DS LAB CS513: Assignment 04 (GRAPH)

Vijay Purohit | 214101058

M.Tech CSE, IITG

Contents

List of Figures

1 FILES AND FUNCTIONS DESCRIPTION

1.1 main.cpp

This file contains the GRAPH Structure and definition of functions for GRAPH Class "DiGraph". It contains the functions for operations required in the assignment. V is the Number of the Vertex, E is the number of the edges. Vertex Value is a sequence from 0 to MaxVertex-1.

1.1.1 void addDirectedEdge(int u, int v, int wt)

This function add the directed edge (u,v) with weight (wt) in the Graph Adjacency List.

1.1.2 void showDiGraphAdjList()

This function is used to show the adjacency list of the graph on the console screen.

1.1.3 void printDiGraph(const string& file_name)

This Function is used to print the original DiGraph in the 'png' format using GraphViz Command.

1.1.4 void DijkstraSSSPAlgo(int src)

This function calculates the shorest path to any target vertex from the source vertex using Dijkstra Single Source Shortest Path Algorithm. Source Vertex is given by src.

1.1.5 void showDiGraphDijkstraValues(int src)

This function prints the each vertex shortest distance and its parent calculated using Dijkstra SSSP Algorithm on the console screen.

1.1.6 void printGraphVizDijkstra(const string& file_name)

This function print the Shortest Path from source to each vertex and node distance from its source using the Graphviz command. Output is generated with appropriate name with 'png' format in the output folder.

1.1.7 void DFSDiGraph(int src)

This is DFS function which performs the Depth First Traversal of the input graph. It calculates the start time, end time, low number of the node. It also calculates the number of connected component and its node. All in one single DFS pass.

1.1.8 void DFSRec(int s, vector<bool> &dfs_rec, stack<int> &st)

This is DFS Helper Recursive Function which performs the DFS recursion. s is the source vertex provided. dfs_rec is the array which keep track of the node in the stack. st is the stack used for maintaining connected components.

1.1.9 void printGraphDFSTraversal(const string& file_name)

This prints the DFS Graph with each node start time, end time. It also classify Edge in 4 category - Tree Edge, Forward Edge, Cross Edge and Back Edge. This function does a level order traversal of the Graph and generate two files according to file name provided. One is '.gv' file for graph-viz to read and another final 'png' output file of the tree. It will automatically execute the graph-viz command to generate the image file.

1.1.10 void printGraphSCCTarjan(const string& file_name)

This prints the Components Graph using Graphviz command in the output folder. Components are indicated using box. It uses the components generated during DFS traversal of the graph.

1.1.11 bool CheckSemiConnectedDiGraph()

This function check the semi-connectedness of the graph. It first make a condensed graph of the original graph then apply a topological sort in the condensed graph. If a linear chain of nodes in condensed graph can be created then it is semi-connected.

1.1.12 void ReduceMainSCCEdges()

This function is used to reduce the strongly connected component graph by removing the redundant edges within scc and between scc. It build the minimal graph by adding necessary edges only.

1.1.13 void SCCMinimalDFSHelper(int s, unordered_map<int, bool> &vis, unordered_map<int, unordered_map<int,int>> &SCCedgeAdded)

This is DFS helper function to check the reach ability of the node with other nodes using Depth First Traversal. s is the source node. vis is the node map to check if node is visited or not. SCCedgeAdded is the the map to check if node is already added in the minimal graph or not.

1.1.14 void printGraphMinimalSCC(const string& file_name)

This prints the Components Graph with minimum edges using graphviz command in 'png' format.

1.1.15 void ReadGraphFromFile(const string& ip_file_name, int firstTime)

This function is used to read inputs of the graph from the respective input text file. First line is Num_Of_Vertex Num_of_Edges. Each Next line is Edge (int): u v wt. Make sure VERTEX VALUE start from 0 to Num_Of_Vertex-1

2 GRAPH OPERATIONS LOGIC

Graph Data Structure store the information in the nodes and relationship with the nodes using the edges. Edges can be directed (one way) and undirected (two way).

2.1 Print Function Logic

We print the graph using package graphviz. Initially we traverse the tree in Level order manner (BFS Traversal) and store the node information, its children in the file with extension '.gv' according to graphviz format. Then program execute the graphviz command to generate the respective '.png' file. In a program, we provide the timestamp suffix in order to distinguish our different operations and generate different '.png' file without overwriting.
For each Operation of the Graph in the Assignment, we have modified the program to suitably print the relevant information of the Node. For Example, Start time, End Time and Edge Type for DFS. Distance and Parent information in Dijkstra. Component marking in Tarjan algorithm.

2.2 Depth First Search

(Function Reference: void DFSRec(int s, vector<bool> &dfs_rec, stack<int> &st), void DFSDiGraph(int src))
Depth First Search is a very useful technique for analysing graphs.
Let $G = (V, E)$ be a directed graph, where V is the vertex set and E is the edge set. We assume the graph is represented as an adjacency structure, that is, for every vertex v there is a set $adj(v)$ which is the set of vertices reachable by following one edge out of v. To do a depth first search we keep two pieces of information associated with each vertex v. One is the depth first search numbering $num(v)$, and the other is $mark(v)$, which indicates that v is currently on the recursion stack.

```
$i \leftarrow 0$
for all $v \in V$ do $num(v) \leftarrow 0$
end for
for all $v \in V$ do $mark(v) \leftarrow 0$
```

```
end for
for all $v \in V$ do
    $num(v) = 0$ then $DFS(v)$
end for
procedure DFS($v$)
    $i \leftarrow i + 1$
    $num(v) \leftarrow i$
    $mark(v) \leftarrow 1$
    for all $w \in adj(v)$ do
        if $num(w) = 0$ then $DFS(w)$                    ▷ [$(v, w)$ is a tree edge]
        else if $num(w) > mark(v)$ then                  ▷ [$(v, w)$ is a forward edge]
        else if $mark(w) = 0$ then                       ▷ [$(v, w)$ is a cross edge]
        else                                             ▷ [$(v, w)$ is a back edge]
        end if
    end for
    $mark(v) \leftarrow 0$
end procedure
```

If the graph has n vertices and m edges then depth first search can be used to solve all of these problems in time $O(n + m)$, that is, linear in the size of the graph.
This classification of the edges is not a property of the graph alone. It also depends on the ordering of the vertices in $adj(v)$ and on the ordering of the vertices in the loop that calls the DFS procedure.

- tree edges form a forest of trees.
- forward edges are from a vertex to a descendant in the tree.
- cross edges are from node a to node b where the subtrees rooted at a and b are disjoint.
- back edges are from a vertex to an ancestor.

2.3 Tarjan's Algorithm

(Function Reference: void DFSRec(int s, vector<bool> &dfs_rec, stack<int> &st), void printGraphSCCTarjan(const string& file_name))
Two vertices v and w of a graph G are equivalent, denoted $v \equiv w$, iff there exists a path from v to w in G, and there exists a path from w to v in G. This equivalence relation induces a partitioning of the vertices of the graph into components in which each pair of vertices is a component is equivalent. These are called the strongly connected components (or sometimes just strong components) of the graph. A graph is said to be strongly connected if it has one strongly connected component.
$lowlink(v)$ = *The lowest numbered vertex in the same strong component as v reachable from v using zero or more tree edges followed by at most one back or cross edge.*
Consider the vertex numbering produced by running DFS on a graph G. Each strongly connected component (SCC) of G has some lowest numbered vertex in it. Call that the **base** of the SCC. A vertex is a base if and only if $num(v) = lowlink(v)$.
Tarjan's SCC Algorithm:
Do a complete DFS as explained in Depth First Search. Maintain a stack of vertices. When we first visit a vertex v compute $num(v)$ in the usual way, and assign $lowlink(v)$ to be $num(v)$. We also put v on the stack. Then we call DFS recursively on w for each tree edge (v, w). This will compute $lowlink(w)$. Upon the return of $DFS(w)$ we udpate $lowlink(v)$ as follows:

$$lowlink(v) \leftarrow min(lowlink(v), lowlink(w)).$$

If (v, w) is a cross edge or back edge, and w is on the stack we do the assignment:

$$lowlink(v) \leftarrow min(lowlink(v), num(w)).$$

When we complete the $DFS(v)$ function we test if $lowlink(v) = num(v)$. If so, v is a base vertex, so we bulk pop vertices off the stack until v is popped off, then we stop popping. Each such group of popped vertices is a **SCC**.

2.4 Semi-Connected Graph

(Function Reference: bool CheckSemiConnectedDiGraph()
A directed graph $G = (V, E)$ is said to be **semi-connected** if, for all pairs of vertices $u,\ v \in V$ we have $u \to v$ or $v \to u$ path.
A DAG is semi-connected in a topological sort, for each i, there there is an edge (v_i, v_{i+1}).
Given a DAG with topological sort $v_1, v_2, ..., v_n$: If there is no edge (v_i, v_{i+1}) for some i, then there is also no path (v_{i+1}, v_i) (because it's a topological sort of a DAG), and the graph is not semi connected.
If for every i there is an edge (v_i, v_{i+1}), then for each i, j $(i < j)$ there is a path $v_i \to v_{i+1} \to \cdots \to v_{j-1} \to v_j$, and the graph is semi-connected.
Procedure:

- We find Maximal SCCs in the graph.
- Then we build a condensed graph of the SCC $G' = (U, E')$ such that U is a set of SCCs. $E' = (V_1, V_2)|$ there is v_1 in V_1 and v_2 in V_2 such that $(v_1, v_2) \in E)$
- Do Topological Sort on G'
- Check if for every i, there is edge V_i, V_{i+1}

Time complexity for finding SCC using Tarjan's algorithm is $O(V + E)$, for building a DAG is $O(V + E)$, for Topological Sort $O(V + E)$ and $O(V)$ for checking semi-connected property of the graph. Overall it is $4(V + E)$ i.e. $O(V + E)$.

2.5 Dijkstra Algorithm

(Function Reference: void DijkstraSSSPAlgo(int src)
Given a graph and a source vertex in graph, *Dijkstra Algorithm find shortest paths from the source to all vertices in the given graph.* For Dijkstra's algorithm, it is always recommended to use heap (or priority queue) as the required operations (extract minimum and decrease key) match with speciality of heap (or priority queue). However, the problem is, priority_queue doesn't support decrease key. To resolve this problem, do not update a key, but insert one more copy of it. So we allow multiple instances of same vertex in priority queue. This approach doesn't require decrease key operation and has below important properties. Whenever distance of a vertex is reduced, we add one more instance of vertex in priority_queue. Even if there are multiple instances, we only consider the instance with minimum distance and ignore other instances.
Procedure:

1. Initialize distances of all vertices as infinite.
2. Create an empty priority_queue *pq*. Every item of *pq* is a pair $(weight, vertex)$. Weight (or distance) is used used as first item of pair as first item is by default used to compare two pairs.
3. Insert source vertex into *pq* and make its distance as 0.
4. While either *pq* doesn't become empty.
   (a) Extract minimum distance vertex from *pq*. Let the extracted vertex be u.
   (b) Loop through all adjacent of u and do following for every vertex v.
   If $dist[v] > dist[u] + weight(u, v)$
      i. Update distance of v, i.e., do $dist[v] = dist[u] + weight(u, v)$
      ii. Insert v into the pq (Even if v is already there)

Time complexity remains $O(E \log V)$ as there will be at most $O(E)$ vertices in priority queue and $O(\log E)$ is same as $O(\log V)$.

2.6 Minimal Strongly Connected Graph

(Function Reference: void ReduceMainSCCEdges()
Minimal SCC graph is a graph obtained from the original graph which has same component graph and same strongly connected components but minimum possible edges. For performing these operations, first we run a DFS using vector to get the edge classifications and node low value. Then we use Tarjan's Algorithm to find all the strongly connected components. We divide the further process into two steps.

1. Adding the necessary edges between two different SCC.
2. Adding the necessary edges within SCC.

2.6.1 Adding the necessary edges between two different SCC.

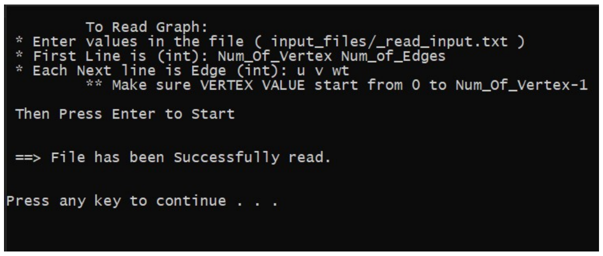
If there are multiple edges from one strongly connected component to another, then we consider only one out of them.

2.6.2 Adding the necessary edges within SCC

- Adding all the tree edges.
- We do not add the forward edge as we can reach the same destination using the tree edges using the forward edge.
- If there are multiple cross edges from a single source vertex to different destinations, we consider only one cross edge that reaches the destination with lowest starting time.
- If removing the edge we can still reach the destination node then we will remove it, else we will not remove the back edge

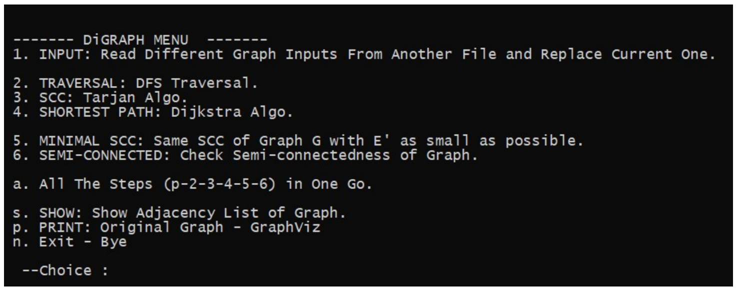
3 TEST CASES

This Section contains the test cases for the Graph operations performed in the program.



```
          To Read Graph:
 * Enter values in the file ( input_files/_read_input.txt )
 * First Line is (int): Num_Of_Vertex Num_of_Edges
 * Each Next line is Edge (int): u v wt
          ** Make sure VERTEX VALUE start from 0 to Num_Of_Vertex-1

 Then Press Enter to Start


 ==> File has been Successfully read.


Press any key to continue . . .
```

Figure 1: DiGraph: Reading File in the Beginning of Program

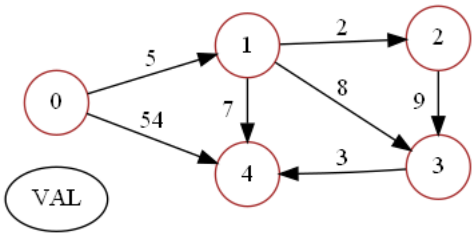


```
------- DiGRAPH MENU  -------
1. INPUT: Read Different Graph Inputs From Another File and Replace Current One.

2. TRAVERSAL: DFS Traversal.
3. SCC: Tarjan Algo.
4. SHORTEST PATH: Dijkstra Algo.

5. MINIMAL SCC: Same SCC of Graph G with E' as small as possible.
6. SEMI-CONNECTED: Check Semi-connectedness of Graph.

a. All The Steps (p-2-3-4-5-6) in One Go.

s. SHOW: Show Adjacency List of Graph.
p. PRINT: Original Graph - GraphViz
n. Exit - Bye

 --Choice :
```

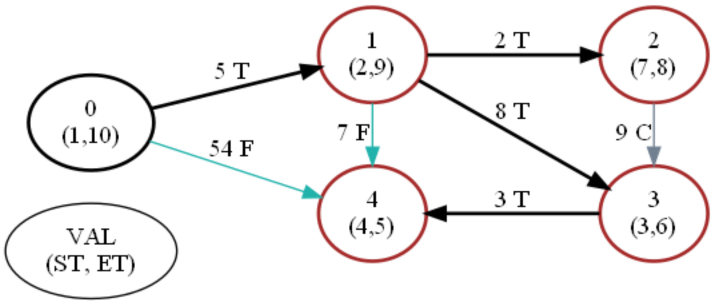
Figure 2: DiGraph: Main Menu

3.1 DFS Traversal

- **Test Case 1:**



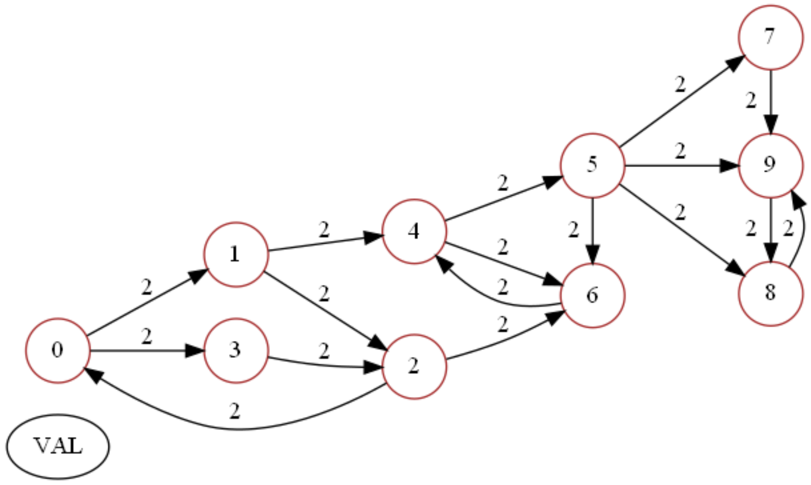
(a) Original Graph



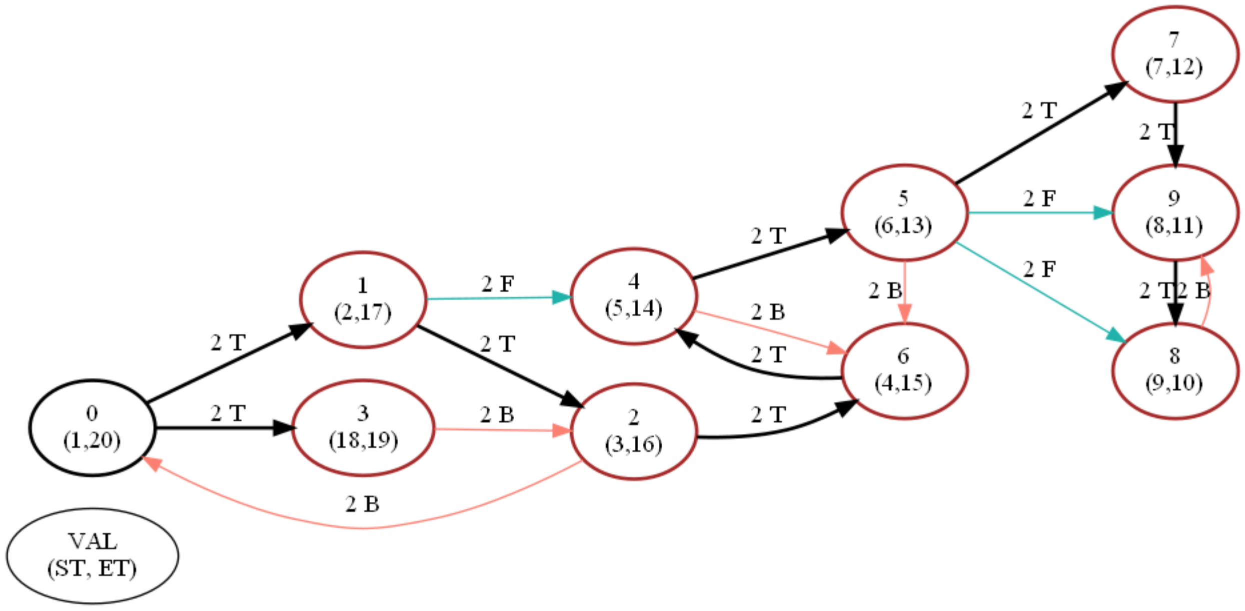
(b) DFS Traversal

Figure 3: DFS Traversal: Graph 1

- **Test Case 2:**



(a) Original Graph

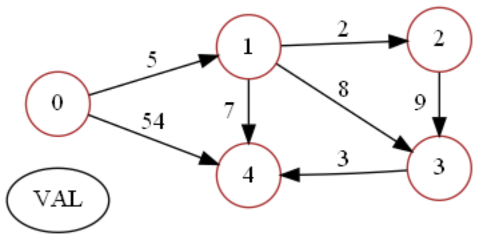


(b) DFS Traversal

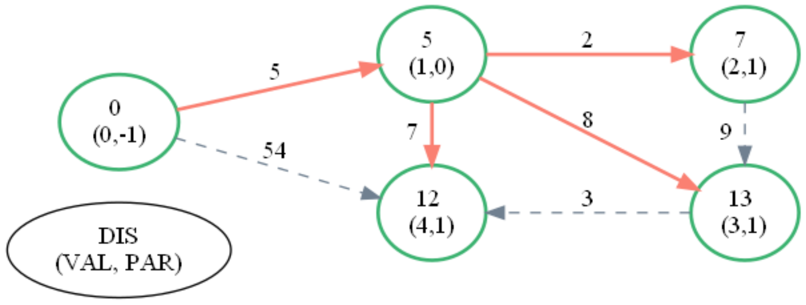
Figure 4: DFS Traversal: Graph 2

3.2 DIJKSTRA SSSP

- **Test Case 1:**



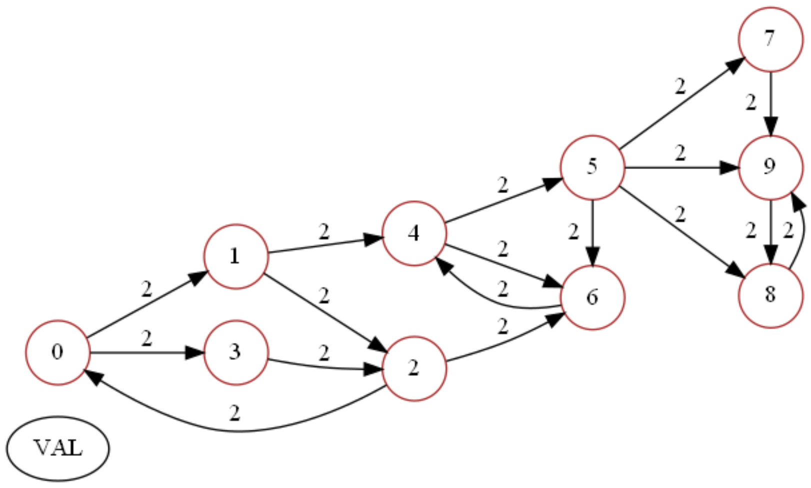
(a) Original Graph



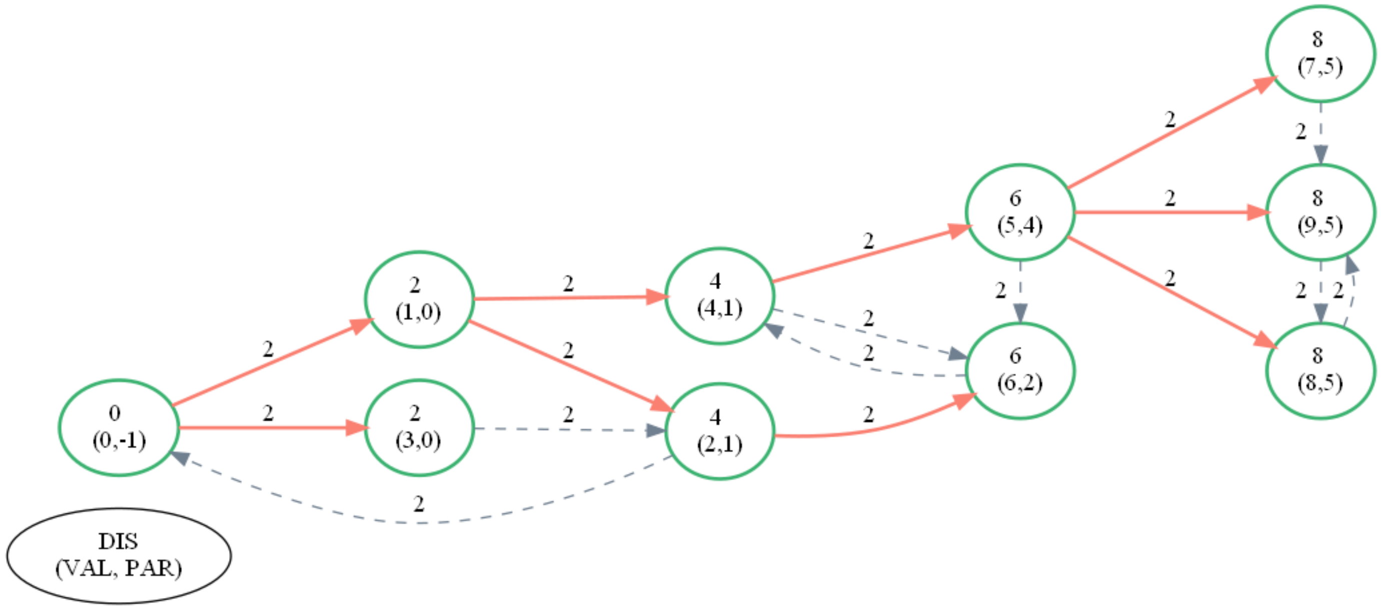
(b) Shortest Distance

Figure 5: DIJKSTRA SSSP: Graph 1

- **Test Case 2:**



(a) Original Graph

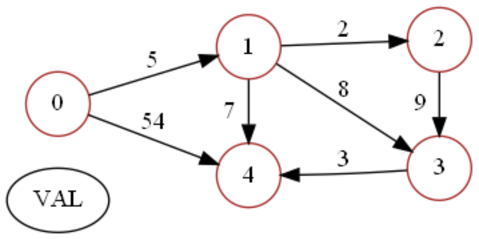


(b) Shortest Distance

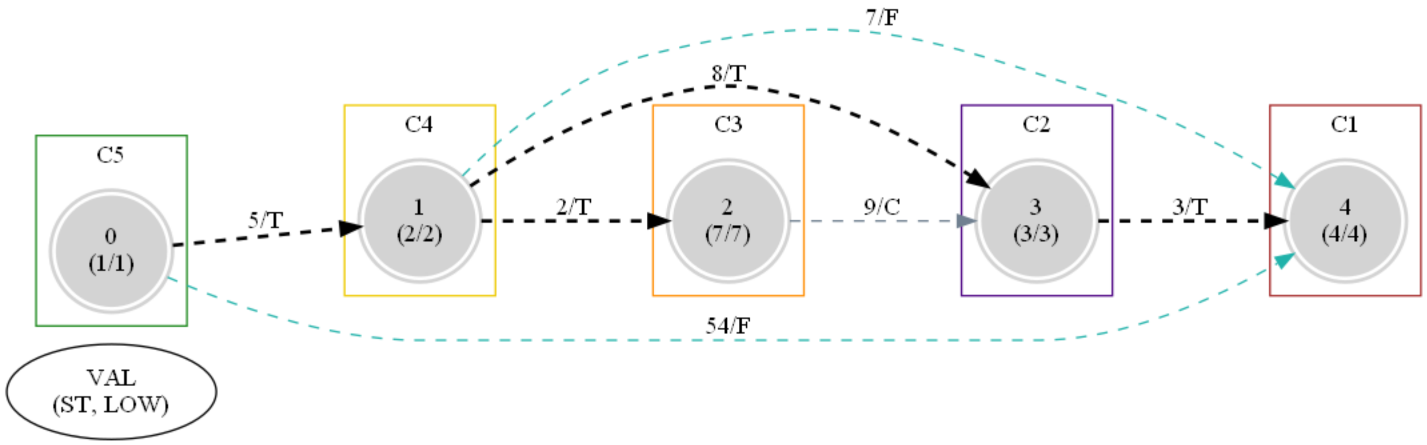
Figure 6: DIJKSTRA SSSP: Graph 2

3.3 Tarjan's SCC

- **Test Case 1:**



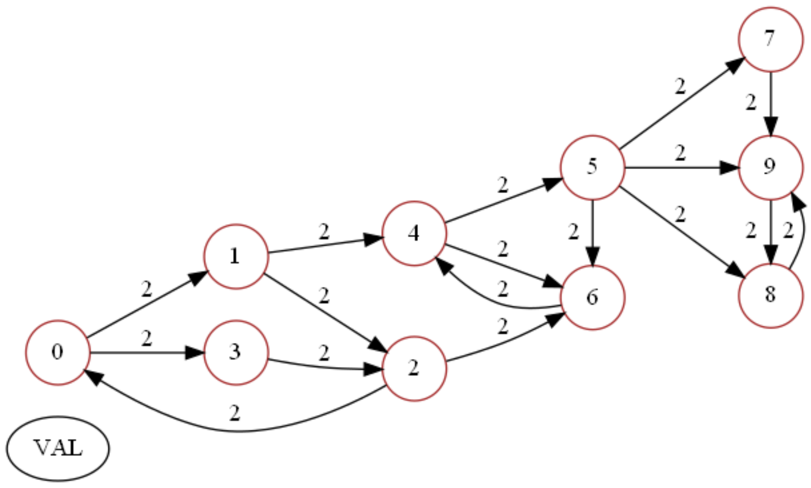
(a) Original Graph



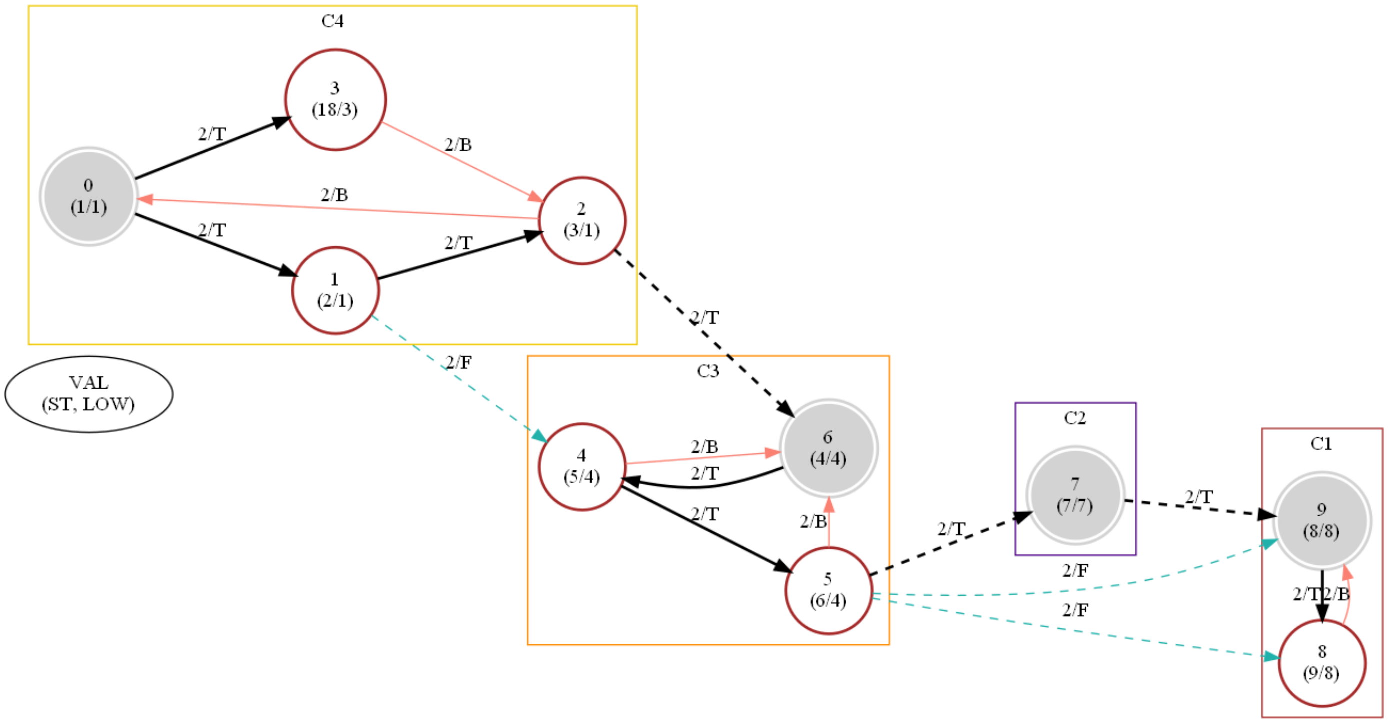
(b) SCC = 5

Figure 7: Tarjan SCC: Graph 1

- **Test Case 2:**



(a) Original Graph

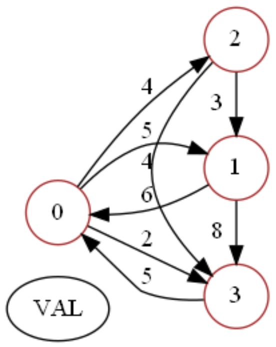


(b) SCC

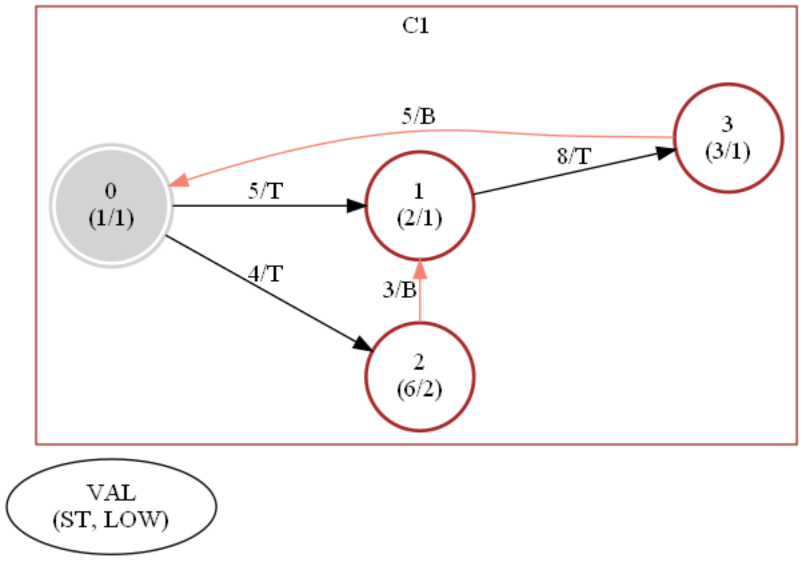
Figure 8: Tarjan SCC: Graph 2

3.4 Minimal Graph

- **Test Case 1:**



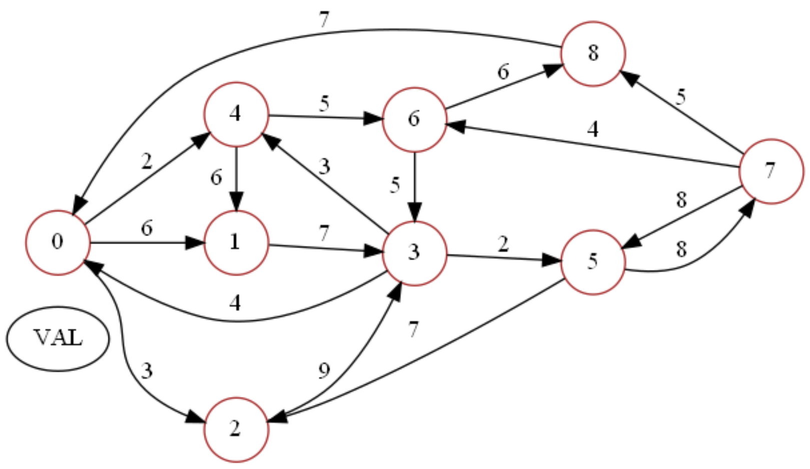
(a) Original Graph



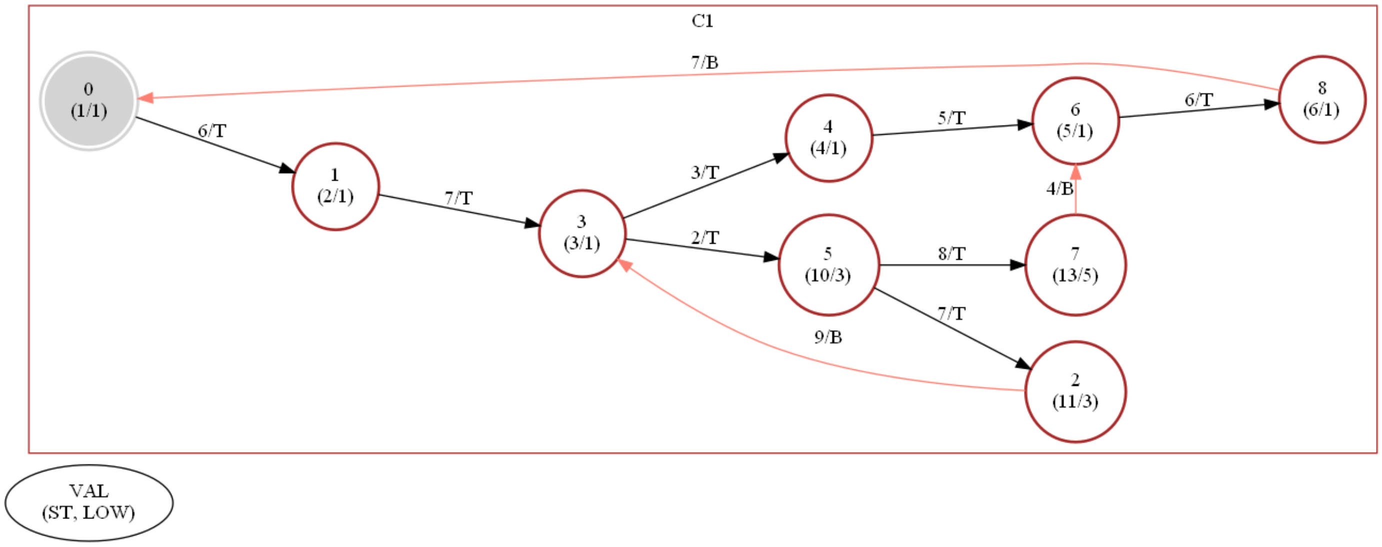
(b) Minimal Graph

Figure 9: Minimal Graph: Graph 1

- **Test Case 2:**



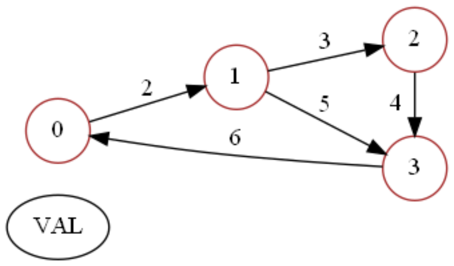
(a) Original Graph



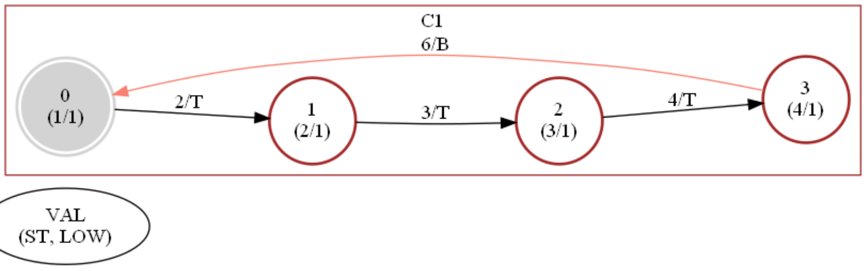
(b) Minimal Graph

Figure 10: Minimal Graph: Graph 2

- **Test Case 3:**



(a) Original Graph

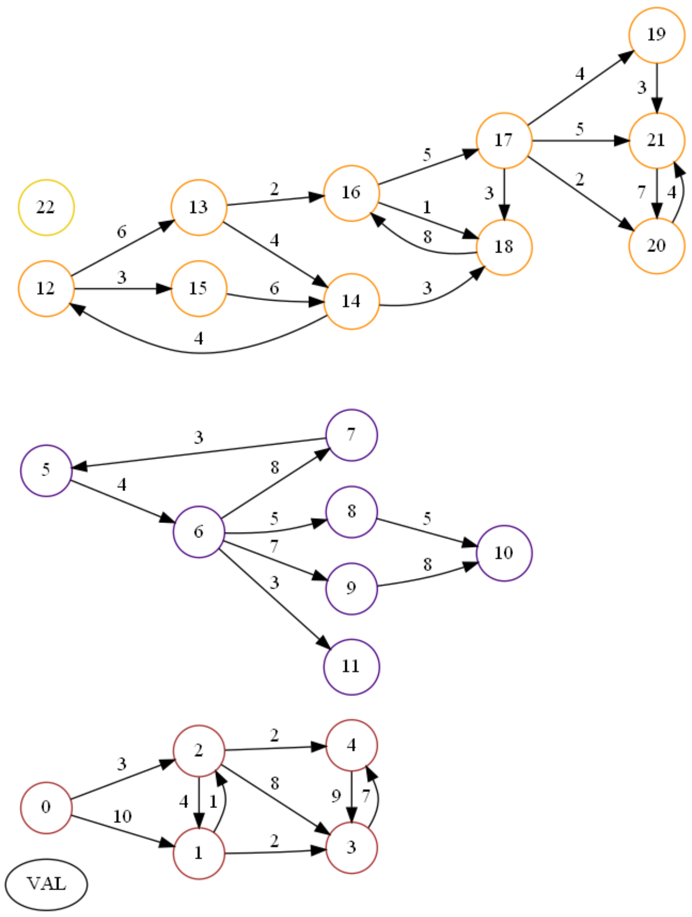


(b) Minimal Graph

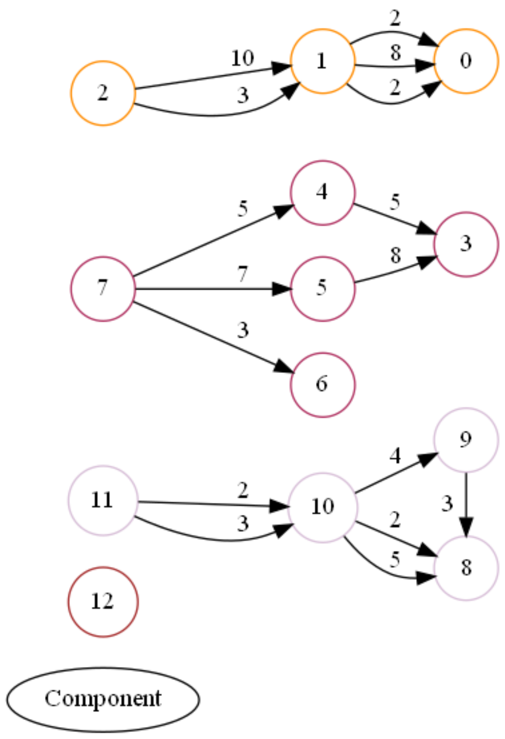
Figure 11: Minimal Graph: Graph 3

3.5 Semi-connected

- **Test Case 1:**

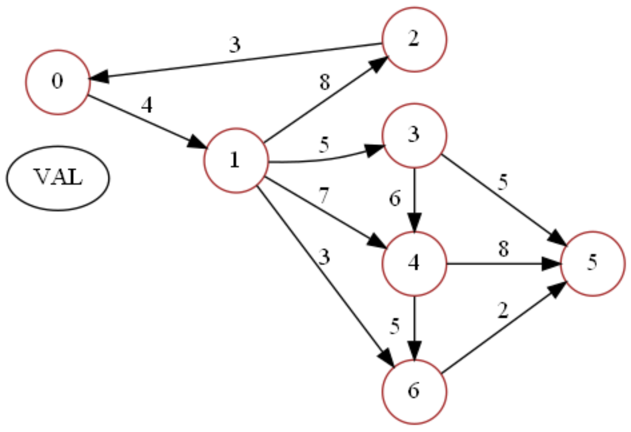


(a) Original Graph

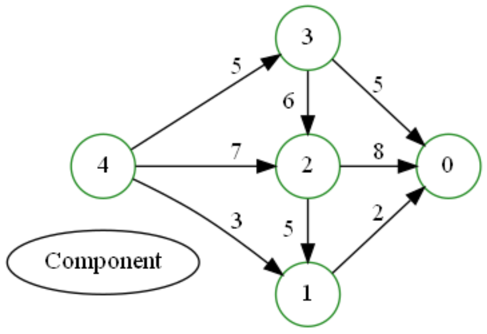


(b) Condensed Graph

Figure 12: Semi-connected: False



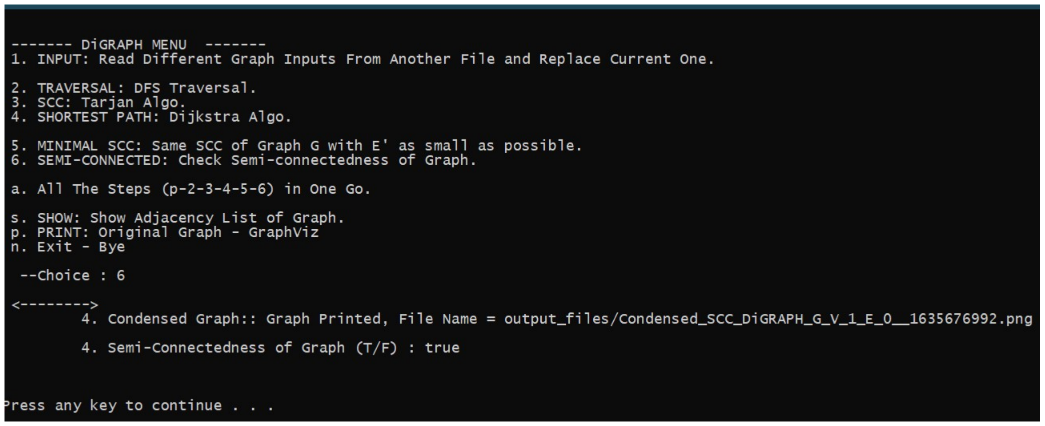
(a) Original Graph



(b) Condensed Graph

Figure 13: Semi-connected: True

- **Test Case 3:**



```
------- DiGRAPH MENU  -------
1. INPUT: Read Different Graph Inputs From Another File and Replace Current One.

2. TRAVERSAL: DFS Traversal.
3. SCC: Tarjan Algo.
4. SHORTEST PATH: Dijkstra Algo.

5. MINIMAL SCC: Same SCC of Graph G with E' as small as possible.
6. SEMI-CONNECTED: Check Semi-connectedness of Graph.

a. All The Steps (p-2-3-4-5-6) in One Go.

s. SHOW: Show Adjacency List of Graph.
p. PRINT: Original Graph - GraphViz
n. Exit - Bye

 --Choice : 6

<-------->
        4. Condensed Graph:: Graph Printed, File Name = output_files/Condensed_SCC_DiGRAPH_G_V_1_E_0__1635676992.png

        4. Semi-Connectedness of Graph (T/F) : true


Press any key to continue . . .
```

Figure 14: Semi-connected: Output on the console screen