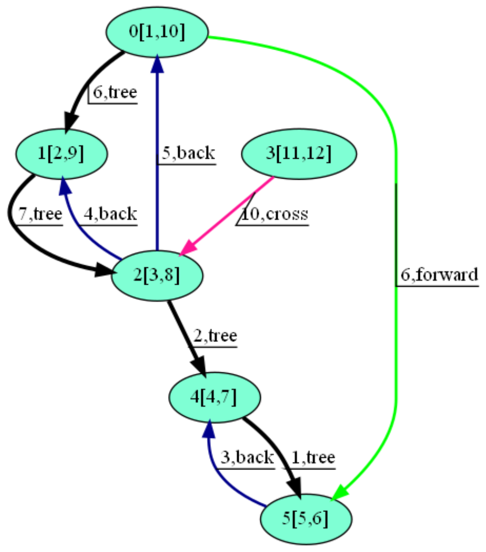


```
Edge classification :
0->1 : TREE EDGE
0->5 : FORWARD EDGE
1->2 : TREE EDGE
2->0 : BACK EDGE
2->1 : BACK EDGE
2->4 : TREE EDGE
3->2 : CROSS EDGE
4->5 : TREE EDGE
5->4 : BACK EDGE
```



```
To generate image file for graph , Enter filename : abc
Image has been generated!
Check abc_graph.png to view the image.
```

Generated image file where the nodes are labelled with node id ,start time and finish time (eg. 0[1,10] where 0 is node id, 1 is start time and 10 is finish time) and edges are labelled with edge weight and edge type(tree,cross,back,forward) and are colored with different colors based on their types-



Tarjan's Algorithm

- A strongly connected component of a directed graph is the maximal subgraph in which there is a path between every pair of vertices. For example, for every pair of vertices (u,v) there is a path from u to v and v to u.

- Tarjan's algorithm is used to compute all the SCCs in a graph efficiently .

- In this algorithm, dfs traversal is performed on the graph and low values for each node is calculated .A stack is used to store the visited nodes of the graph.

- Low value of a node tells the topmost reachable ancestor with minimum possible disc value via the subtree of that node.

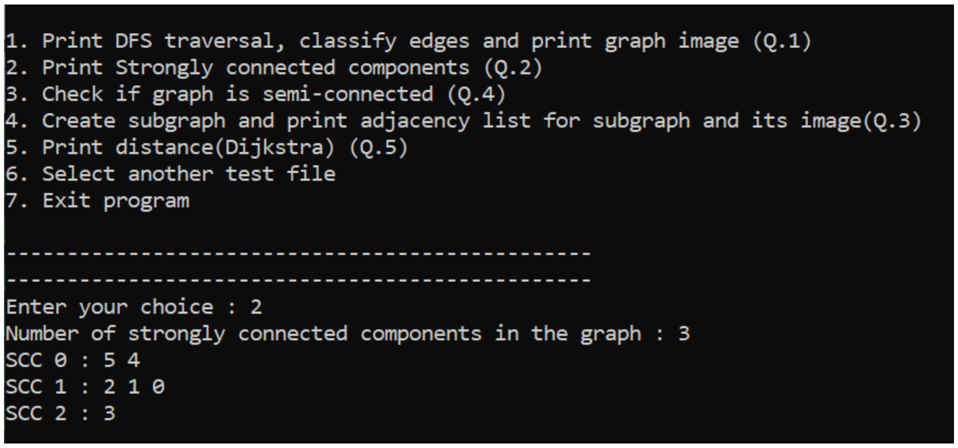
- low(v) is minimum of
    - disc[v]
    - The lowest disc[u] among all back edges (v,u)
    - The lowest low(u) among all tree edges (v,u).

- Here disc[v] is the discovery of time of node v which simply means that the time at which dfs traversal on node v is performed.

- For a node v, if its low value and disc value are same then node v is the head of a strongly connected component and we start popping out elements from the stack until node v is found. All these edges together form a strongly connected component of the graph.

- Time complexity - dfs traversal takes O(V+E) and popping nodes from stack and storing in a separate vector takes O(V). So total time complexity is **O(V+E)**.

- Sample input output for tarjan's algo-



```
1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

-----------------------------------------------
-----------------------------------------------
Enter your choice : 2
Number of strongly connected components in the graph : 3
SCC 0 : 5 4
SCC 1 : 2 1 0
SCC 2 : 3
```

<u>Checking semi connectedness of a graph</u>

- A graph is semi connected if for every pair of nodes u,v in V there is either a path from u to v or a path from v to u or both. So a strongly connected graph is also semi connected.

- Approach - the component graph is created by taking one scc as a node of the component graph and then it is stored as an adjacency list.

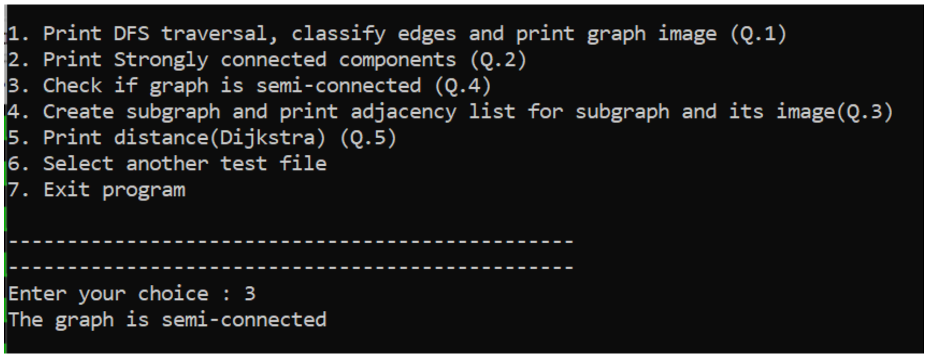
- The resulting component graph is a directed acyclic graph(DAG) and a topological ordering exists between the different components of the graph.

- The components are topologically sorted by choosing a vertex with 0 in-degree and removing it from the graph and then deleting all outgoing edges and continue this process until all the SCCs are topologically ordered.A priority queue is used to extract the component with 0 indegree.

- After getting the topological ordering of the components, check if there is an edge in the component graph between every two components adjacent in topological order. If edges exist then the graph is semi-connected else it is not.

- Time complexity analysis :
    - Component graph is stored as an adjacency list so creating and traversing it takes O(V+E) time.
    - Extracting components with 0 in-degree is done using a priority queue so it takes O(logV) time.
    - So total time complexity is **O(V+E)**.

Sample input output for checking semi-connectivity -



```
1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

------------------------------------------------
------------------------------------------------
Enter your choice : 3
The graph is semi-connected
```

<u>Creating sub graph with same SCCs and component graph as G</u>

- Approach - The parallel edges between two edges can be removed without altering the semi connected components of the graph and component graph.

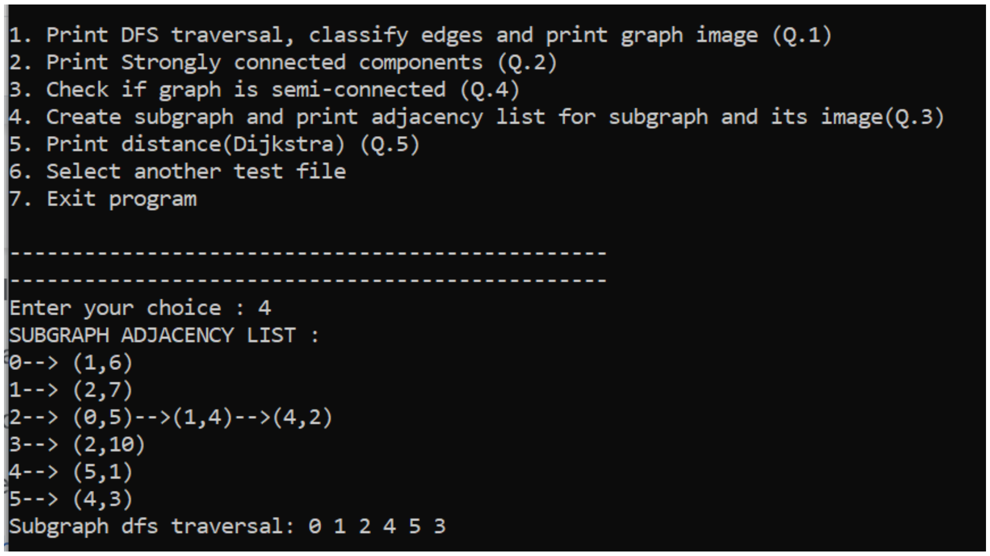
- The component graph created in the previous function is used here also. Traverse the edges of graph G and check in what components the source and destination vertex of the edge belong to.

- If the nodes belong to the same component then add the edge in the subgraph.

- If the nodes belong to different components then check the count of edges between those two components. If the count is 1 then add the edge and if the count is more than 1 then skip that edge and decrease the value of count by 1.

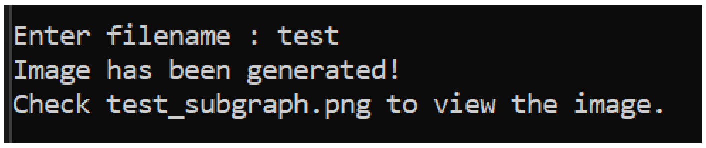
- Time complexity : traversing the adjacency list of the graph takes O(V+E). Traversing the component graph takes O(V+E) in the worst case. So total time complexity is O(V+E).

- Sample input output for creating subgraph -



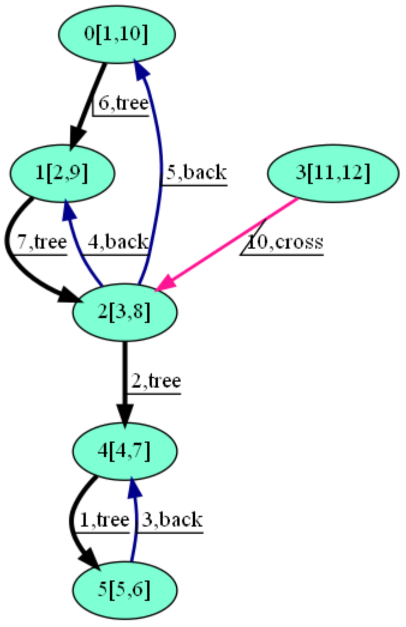
```
1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

-----------------------------------------------
-----------------------------------------------
Enter your choice : 4
SUBGRAPH ADJACENCY LIST :
0--> (1,6)
1--> (2,7)
2--> (0,5)-->(1,4)-->(4,2)
3--> (2,10)
4--> (5,1)
5--> (4,3)
Subgraph dfs traversal: 0 1 2 4 5 3
```

On selecting option 4 , the subgraph is created and its adjacency list and dfs is printed on the output screen. It then asks the user to enter a file name to print the image of the subgraph using graphviz.



```
Enter filename : test
Image has been generated!
Check test_subgraph.png to view the image.
```

Generated image file for subgraph -



<u>**Dijkstra’s single source shortest path algorithm**</u>

- Dijkstra’s single source shortest path algorithm is a greedy algorithm used for finding the shortest distance from source node to all the other nodes.

- A vector of size equal to the number of nodes in the graph is created. All its entries are initially filled with infinity.

- A source node is taken as input from the user which is passed as an argument to the Dijkstra function. The distance of the source node is then updated to zero.

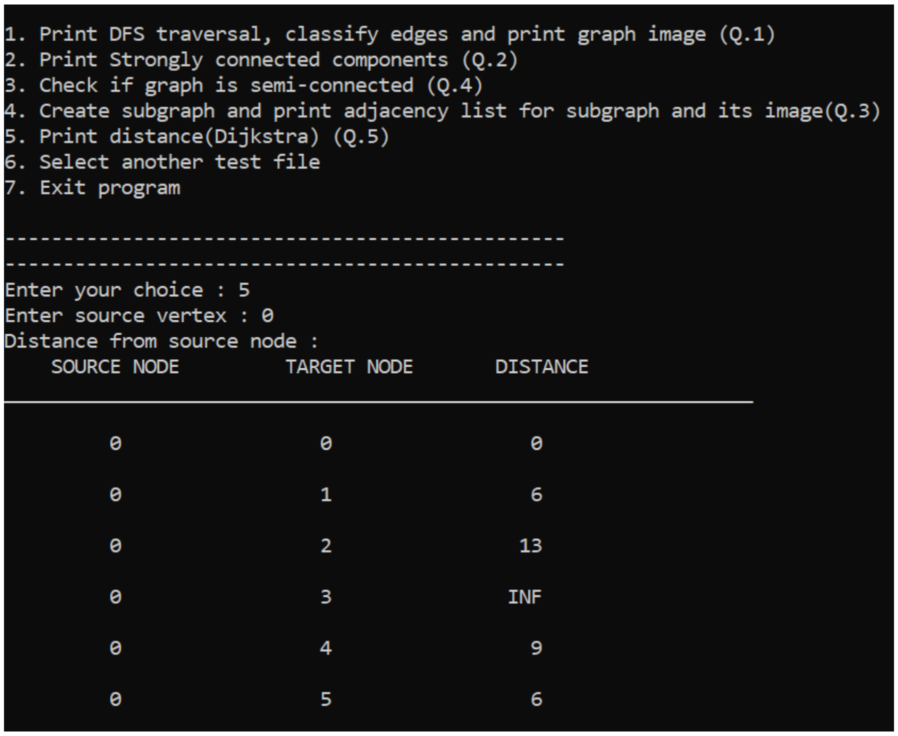
- A priority queue is initialized in which a pair of discovered nodes and their calculated distance is stored as a pair(eg. <distance , node_id>).

- The source node is inserted into the empty priority queue with distance zero.

- The elements from the priority queue are popped out one by one until the priority queue is empty. The element popped out is the one with minimum distance.Let this node be v. Then the distance of nodes adjacent to node v are calculated and updated if the new distance is less than the previously calculated distance and then it is inserted into the priority queue.

- Time complexity : O(ElogV)

Sample input output for dijkstra's algorithm implementation.



```
1. Print DFS traversal, classify edges and print graph image (Q.1)
2. Print Strongly connected components (Q.2)
3. Check if graph is semi-connected (Q.4)
4. Create subgraph and print adjacency list for subgraph and its image(Q.3)
5. Print distance(Dijkstra) (Q.5)
6. Select another test file
7. Exit program

------------------------------------------------
------------------------------------------------
Enter your choice : 5
Enter source vertex : 0
Distance from source node :
    SOURCE NODE         TARGET NODE       DISTANCE
________________________________________________________________

        0                 0                 0

        0                 1                 6

        0                 2                13

        0                 3               INF

        0                 4                 9

        0                 5                 6
```