CS 513

DS Lab

Assignment 4

Graph

Abhijeet Padhy

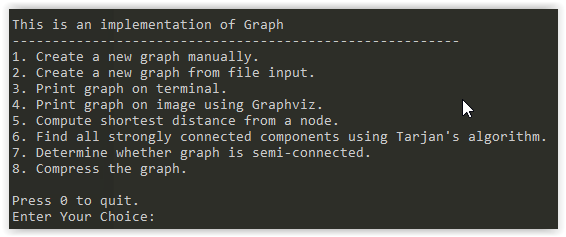
Roll: 214101001

# Index

1. Basic Information
2. Class Implementation
3. DFS Traversal and Classification of edges
4. Strongly connected components using Tarjan’s Algorithm
5. Removal of Edges from a graph
6. Semi connected Component
7. Dijsktra’s shortest path algorithm

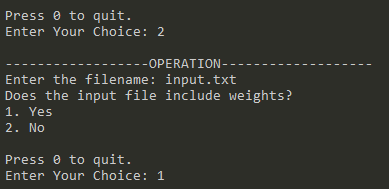
# Basic Information

The program starts with the following menu:

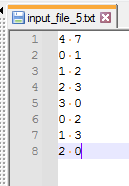


Option 1: Use this to enter the edges of the graph manually.

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



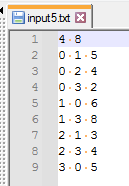
Enter the name of the input file. It then asks if the input file also contains weights. Press 1 if input file contains weights. Press 0 if input file does not contain weights. The format of the input file is shown below:



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:

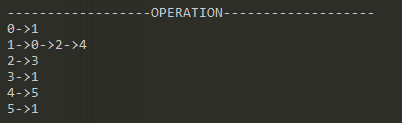


Input file with weights.

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:



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.

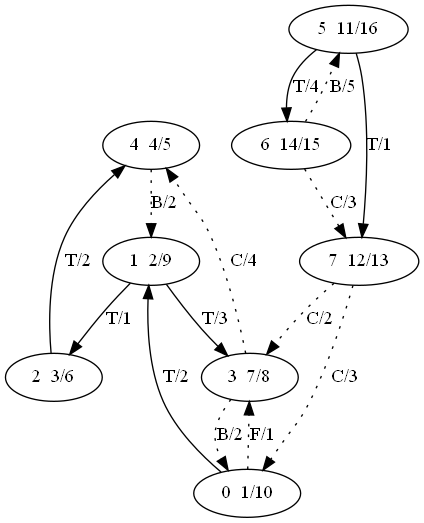
Options 4 to 6 will be explained in detailed in the following sections.

**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.**

# 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 4 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 |

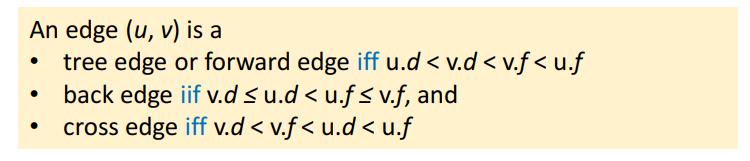
The vertex from which we start DFS Traversal will have start time as 1. All the tree edges are denoted by solid edges. Thus we can understand the DFS Tree just by following the solid tree edges.

Function Prototypes:

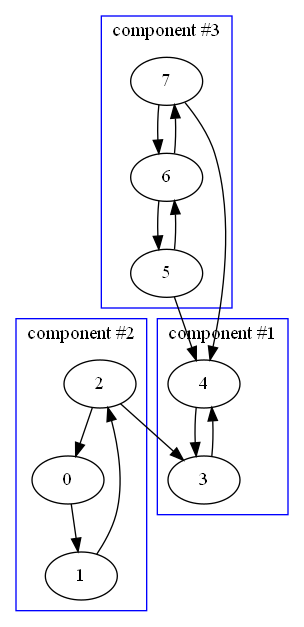
**void dfs\_traversal(bool[], int[], int[], int, FILE \*);**

**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. 

# 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:

**vector<vector<int>> \*find\_scc();**

**void find\_scc(int, bool[], int[], bool[], stack<int>\*, int[], vector<vector<int>>\*);**

**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
   1. Update visited and DFS number. Set low value as dfs number when it is visited for the first time. Add the node to stack.
   2. For all the vertices v adjacent to the current vertex u:
      1. 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.
      2. If it is a back edge, update low value of u as the minimum of low value of u and dfs number of v.
   3. 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.