Contents

1 Introduction 1

2 Related Work 1
2.1 Creating batch files or shell scripts . . . 1
2.2 Storing the entire execution process . . . 1
2.3 One combined print function . . . 1
2.4 Use of unordered_map . . . 1
2.5 Variables to be fixed before execution . . . 2

3 Build Mode 2
3.1 void createGraph() . . . 2
3.2 void generateInput() . . . 2
3.3 void executeModification() . . . 2
3.4 void performBulkModification() . . . 2

4 Operations Mode 2
4.1 Depth first search and edge classification . . . 3
4.1.1 vector runDFS(vector,int) . . . 3
4.1.2 vector traverseDFS(int,vector) . . . 3
4.1.3 Example . . . 4
4.2 Strongly Connected Components using Tarjan's Algorithm . . . 5
4.2.1 vector runTarjan(vector) . . . 5
4.2.2 vector runDFSForTarjan(int,vector) . . . 5
4.2.3 Example . . . 5
4.3 Build a minimal graph by removing the redundant edges . . . 6
4.3.1 void addEdgesWithinSCC(Graph,vector,vector) . . . 6
4.3.2 int * isSubset(vector,int,int,int) . . . 6
4.3.3 void addEdgesInterSCC(Graph,Graph,vector,vector) . . . 6
4.3.4 Example . . . 7
4.4 Check if the graph is semi connected or not . . . 7
4.4.1 void buildDAG(vector,Graph) . . . 7
4.4.2 int * runTopologicalSort() . . . 8
4.4.3 bool checkSemiConnected(int *) . . . 8
4.4.4 Example . . . 8
4.5 Find shortest paths using Dijkstra's Algorithm . . . 9
4.5.1 int * runDijkstra(int) . . . 9
4.5.2 int * relaxEdge(Heap,int *,int,int,int) . . . 9
4.5.3 Example . . . 9

5 Flow of Execution 11
5.1 User Prompt . . . 11
5.2 Example Testcase . . . 13
5.2.1 For creating a graph . . . 13
5.2.2 Modifying a graph in bulk . . . 14
5.3 Instructions to execute the code . . . 14

6 Testing **15**
6.1 Minimal Graph Example - 1 15
6.2 Minimal Graph Example - 2 15
6.3 Minimal Graph Example - 3 16
6.4 Minimal Graph Example - 4 16
6.5 Minimal Graph Example - 5 17
6.6 Dijkstra's Algorithm : Source = 14 18

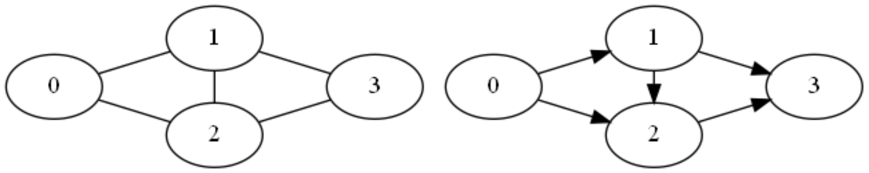
CS513: Graph

Prateekshya Priyadarshini

M.Tech CSE

1 Introduction

Graph is a data structure which can store relevant information in the nodes and connect them in a meaningful way with edges. These are the examples of some graphs.



The left one is an undirected graph and the second one is a directed graph. We can also add weights to the edges. Those kind of graphs would be called **weighted graphs**. We can represent graphs using adjacency matrix or adjacency lists.
For simplicity, here we do not store any information in the nodes. We simply number them from 0 to $V - 1$ where V is the number of nodes.

2 Related Work

2.1 Creating batch files or shell scripts

Since the commands to generate a .png file from a .gv file are not straight forward for a beginner, this program generates batch file in windows and shell script in linux which contains all those commands. Every time a new tree is printed, these files get executed and the respective images get generated. For reference, the images won't be deleted or replaced till the program is being executed.

2.2 Storing the entire execution process

The entire execution process is stored in **output.txt**. This file can be referred to check what went wrong, which images refer to which trees etc.

2.3 One combined print function

There is only one function to generate the images of graphs which takes variable arguments as per the requirement.

2.4 Use of unordered_map

Since there is a significant usage of unordered maps, follow the steps carefully if you're using Dev cpp for running the program.

2.5 Variables to be fixed before execution

There are a few variables those can be chaged before ecxecution. For now they are set to some default stable values. Those variables are listed in first four lines of **functions.h**. You can add more colours and set the noc values to the number of colours. Percentage ranges of dense, sparse or medium graphs can be changed as per requirement. There are two prime numbers being used to create custom hash values for edges. Try setting it to some higher values for avoiding key clashes.

This application works in two modes i.e. Build mode and Operations mode. The next two sections are dedicated for the description of these modes.

3 Build Mode

In build mode, we can create a new graph, add or remove edges from the existing graph and print the graph. We can also write a sequence of addition and removal of edges in a file and put it to execute those operations in bulk. This mode is present to make the execution flexible.

3.1 void createGraph()

This function asks us whether we want to give a manual input file or generate a random graph. For manual input file, we need to write a .txt file in the given format and write the file name in the console. If we choose random input file, it will further ask to choose the type of graph we want. We can choose one out of (sparse/medium/dense) graph. Alternatively we can give a percentage value. If we give the value as 15, it will create 15% of total possible edges in the graph. Self loops are not considered in total possible edges calculation, but it adds self loops in the graph. It generates an image for the newly created graph.

3.2 void generateInput()

It generates a random input file based on the parameters chosen.

3.3 void executeModification()

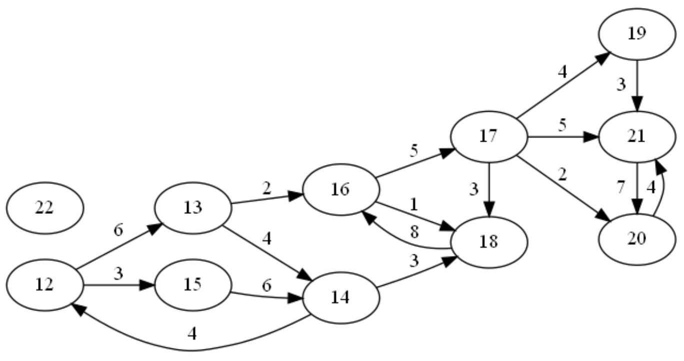
This function is meant for modifying an existing graph by adding or removing edges. If we try to perform a bulk modification, then int prints the new graph.

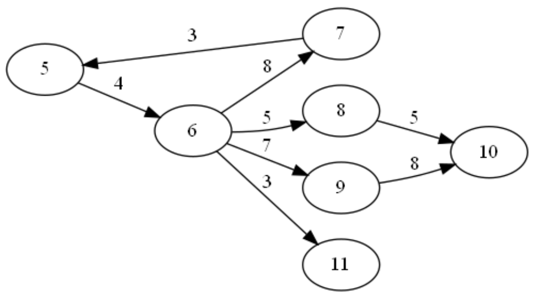
3.4 void performBulkModification()

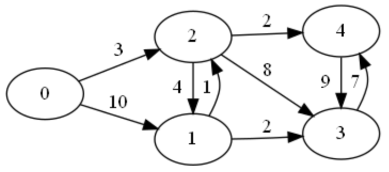
This function takes a .txt file and performs all the operations written in it.

4 Operations Mode

In this mode, we perform all the required operations. We shall use the following graph as an example for all of the operations performed.







4.1 Depth first search and edge classification

There are two functions that perform DFS for every unvisited node and classify the edges.

4.1.1 vector runDFS(vector,int)

This function is called first. It creates a few data structures for storing the start and end times of visiting a vertex, order of nodes reached during the search, parents and children in the DFS tree. Then for every vertex it checks if the vertex is unvisited, then it calls the below function for that vertex.

4.1.2 vector traverseDFS(int,vector)

This function stores all the information in the above mentioned data structures. For detecting the edge types, the logic is as follows-
For every edge-

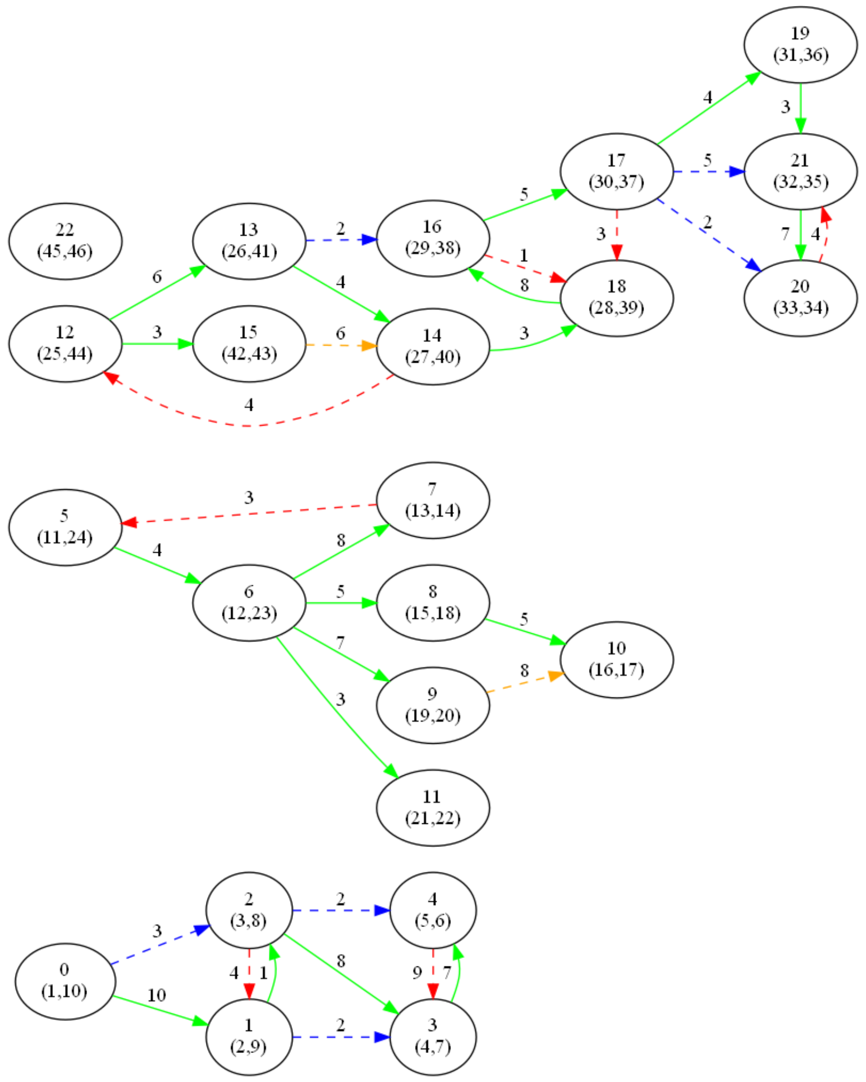
- **If the destination node is unvisited**
  This edge points to a visited node from an unvisited node. So, it will be a tree edge, always.
- **Else**
  - **If end time of the destination node is not yet calculated**
    This edge points from a visited node to a visited but unfinished node, so it will be a back edge.

– **Else if start time of source node is smaller than that of the destination node**
This edge points to a node which is visited later from a node which is visited earlier. But, since it is not a tree edge, so, it will be a forward edge.

– **Else**
For all other cases, the edge will be a cross edge.

4.1.3 Example

The following is a graph indicating all the start and end times along with the edge classifications.



Solid green edges: - Tree edges
Dashed red edges: - Back edges
Dashed blue edges: - Forward edges
Dashed yellow edges: - Cross edges
Time complexity is $O(V + E)$.

4.2 Strongly Connected Components using Tarjan's Algorithm

There are two functions that implement **Tarjan's Algorithm** for finding the strongly connected components.

4.2.1 vector runTarjan(vector)

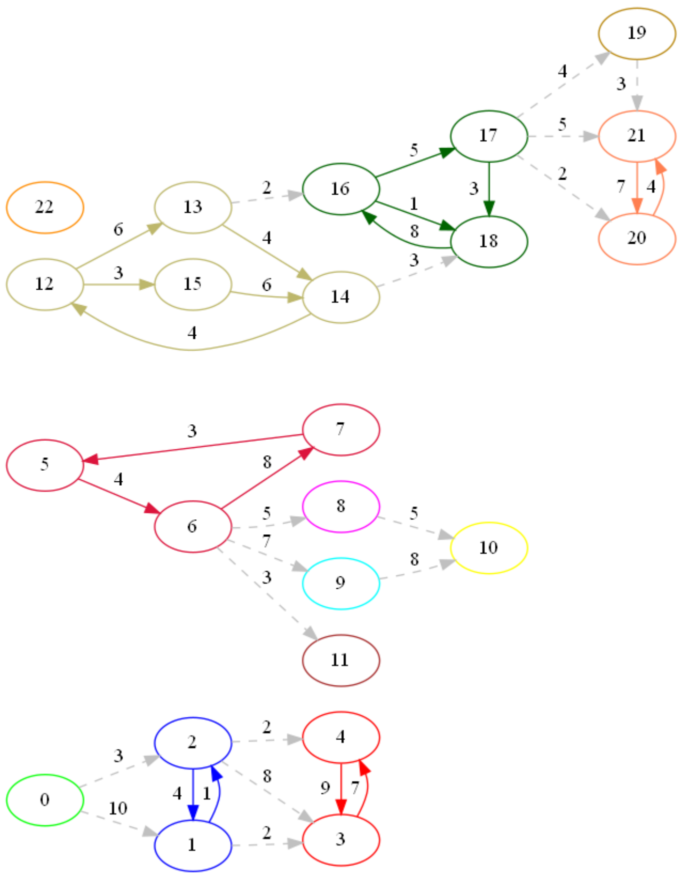
This function is called first. It creates a few data structures for storing discovery and low times of all the vertices. Then for each vertex whose discovery time is 0 i.e. which is not yet discovered, it calls the DFS function specific for Tarjan's algorithm i.e. the below function for that vertex.

4.2.2 vector runDFSForTarjan(int,vector)

This function stores all the informations in the above data structures and follows Tarjan's algorithm to return the list of strongly connected components. In the algorithm, the key step is to update the low value to be the minimum of low of source and destination vertices if the edge is a tree edge and the minimum of low of source vertex and discovery of destination vertex if the edge is a back edge. We use a stack to keep track of the order of vertices visited. Then we keep popping from the stack till we reach the current vertex. All the popped vertices belong to a single strongly connected component.

4.2.3 Example

The following is a graph indicating all the strongly connected components.



Each different colour represents a different strongly connected component. That means all the vertices and edges belonging to a single strongly connected component have one colour and the colour is unique.

The dashed grey edges represent the edges connecting two strongly connected components.
Time complexity is $O(V + E)$.

4.3 Build a minimal graph by removing the redundant edges

There are two functions that find the minimal graph of a given graph. Minimal graph is a graph obtained from the original graph which has same component graph and same strongly connected components but minimum possible edges. That means we reduce the edges and not the reachability. For performing these operations, first we run a DFS using vector runDFS(vector,int) to get the edge classifications and the parents and children. Then we use vector runTarjan(vector) to find all the strongly connected components. Then we use void buildDAG(vector,Graph) to get the DAG of the graph. All these information will be used while adding the set of minimal edges to the new graph.
We divide the further process into two steps.

1. Adding the edges falling within the strongly connected components.

2. Adding the edges connecting two different strongly connected components.

4.3.1 void addEdgesWithinSCC(Graph,vector,vector)

When we add the edges inside a strongly connected component, we follow a few steps.
Step-1: Add all the tree edges
This step makes the further steps easier.
Step-2: Ignore all the forward edges
Since we can reach from the same source to the same destination by using the tree edges, we do not need any forward edge.
Step-3: Assumptions for cross edges
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. This is a greedy choice, since by doing this we can reach a larger set of vertices using the cross edge followed by some tree edges.
Step-4: Assumptions for back edges
Addition of back edges is the lengthiest step. Because, till now we have reachability from the root of the DFS tree to the leaves. The remaining step is to add the edges which will help us reaching back the root from the leaves. So, for each leaf node we have to find a path back to the root of the strongly connected component. Hence, we find all the nodes who do NOT have any children or their children belong to a different strongly connected component and add such nodes to a queue. Till this queue becomes empty we dequeue them one by one and add them to another queue.
So for each leaf node, till the second queue becomes empty we dequeue them one by one and for all the backedges starting from that vertex, we choose the edge having a larger reachability i.e. we choose the edge whose destination is having the lowest start time. Then we add it to the minimal graph only if it the nodes on the path are not subset of any other backedge or vice versa. This checking is done by int * isSubset(vector,int,int,int). Then we enqueue the destination to the second queue. If the vertex does not have any backedge then we add its children to the queue. We repeat this process till we reach the root of the component.

4.3.2 int * isSubset(vector,int,int,int)

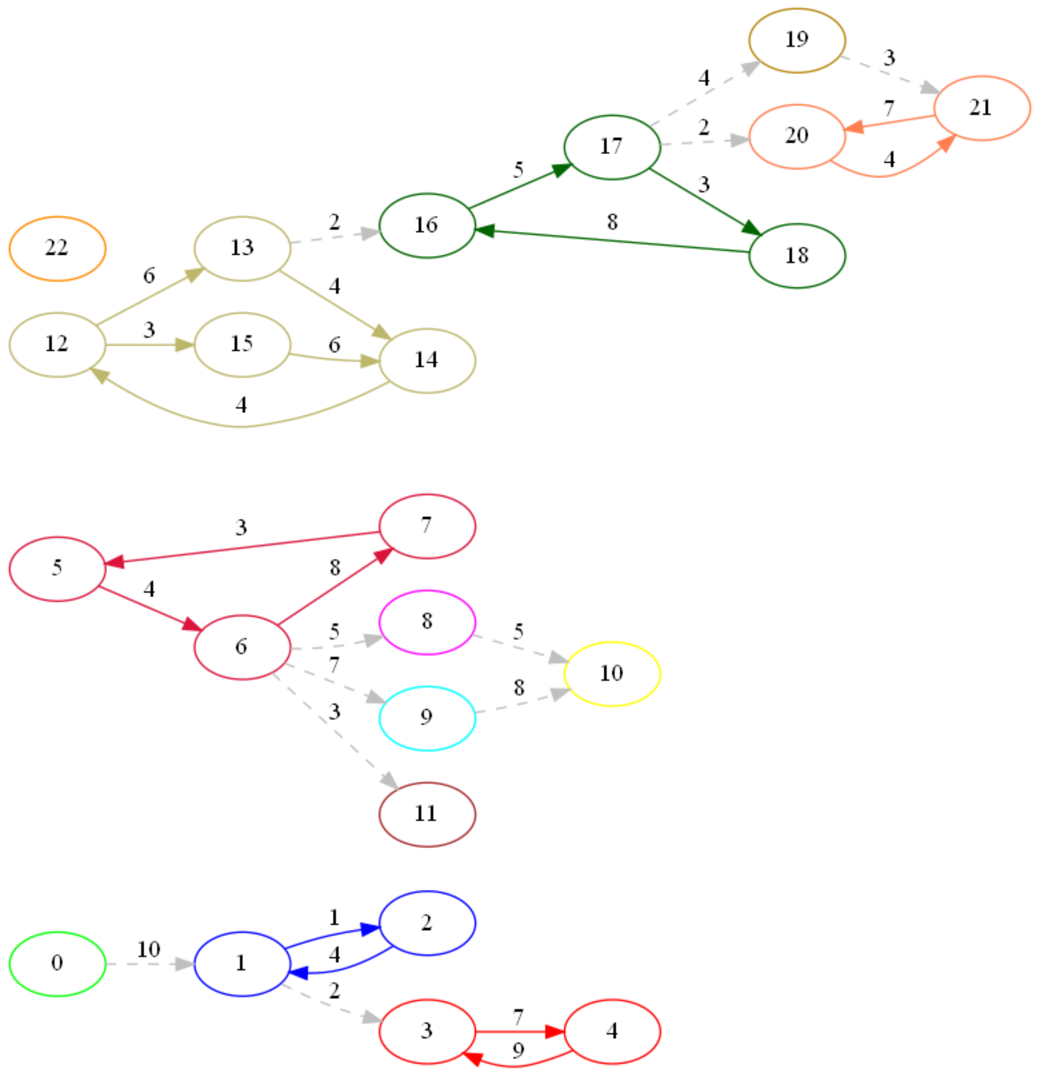
This function checks whether the current backedge is a subset of any previously added back edge or vice versa. The backedge which is a subset will be removed from the final graph.

4.3.3 void addEdgesInterSCC(Graph,Graph,vector,vector)

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

4.3.4 Example

The following is the minimal graph of the above considered graph.



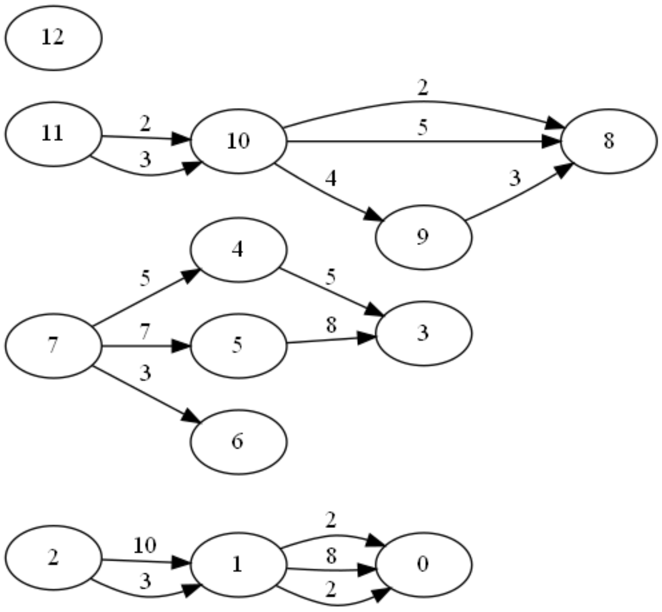
We can see all the strongly connected components are intact and the edges are minimal. Any further removal will destroy the connectivity of the graph.

4.4 Check if the graph is semi connected or not

There are three functions that help in checking whether the graph is semi connected or not. First we find all the strongly connected components using vector runTarjan(vector). Then we use those components to build a Directed Acyclic Graph by calling below function.

4.4.1 void buildDAG(vector,Graph)

The DAG for the above graph will look like-



It actually takes one strongly connected component as one vertex. So we are not left with any more cycles in the new graph. Then we run Topological Sort on this graph by calling the below function.

4.4.2 int * runTopologicalSort()

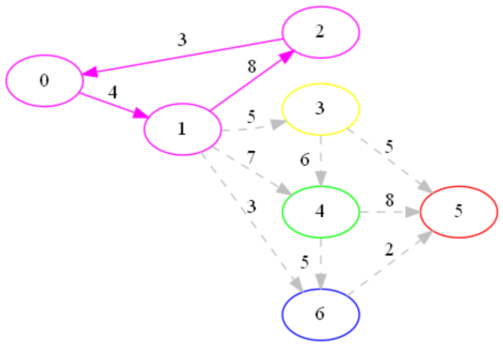
This function implements topological sort and returns the sorted vertices. Then we call the below function.

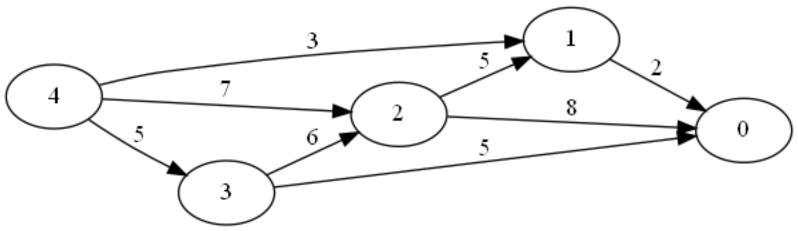
4.4.3 bool checkSemiConnected(int *)

This function checks if each adjacent verteices in the topological order has a path either $u - - > v$ or $v - - > u$ or both.
Since, we can see this is not the case for the above DAG, so our example is not semi connected.

4.4.4 Example

The following is a graph which is semi connected. The DAG is also given after it.





Time complexity is $O(V+E)$ for Tarjan's algorithm, $O(V+E)$ for building a DAG, $O(V+E)$ for Topological Sort and $O(V)$ for checking semiconnection property. In total it is $O(4(V+E))$ i.e. $O(V+E)$.

4.5 Find shortest paths using Dijkstra's Algorithm

There are two functions that implement **Dijkstra's Algorithm** for finding the shortes path distance from a particular source vertex to all other vertices.

4.5.1 int * runDijkstra(int)

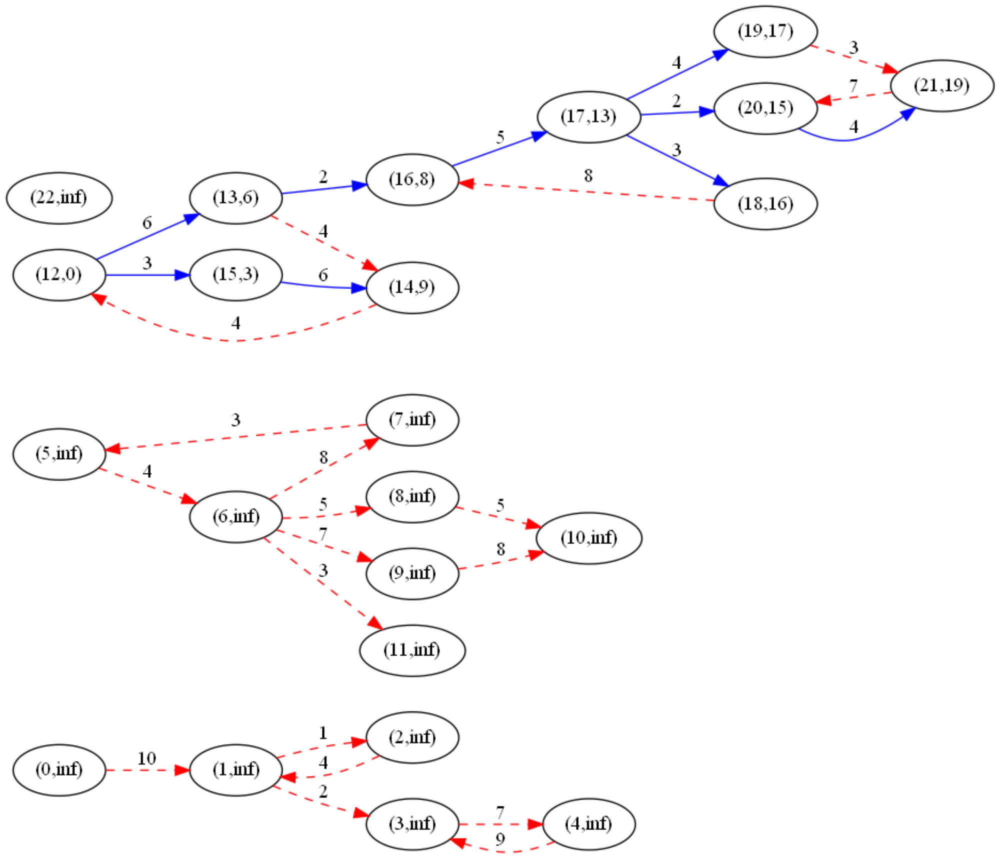
This function is called first. It creates a few data structures for storing the path costs and the parent nodes. A min heap is implemented from scratch to keep track of the minimum cost vertices. This function extracts the root of the min heap $V-1$ times and calls the below function for each edge starting from that particular vertex.

4.5.2 int * relaxEdge(Heap,int *,int,int,int)

This function relaxes an edge. If the cost of the destination vertex is updated during relaxation, the cost is updated and decrease key operation is done on the heap.

4.5.3 Example

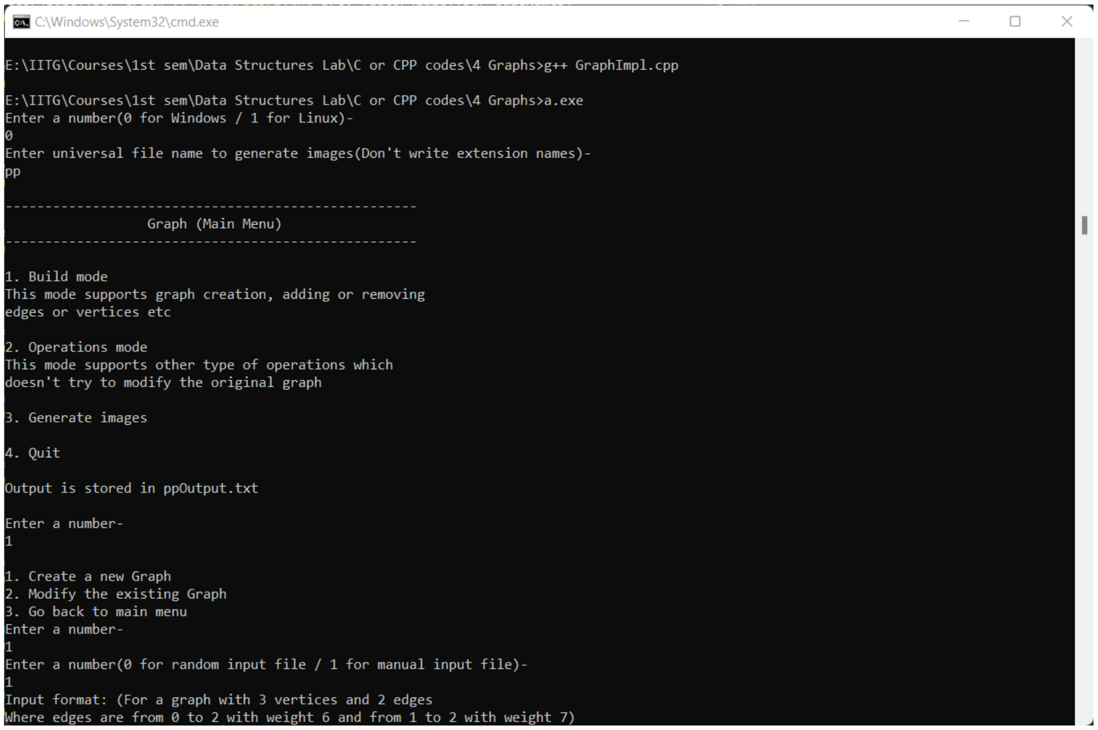
The following is a graph indicating the edges which participate in shortes path.



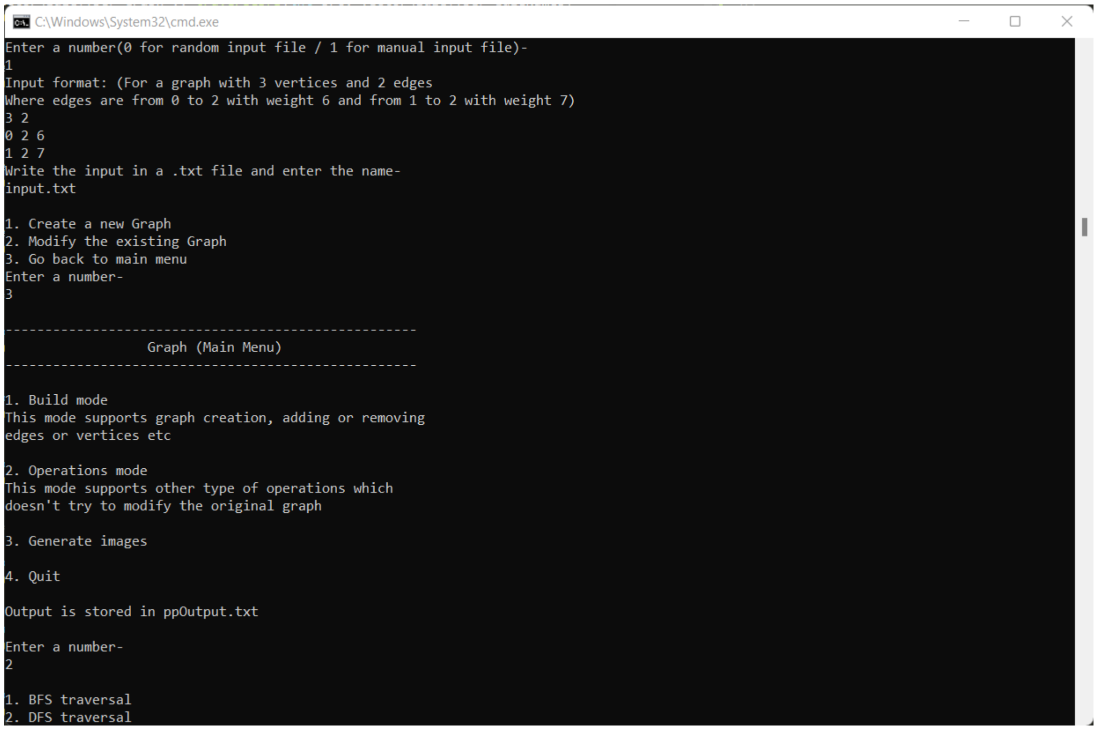
Source vertex is 12. Solid blue edges participate in shortest path while the dashed red edges do not. Time complexity is $O(V + E \log V)$ since a min heap is implemented from scratch.

5 Flow of Execution

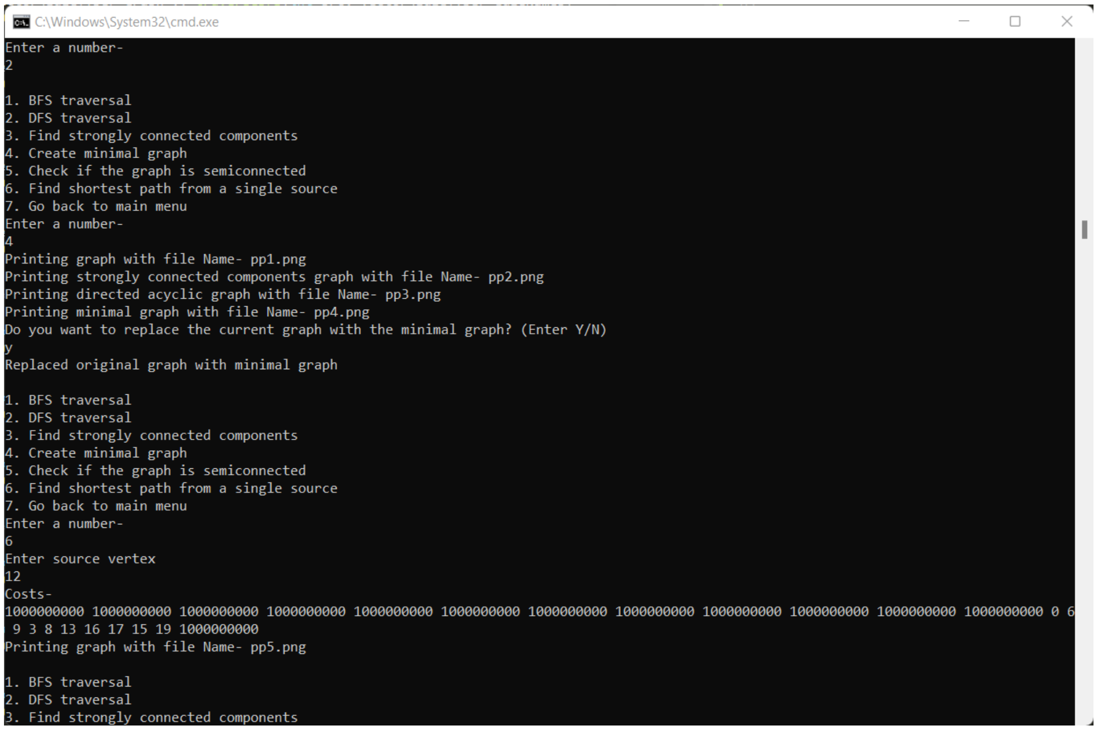
5.1 User Prompt



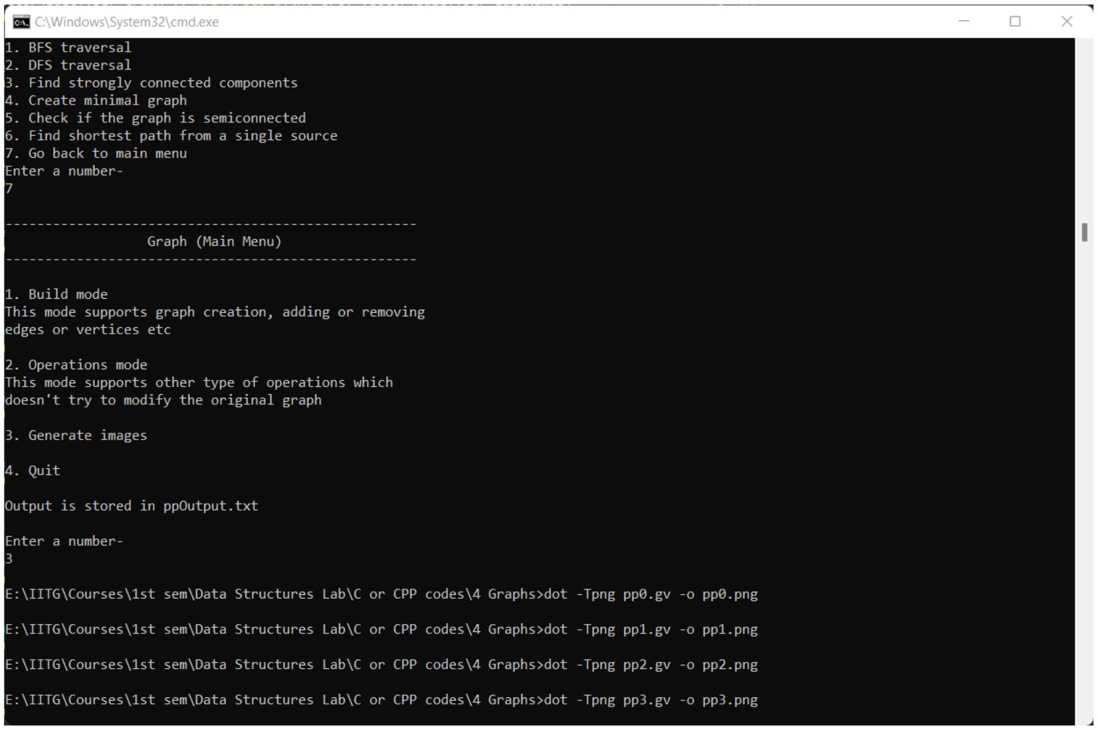
```
E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>g++ GraphImpl.cpp

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>a.exe
Enter a number(0 for Windows / 1 for Linux)-
0
Enter universal file name to generate images(Don't write extension names)-
pp

----------------------------------------------------
                  Graph (Main Menu)
----------------------------------------------------

1. Build mode
This mode supports graph creation, adding or removing
edges or vertices etc

2. Operations mode
This mode supports other type of operations which
doesn't try to modify the original graph

3. Generate images

4. Quit

Output is stored in ppOutput.txt

Enter a number-
1

1. Create a new Graph
2. Modify the existing Graph
3. Go back to main menu
Enter a number-
1
Enter a number(0 for random input file / 1 for manual input file)-
1
Input format: (For a graph with 3 vertices and 2 edges
Where edges are from 0 to 2 with weight 6 and from 1 to 2 with weight 7)
```



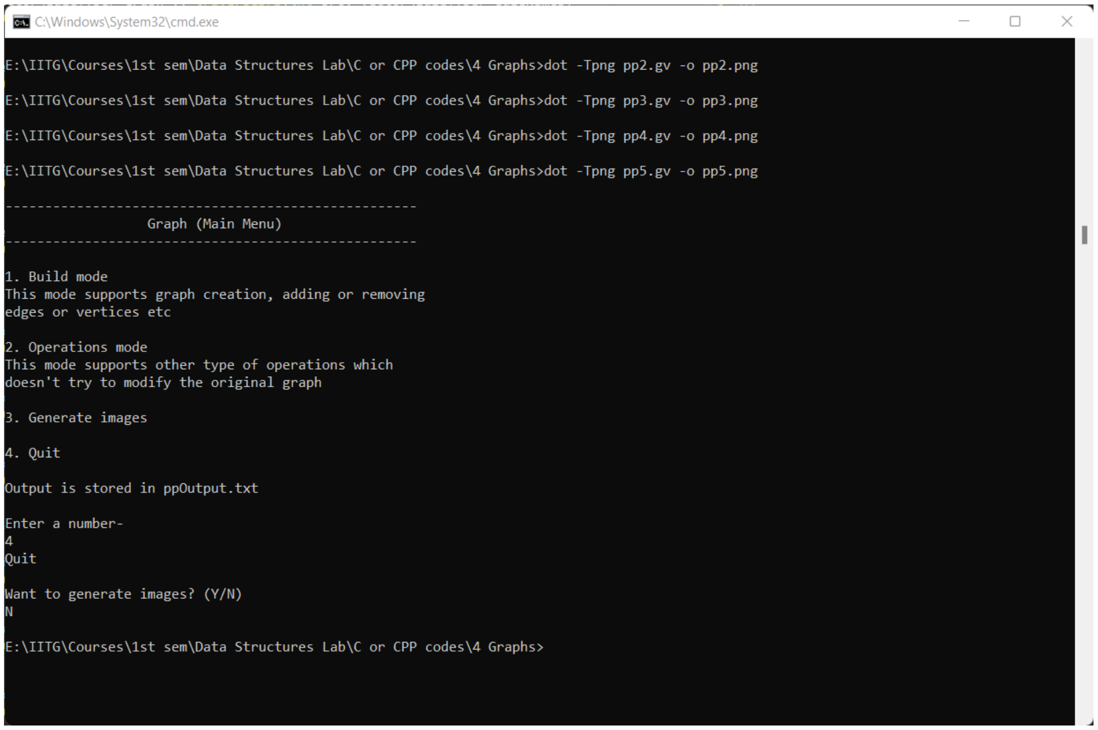
```
Enter a number(0 for random input file / 1 for manual input file)-
1
Input format: (For a graph with 3 vertices and 2 edges
Where edges are from 0 to 2 with weight 6 and from 1 to 2 with weight 7)
3 2
0 2 6
1 2 7
Write the input in a .txt file and enter the name-
input.txt

1. Create a new Graph
2. Modify the existing Graph
3. Go back to main menu
Enter a number-
3

----------------------------------------------------
                  Graph (Main Menu)
----------------------------------------------------

1. Build mode
This mode supports graph creation, adding or removing
edges or vertices etc

2. Operations mode
This mode supports other type of operations which
doesn't try to modify the original graph

3. Generate images

4. Quit

Output is stored in ppOutput.txt

Enter a number-
2

1. BFS traversal
2. DFS traversal
```



```
C:\Windows\System32\cmd.exe
Enter a number-
2

1. BFS traversal
2. DFS traversal
3. Find strongly connected components
4. Create minimal graph
5. Check if the graph is semiconnected
6. Find shortest path from a single source
7. Go back to main menu
Enter a number-
4
Printing graph with file Name- pp1.png
Printing strongly connected components graph with file Name- pp2.png
Printing directed acyclic graph with file Name- pp3.png
Printing minimal graph with file Name- pp4.png
Do you want to replace the current graph with the minimal graph? (Enter Y/N)
y
Replaced original graph with minimal graph

1. BFS traversal
2. DFS traversal
3. Find strongly connected components
4. Create minimal graph
5. Check if the graph is semiconnected
6. Find shortest path from a single source
7. Go back to main menu
Enter a number-
6
Enter source vertex
12
Costs-
1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 1000000000 0 6
 9 3 8 13 16 17 15 19 1000000000
Printing graph with file Name- pp5.png

1. BFS traversal
2. DFS traversal
3. Find strongly connected components
```



```
C:\Windows\System32\cmd.exe
1. BFS traversal
2. DFS traversal
3. Find strongly connected components
4. Create minimal graph
5. Check if the graph is semiconnected
6. Find shortest path from a single source
7. Go back to main menu
Enter a number-
7

----------------------------------------------------
                  Graph (Main Menu)
----------------------------------------------------

1. Build mode
This mode supports graph creation, adding or removing
edges or vertices etc

2. Operations mode
This mode supports other type of operations which
doesn't try to modify the original graph

3. Generate images

4. Quit

Output is stored in ppOutput.txt

Enter a number-
3

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp0.gv -o pp0.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp1.gv -o pp1.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp2.gv -o pp2.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp3.gv -o pp3.png
```



```
C:\Windows\System32\cmd.exe

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp2.gv -o pp2.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp3.gv -o pp3.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp4.gv -o pp4.png

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>dot -Tpng pp5.gv -o pp5.png

----------------------------------------------------
                  Graph (Main Menu)
----------------------------------------------------

1. Build mode
This mode supports graph creation, adding or removing
edges or vertices etc

2. Operations mode
This mode supports other type of operations which
doesn't try to modify the original graph

3. Generate images

4. Quit

Output is stored in ppOutput.txt

Enter a number-
4
Quit

Want to generate images? (Y/N)
N

E:\IITG\Courses\1st sem\Data Structures Lab\C or CPP codes\4 Graphs>
```

5.2 Example Testcase

5.2.1 For creating a graph

```
23 34
0 1 10
0 2 3
1 2 1
1 3 2
2 1 4
2 3 8
2 4 2
3 4 7
4 3 9
5 6 4
6 7 8
7 5 3
6 8 5
6 9 7
6 11 3
8 10 5
9 10 8
12 13 6
12 15 3
13 14 4
13 16 2
14 12 4
14 18 3
15 14 6
```

16 17 5
16 18 1
17 18 3
17 19 4
17 20 2
17 21 5
18 16 8
19 21 3
20 21 4
21 20 7

It means there are 23 vertices and 34 edges. The rest of the rows are in the format **source destination weight**.

5.2.2 Modifying a graph in bulk

1 2 4 5
1 4 6 7
2 4 5

It means
1 2 4 5 (Add an edge from 2 to 4 with weight 5)
1 4 6 7 (Add an edge from 4 to 6 with weight 7)
2 4 5 7 (Remove the edge from 4 to 5 with weight 7)

5.3 Instructions to execute the code

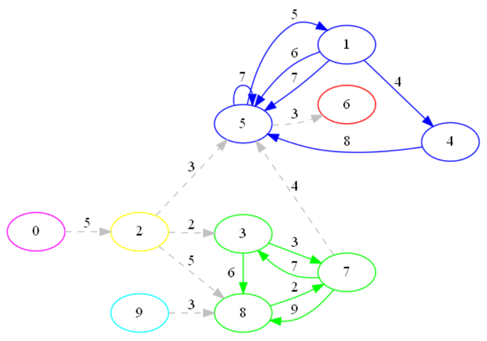
1. If you're using Dev C++, Follow the steps to support *unordered_map*. Go to tools–compiler option–general tab–tick mark option (add the following commands when calling compiler)–add -std=c++11 there. Then build and execute the project.
2. In linux or GNU windows compiler, open the terminal or command prompt and type "g++ GraphImpl.cpp" and hit enter.
3. Then for linux type "./a.out". For windows type "a.exe" or "GraphImpl.exe" (One of them should work). Hit enter.
4. You can also write "./GraphImpl" and hit enter to execute the makefile which is provided.
5. Now the prompt will be displayed. Enter 0 or 1 for windows or linux respectively.
6. Give a name for all the files which are going to be generated during execution.
7. Write a .txt file according to the given format and enter the file name there.
8. The entire execution process can be visualized in $< user\_given\_name > Output.txt$.
9. After printing a graph, you can view the images in the same directory. The respective file names are written in the output file.
10. You can give multiple .txt files one after another. For example- in the first run, you can create a very dense graph, find the minimal graph and replace it with the original one, remove a few edges and then proceed for shortest paths or strongly connected components. Later you can create a new graph and do the same processing on it. All the images will be stored simultaneously.
11. You can run the batch file or shell script to manually generate the images.

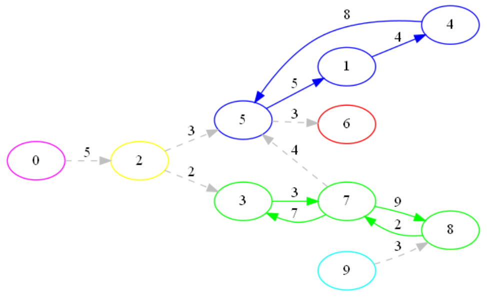
12. Avoid giving percentage more than 50. It is going to create a mess and you won't be able to visualize it. Moreover, there is a huge possibility of having only a single strongly connected component.

13. There will always be an option to go back to the main menu so that you can either generate all the images for the graphviz files written till now or quit.

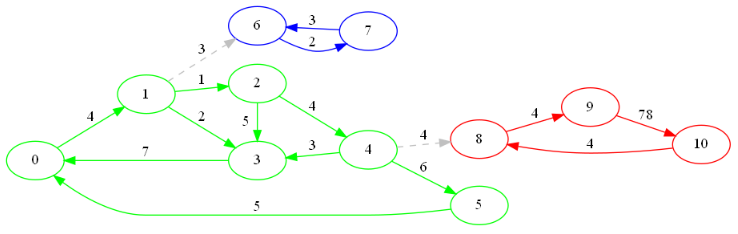
6 Testing

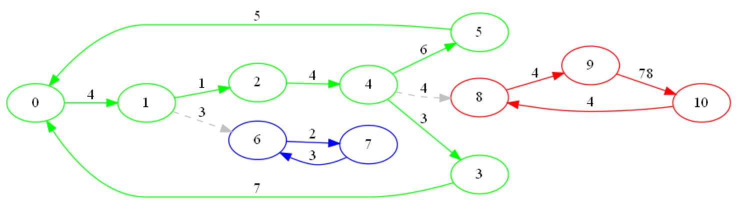
6.1 Minimal Graph Example - 1



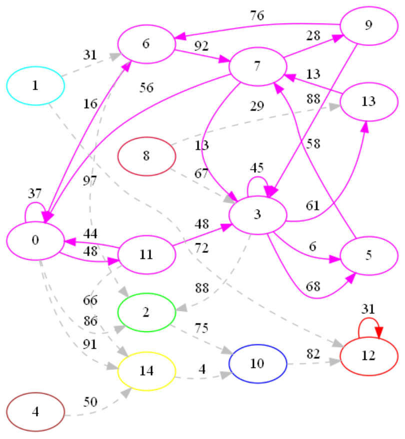


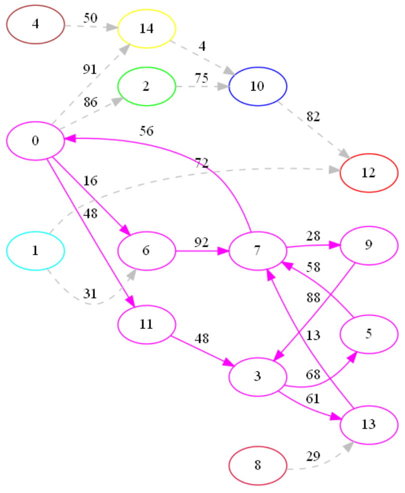
6.2 Minimal Graph Example - 2



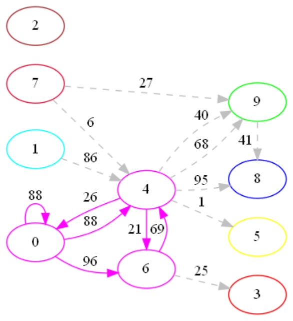


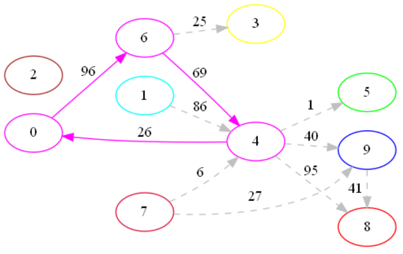
6.3 Minimal Graph Example - 3



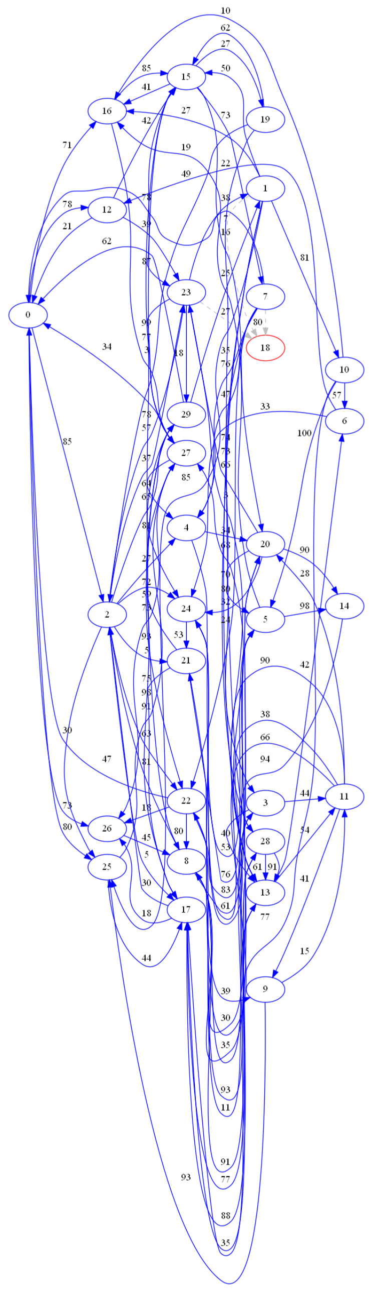


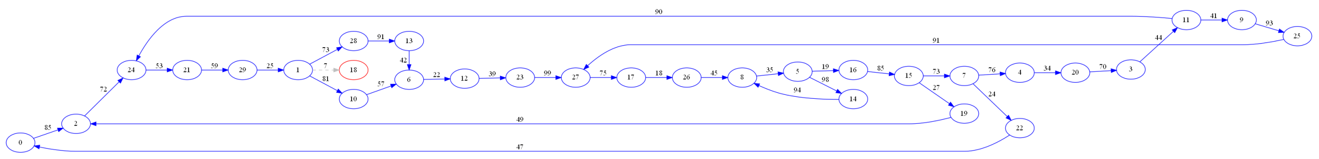
6.4 Minimal Graph Example - 4



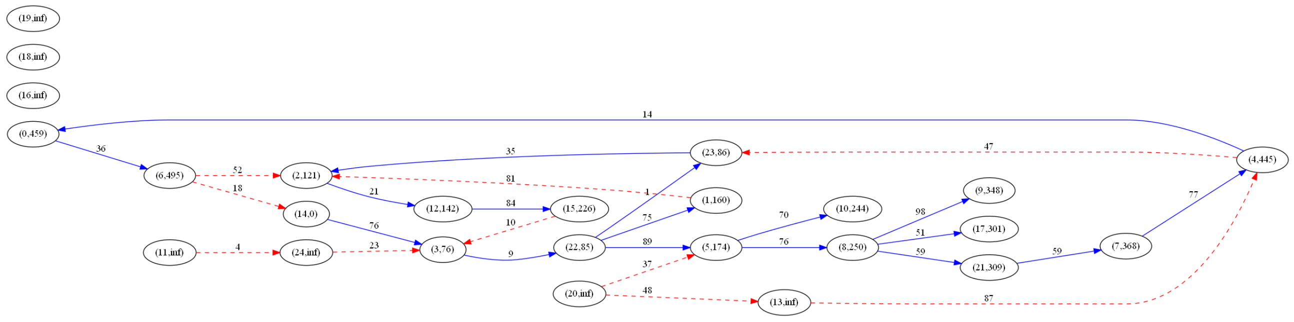


6.5 Minimal Graph Example - 5





6.6 Dijkstra's Algorithm : Source = 14


(19,inf)
(18,inf)
(16,inf)
(0,459)
(6,495)
(2,121)
(14,0)
(11,inf)
(24,inf)
(12,142)
(15,226)
(3,76)
(22,85)
(23,86)
(1,160)
(5,174)
(20,inf)
(13,inf)
(10,244)
(8,250)
(9,348)
(17,301)
(21,309)
(7,368)
(4,445)
14
36
52
18
35
81
21
84
10
76
4
23
9
4
75
89
37
48
70
76
98
51
59
59
77
47
87