COMP 6651

Ford Fulkerson Implementation

Submitted To: Dr. Thomas Fevens

Student Name: Raveena Choudhary

Student ID: 40232370

TABLE OF CONTENTS

1. Problem Description

This project focuses on implementing and assessing four augmenting path algorithms designed for the Ford-Fulkerson algorithm, aiming to determine the maximum flow within a graph. The goal is to scrutinize the efficiency and computational performance of these algorithms on randomly created source-sink graphs. The study seeks to offer insights into the merits and drawbacks of each algorithm, contributing to a deeper understanding of their roles in the optimization of network flow.

2. Implementation Details

2.1 Overview

The objective of the program is to run ford Fulkerson by finding augmenting paths using 4 different variants of Dijkstra's algorithm i.e., Shortest Augmenting Path (SAP), DFS Like, Maximum Capacity (MaxCap), Random. The main aim of Ford Fulkerson to calculate maximum flow on the graphs. These graphs are generated using Random Source-Sink Graph Generator algorithm described in next section. Generated graph can be cyclic depending on inputs such as number of nodes (n), distance(r) and upperCap (weight of edge), so appropriate handling of cycles are taken care of in all the required algorithms which can form cycles while execution.

2.2 Random Source-Sink Graph Generator

This algorithm generates directed version of a Euclidean neighbor graph. I assigned random (x, y) coordinates between 0 and 1 to each node and then created a list of vertices. To add edges for every pair of vertices u and v located within a distance r, a directed edge (u, v) or (v, u) is randomly selected and added to the graph, but not both. If the edge (u, v) is already present, avoided adding any edges in both (u,v) and (v,u) directions. For each edge in the graph, a capacity is chosen randomly as an integer within the range [1 .. upperCap]. I have created Adjacency list named as adjList to store vertices with their set of edges. Please find the below pseudocode for vertices, edges, and graph creation. For detailed code, please refer to RandomSourceSinkGraphGenerator.java class in Algorithms folder.

Here, vertex denotes id, x, y i.e. id of vertex, and x and y coordinates.

```
addVertices (n,adjList):
vertices = empty list
for v=0 to n:
    x = generateRandomNumberBetweenGivenRange(0, 1)
    y = generateRandomNumberBetweenGivenRange(0, 1)
    vertex = create new Vertex with id=v, x=x, y=y
    add vertex to vertices list
    add vertex v to the adjList
return vertices
```

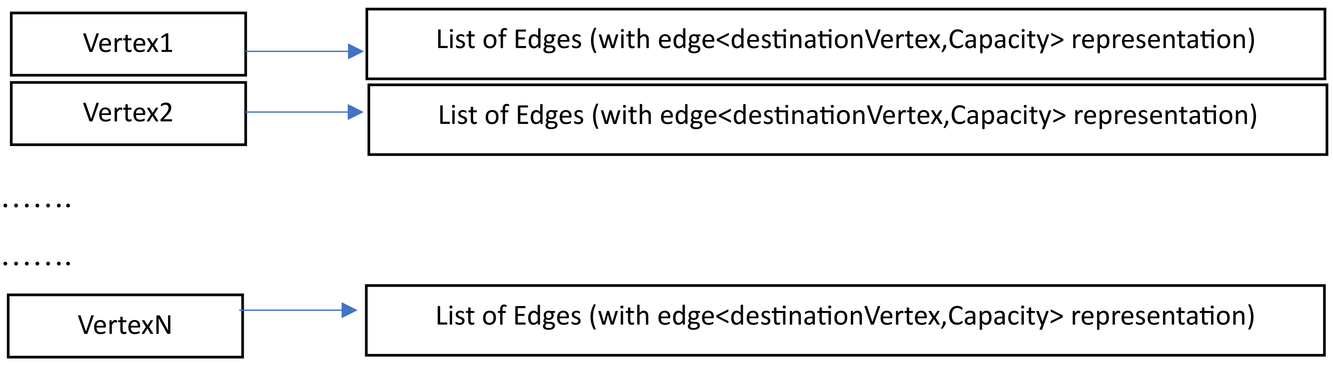
Here, DirectedEdge denotes source(u), destination(v) and capacity for an edge.

```
addEdge(n, r, upperCap, vertices, adjList):
edges = empty list
for u=0 to n:
    for v=0 to n:
        if u ≠ v:
            distance = getEuclideanDistanceBetweenVertices(u, v, vertices)
            if distance ≤ r:
                rand = generateRandomNumberBetweenGivenRange(0, 1)
                randomCapacity = getRandomCapacity(upperCap) //gets the random number for capacity between 1 to upperCap
        if rand < 0.5:
          if edge (u, v) does not exist in edges and edge (v, u) does not exist in edges:
            add directed edge (u, v) to edges with randomCapacity
            update neighbors in adjacency list for vertex u
        else:
          if edge (u, v) does not exist in edges and edge (v, u) does not exist in edges:
            add directed edge (v, u) to edges with randomCapacity
            update neighbors in adjacency list for vertex v
return edges
```

The below method generates the graph using set of vertices and edges generated by above methods. This graph is denoted as adjacency list.

```
generateGraph(n, r , upperCap):
adjList = createNewAdjList(n)  // create an empty adjacency list
vertices = addVertices(n, adjList)  // add vertices to the graph and update adjacency list
if n > 0 and r > 0 and upperCap > 0:
  addEdges(n, r, upperCap, vertices, adjList)  // add edges to the graph based on conditions
else:
  throw InvalidObjectException("n, r, and upperCap should be > 0")
return adjList  // return the populated adjacency list
```

Adjacency List Representation:


Vertex1
List of Edges (with edge<destinationVertex,Capacity> representation)
Vertex2
List of Edges (with edge<destinationVertex,Capacity> representation)
…….
…….
VertexN
List of Edges (with edge<destinationVertex,Capacity> representation)

2.3 Longest Acyclic Path

To identify the Longest Acyclic path, I applied the breadth-first search algorithm and stored the parent of each node in Parent map <target,parent> which later on I utilised to construct a valid path. The starting vertex (source) and target vertex (sink) of the computed path were then utilized as inputs for Augmenting Path algorithms to determine the augmenting paths for Ford Fulkerson algorithm. Please find below pseudocode for reference. For detailed code, please refer to LongestAcyclicPath.java in Algorithms folder.

```
findLongestAcyclicPath(n,graph):
selectedSources = empty set
longestPath = empty list
for i = 0 to n-1:
  source = getRandomSource(n, selectedSources) //randomly select source from n nodes.
```

```
   add source to selectedSources set //to avoid using the same source again
   distance = array of size n, initialized with -1 // maintain distance of all nodes from the source
   distance[source] = 0
   path = bfs(source, graph, distance)  // perform breadth-first search from the source
   if len(longestPath)<= len(path):
      clear all elements in longestPath
      add all elements from path to longestPath
return longestPath
```

When calculating the longest acyclic path in a graph, I want to maximize the sum of edge weights or distances along the path that is why I have taken Comparator to compare distances of 2 vertices. Also, since we can have vertices in a graph without any outgoing edges, to handle this case, I have assigned such vertex as v.id=-1 and capacity=-1 , since in my graph the vertices would start from 0 not from -1. So, if any vertex have v.id = -1 , that means we don’t need to consider such vertices during traversal.

```
bfs(source, graph, distance):
queue = PriorityQueue<Vertex> with custom comparator //Vertex represents a vertex with id and its distance
parentMap = empty HashMap
visitedNodes = empty list  // stores the visited nodes during bfs traversal
path = empty list
sourceVertex = Vertex object for the source with distance[source]
queue.insert(sourceVertex)
visitedNodes.add(source)
target = source
while queue is not empty:
   current = queue.poll()
   u = current.getId()
   if graph contains key u:
      for each neighbor in graph[u]:      //also checks that outgoing edge of a neighbor v.id not equal to -1
         v = neighbor.getVertex()
         if v == -1 //as inserting "-1" for invalid vertex(or vertices with no edges), need to check this.
            continue;
          if v is not in visitedNodes and distance[v] < distance[u] + 1:
               updatedVertex = create Vertex object for v with updated distance
               update distance[v] with the new value
               add updatedVertex to the queue
               add u to visitedNodes
               update parentMap[v] to u      //parentMap construct the path later
               set target to v
```

```
//traverse parentMap to construct the path
current = target
while current is not equal to source:
   add current to the path
   current = parentMap[current]
add source to the path
reverse the path
return path
```

2.4 Augmenting Path Algorithms

The below augmenting path algorithms are variants of Dijkstra’s algorithm.

2.4.1 Shortest Augmenting Path (SAP)

It runs Dijkstra's algorithm considering the edge lengths as unit distances rather than their actual Euclidean distances and extracting the node with minimum distance from the queue to explore the neighbours of that node. The main objective of this algorithm is to find shortest augmenting path in the given graph. This will be equivalent to running the BFS (Breadth first search) algorithm.

In this algorithm, the distance is updated for a node while exploration and then checked with existingDistance of the node, if updatedDistance < existingDistance, then update the node in queue with updatedDistance. Also, while creating a residual graph I have updated edge capacity of each vertex as = 1 , to avoid using their actual capacities in given graph. Here, also I have checked the vertex.id should not be equal to -1. Please find the below pseudocode for reference. For detailed code, please refer to ShortestAugmentingPath.java in Algorithms folder.

```
dijkstraToFindAugmentingPath(graph,source,sink,parentMap):
//parentMap is to construct the path
   create empty list of distances<vertex,distance>
   create empty priority queue Q (implemented to extract node<vertex, distance> with minimum distance)
   initialiseSingleSource(graph, source, distance);    //set source.d to 0 and other nodes to infinity
   for each vertex v in graph:
           Q.insert(node<v, distance>)
   while Q is not empty:
      currentVertex = Extract-Min() from Q //removes the min distance vertex from Q for exploration
      current = currentVertex.id
     //construct the path using parentMap
      if (current == sink):
         create empty list of currentPath
         target = sink
         while target!=source:
             add target to currentPath
             if (parentMap does not contain target as key):
                return null                  //not a valid path
             target = parentMap[target]
         add source to currentPath
      reverse the list of currenPath
      return currentPath
//traverse adj list for edges
    for each neighbor of current: //also checks that outgoing edge of a neighbor.id not equal to -1
        updatedDistance[neighbor] = distance[current] + 1; //edgeCapacity =1
        if updatedDistance[neighbor] < distance[neighbor]:
            distance[neighbor] = updatedDistance[neighbor]
            update neighbor in Q
              parentMap[neighbor] = current
  return null //no path found
```

2.4.2 DFS Like

It runs Dijkstra's algorithm by assigning decreasing counter value to each node when its distance is set to infinity. Subsequently, extracting minimum decreasing counter value node from queue to explore the neighbours of that node. This will be equivalent to running the DFS (Depth first search) algorithm.

In this algorithm, the decreasing counter is updated for a node while exploration and then checked with existingCounter of the node, if updatedCounter < exisitingCounter and existingCounter = infinity and the graph might form a cycle, to avoid that I have used the condition that if visitedNodes does not contain the neighbor then add the source to visitedNodes, then update the node in queue with updatedCounter. Please find the below pseudocode for reference. For detailed code, please refer to DFSLikeAlgo.java in Algorithms folder.

```
dijkstraToFindAugmentingPath(graph,source,sink,parentMap):
//parentMap is to construct the path
  create empty list of counters<vertex,decreasingCounter>
  create empty priority queue Q (implemented to extract node<vertex, decreasingCounter> with minimum decreasing counter value)
  initialiseSingleSource(graph, source, counters);    //set source.c to 2 * graph.size() and other nodes to infinity
  for each vertex v in graph:
        Q.insert(node<v, counter>)
  while Q is not empty:
    currentVertex = Extract-Min() from Q //removes the min counter value vertex from Q for exploration
    current = currentVertex.id
    dCounter = counters[source]      //decreasing counter
    //construct the path using parentMap
    if (current == sink):
      create empty list of currentPath
      target = sink
      while target!=source:
          add target to currentPath
          if (parentMap does not contain target as key):
              return null                //not a valid path
          target = parentMap[target]
      add source to currentPath
```

```
        reverse the list of currenPath
        return currentPath

//traverse adj list for edges
for each neighbor of current:           //also checks that outgoing edge of a neighbor.id not equal to -1
        updatedCounter[neighbor] = dCounter - 1;
        if updatedCounter [neighbor] < Counter[neighbor] and Counter[neighbor] = infinity and !visitedNodes.contains(neighborV):
            Counter[neighbor] = updatedCounter[neighbor]
            update neighbor in Q
            visitedNodes.add(current);
            parentMap[neighbor] = current
  return null //no path found
```

2.4.3 Maximum Capacity (MaxCap)

It runs Dijkstra's algorithm to find the augmenting path with maximum capacity. This is done by extracting the node with maximum capacity from the queue. In this algorithm, the updated capacity for the neighbor vertex (neighborV) is calculated as the minimum value between the current capacity of the edge and the capacity of the edge being considered. If updated capacity is greater than the exisitingCapacity, update the node in queue with maximum capacity. Please find the below pseudocode for reference. For detailed code, please refer to MaximumCapacity.java in Algorithms folder.

```
dijkstraToFindAugmentingPath(graph,source,sink,parentMap):
//parentMap is to construct the path
  create empty list of capacities<vertex,capacity>
  create empty priority queue Q (implemented to extract node<vertex,capacity> with maximum capacity)
  initialiseSingleSource(graph, source, capacities);    //set source.c to infinity and other nodes to -infinity
  for each vertex v in graph:
        Q.insert(node<v, capacity>)
  while Q is not empty:
    currentVertex = Extract-Max() from Q //removes the max capacity value vertex from Q for exploration
    current = currentVertex.id
    //construct the path using parentMap
    if (current == sink):
      create empty list of currentPath
      target = sink
      while target!=source:
          add target to currentPath
          if (parentMap does not contain target as key):
             return null                //not a valid path
          target = parentMap[target]
      add source to currentPath
      reverse the list of currenPath
```

```
        distanceOfSink = capacities.get(currentPath.get(currentPath.size()-1));    //used to verify t.d == capacity of the critical edge in augmenting path

        return currentPath

    //traverse adj list for edges

    for each neighbor of current:            //also checks that outgoing edge of a neighbor.id not equal to -1

        edgeCapacity = getEdgeCapacity(graph,current,neighbor)

        updatedCapacity[neighbor] = Math.min(capacities.get(current), edgeCapacity)

        if updatedCapacity[neighbor] > Capacity[neighbor] and !visitedNodes.contains(neighborV):

            Capacity[neighbor] = updatedCapacity[neighbor]

            update neighbor in Q

            visitedNodes.add(current);

            parentMap[neighbor] = current

  return null //no path found
```

2.4.4 Random

It runs Dijkstra's algorithm by assigning random counter value to each node. Subsequently, extracting minimum random counter value node from queue to explore the neighbours of that node.

In this algorithm, the random counter is updated for a node while exploration and then checked with existingRandomCounter of the node, if updatedRandomCounter < existingRandomCounter and addition of nodes does not form a cycle, then update the node in queue with updatedRandomCounter. Please find the below pseudocode for reference. For detailed code, please refer to RandomAlgo.java in Algorithms folder.

```
dijkstraToFindAugmentingPath(graph,source,sink,parentMap):

//parentMap is to construct the path

  create empty list of randomValues <vertex,randomKey>

  create empty priority queue Q (implemented to extract node<vertex, randomKey> with minimum random counter value)

  initialiseSingleSource(graph, source, randomValues);   //set source.c to some random value generated by Math.random() and other nodes to infinity

  for each vertex v in graph:

          Q.insert(node<v, randomValue>)

  while Q is not empty:

    currentVertex = Extract-Min() from Q //removes the min random counter value vertex from Q for exploration

    current = currentVertex.id

    //construct the path using parentMap

    if (current == sink):

        create empty list of currentPath

        target = sink

        while target!=source:

            add target to currentPath

            if (parentMap does not contain target as key):

                return null                //not a valid path

            target = parentMap[target]
        add source to currentPath
        reverse the list of currenPath
        return currentPath
//traverse adj list for edges
      for each neighbor of current:              //also checks that outgoing edge of a neighbor.id not equal to -1
          updatedRandomKey[neighbor] = Math.random()
          if updatedRandomKey[neighbor] < RandomKey[neighbor] and !visitedNodes.contains(neighborV):
            RandomKey[neighbor] = updatedRandomKey[neighbor]
            update neighbor in Q
            visitedNodes.add(current);
            parentMap[neighbor] = current
  return null //no path found
```

2.5 Ford Fulkerson

The goal of Ford-Fulkerson is to determine augmenting paths using algorithms like BFS. In this project, I have utilized Dijkstra's algorithm to compute augmenting paths. These paths are then supplied to Ford-Fulkerson to identify the maximum flow within the augmenting paths.

The basic idea behind the algorithm is to get the augmenting paths supplied by Dijkstra in this project, calculate max flow i.e., taking minimum capacity edge along the augmenting path c(p). Then, create a residual graph for further computation (finding more augmenting paths) by Dijkstra.

```
getMaxFlow(graph,source,sink):
maxflow = 0
empty list of parentMap to store <vertex,parent>
create residual graph from graph
create empty list of augmenting path
augmentingPath = dijkstraToFindAugmentingPath(residualGraph, source, sink,parentMap);
while(augmentingPath!=null):
    if(!this.augmentingPaths.contains(augmentingPath)):
          System.out.println(augmentingPath)
          this.augmentingPaths.add(augmentingPath)
  else:
      exit the loop
minCapacityAlongAugPath = infinity
target = sink
while(target!=source):
      u = parentMap[target]
      edgeCapacity = getEdgeCapacity(residualGraph,from,target)
      for each neighbor of u in residualGraph:
            if(neighbor == target):
                  minCapacityAlongAugPath = Math.min(minCapacityAlongAugPath, edgeCapacity)
```

```
        target = parentMap[target]
        maxFlow = maxflow + minCapacityAlongAugPath
        updatedResidualGraph=updateResidualGraph(residualGraph,augmentingPath,minCapacityAlongAugPath)
        remove all elements from augmenting path for new path
        remove all elements from parentMap for new path
        augmentingPath = dijkstraToFindAugmentingPath(updateResidualGraph, source, sink,parentMap)
return maxflow
```

Metrics calculation implementation for all the classes are implemented in common class named as "Metrics.java". Also, other commonly used methods are all implemented in the classes placed in util.

2.6 Program Execution- Local and ENCS Server

Prerequisites

1) Supported Java Version JDK 1.8

2) Run java -version to check java version in computation.encs.concordia.ca server.

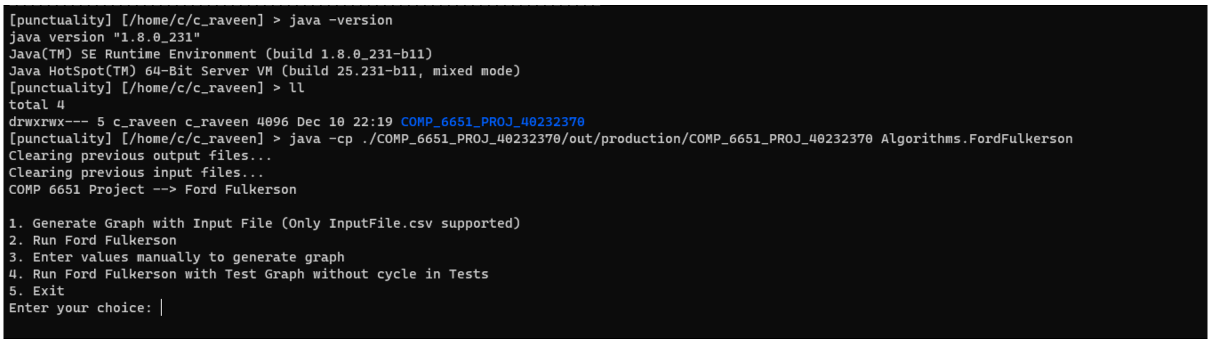
Execution

1) Unzip the file submitted on Moodle and Copy project COMP_6651_PROJ_40232370 on computation.encs.concordia.ca server.

2) From current directory (run step 3)

3) Run following cmd:

java -cp ./COMP_6651_PROJ_40232370/out/production/COMP_6651_PROJ_40232370 Algorithms.FordFulkerson



```
[punctuality] [/home/c/c_raveen] > java -version
java version "1.8.0_231"
Java(TM) SE Runtime Environment (build 1.8.0_231-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.231-b11, mixed mode)
[punctuality] [/home/c/c_raveen] > ll
total 4
drwxrwx--- 5 c_raveen c_raveen 4096 Dec 10 22:19 COMP_6651_PROJ_40232370
[punctuality] [/home/c/c_raveen] > java -cp ./COMP_6651_PROJ_40232370/out/production/COMP_6651_PROJ_40232370 Algorithms.FordFulkerson
Clearing previous output files...
Clearing previous input files...
COMP 6651 Project --> Ford Fulkerson

1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: |
```

Fig 3: Server Execution

4) User should see the below menu:

COMP 6651 Project --> Ford Fulkerson

1. Generate Graph with Input File (Only InputFile.csv supported)

2. Run Ford Fulkerson

3. Enter values manually to generate graph

4. Run Ford Fulkerson with Test Graph without cycle in Tests

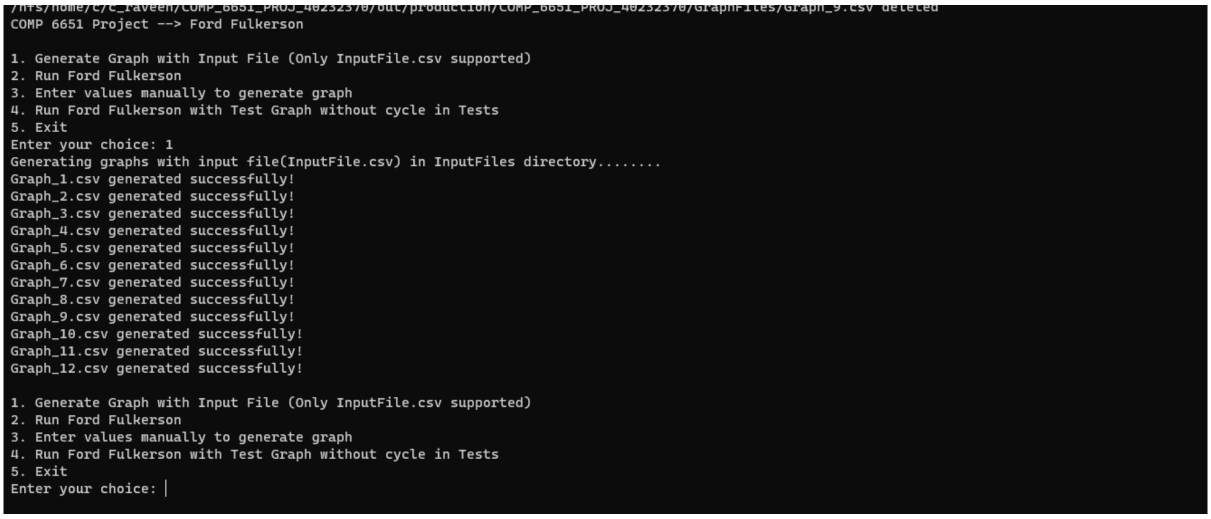
5. Exit

Enter your choice:

5) Select any option from available Options 1 to 5 as per the use case.

6) *Generate Graph with Input File (Only InputFile.csv supported):*

This method will read the input file line by line and generate the random source-sink graph files for each set of n, r and upperCap values mentioned in csv file, comma separated. Further, these graph files are stored in GraphFiles folder with naming convention as Graph_graphNumber.csv

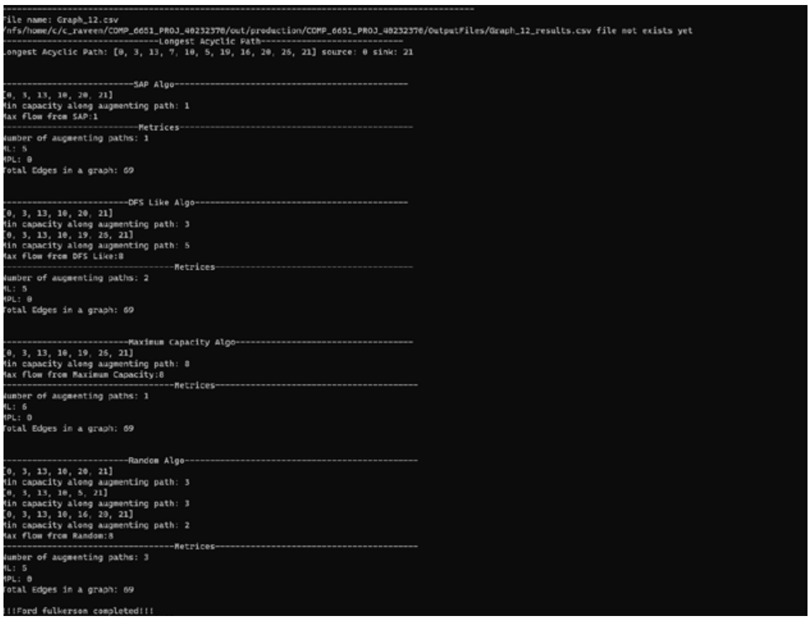


```
/nfs/home/c/c_raveen/COMP_6651_PROJ_40232370/out/production/COMP_6651_PROJ_40232370/GraphFiles/Graph_9.csv deleted
COMP 6651 Project --> Ford Fulkerson

1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: 1
Generating graphs with input file(InputFile.csv) in InputFiles directory........
Graph_1.csv generated successfully!
Graph_2.csv generated successfully!
Graph_3.csv generated successfully!
Graph_4.csv generated successfully!
Graph_5.csv generated successfully!
Graph_6.csv generated successfully!
Graph_7.csv generated successfully!
Graph_8.csv generated successfully!
Graph_9.csv generated successfully!
Graph_10.csv generated successfully!
Graph_11.csv generated successfully!
Graph_12.csv generated successfully!

1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: |
```

Fig 4: Generated Graph Files

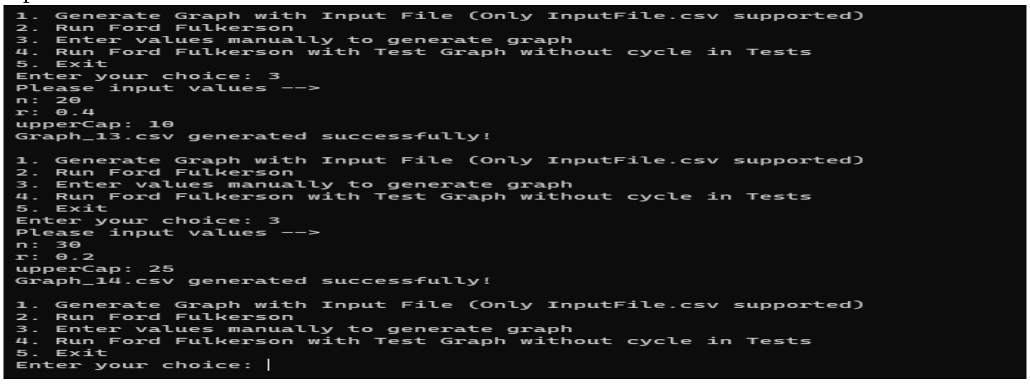
7) *Run Ford Fulkerson:*

This method will create directed acyclic graph and then provide this as input to longest acyclic path, shortest augmenting path, DFS Like, Maximum Capacity, Random algorithms. All these algorithms will be executed within this method and print the results on console, and output file will be generated for each graph in OutputFiles folder with naming convention as Graph_graphNumber_results.csv



```
---------------------------------------------------------------------------------------
ile name: Graph_12.csv
nfs/home/c/c_raveen/COMP_6651_PROJ_40232370/out/production/COMP_6651_PROJ_40232370/OutputFiles/Graph_12_results.csv file not exists yet
-----------------------------Longest Acyclic Path-----------------------------
ongest Acyclic Path: [0, 3, 13, 7, 10, 5, 19, 16, 20, 26, 21] source: 0 sink: 21


------------------------SAP Algo-----------------------------------------------
[0, 3, 13, 10, 20, 21]
in capacity along augmenting path: 1
ax flow from SAP:1
-------------------------Metrices----------------------------------------------
umber of augmenting paths: 1
L: 5
PL: 0
otal Edges in a graph: 69


------------------------DFS Like Algo------------------------------------------
[0, 3, 13, 10, 20, 21]
in capacity along augmenting path: 3
[0, 3, 13, 10, 19, 26, 21]
in capacity along augmenting path: 5
ax flow from DFS Like:8
---------------------------------Metrices--------------------------------------
umber of augmenting paths: 2
L: 5
PL: 0
otal Edges in a graph: 69


------------------------Maximum Capacity Algo----------------------------------
[0, 3, 13, 10, 19, 26, 21]
in capacity along augmenting path: 8
ax flow from Maximum Capacity:8
---------------------------------Metrices---------------------------------------
umber of augmenting paths: 1
L: 6
PL: 0
otal Edges in a graph: 69


------------------------Random Algo---------------------------------------------
[0, 3, 13, 10, 20, 21]
in capacity along augmenting path: 3
[0, 3, 13, 10, 5, 21]
in capacity along augmenting path: 3
[0, 3, 13, 10, 16, 20, 21]
in capacity along augmenting path: 2
ax flow from Random:8
---------------------------------Metrices---------------------------------------
umber of augmenting paths: 3
L: 5
PL: 0
otal Edges in a graph: 69

!!!Ford fulkerson completed!!!
```

8) *Enter values manually to generate graph:* Run program with different set of values(n,r,upperCap), and Option 2 to run Ford fulkerson.



```
1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: 3
Please input values -->
n: 20
r: 0.4
upperCap: 10
Graph_13.csv generated successfully!

1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: 3
Please input values -->
n: 30
r: 0.2
upperCap: 25
Graph_14.csv generated successfully!

1. Generate Graph with Input File (Only InputFile.csv supported)
2. Run Ford Fulkerson
3. Enter values manually to generate graph
4. Run Ford Fulkerson with Test Graph without cycle in Tests
5. Exit
Enter your choice: |
```

9) *Run Ford Fulkerson with Test Graph without cycle in Tests*: Runs TestGraphWithoutCycle.java, test graph is present under Tests Folder.

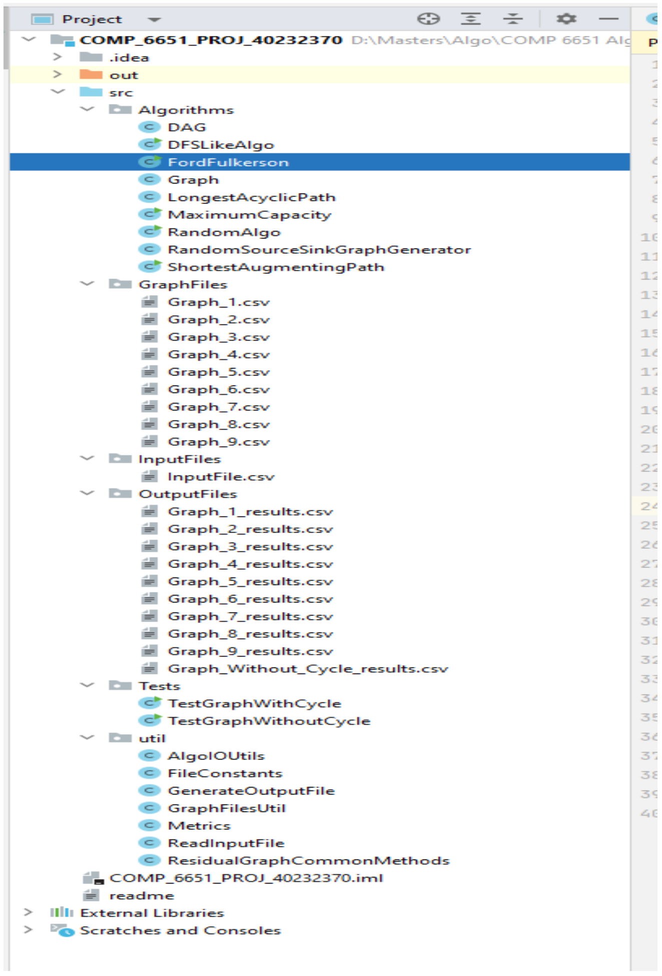
10) *Exit*: Exits the program.

11) You have successfully completed the execution!

12) Program execution on server is also working for me, by following the steps mentioned above.

13) All the results mentioned by below are taken from outputFiles generated while the execution of the program on the server.

Project structure:



```
Project
COMP_6651_PROJ_40232370  D:\Masters\Algo\COMP 6651 Alg
  .idea
  out
  src
    Algorithms
      DAG
      DFSLikeAlgo
      FordFulkerson
      Graph
      LongestAcyclicPath
      MaximumCapacity
      RandomAlgo
      RandomSourceSinkGraphGenerator
      ShortestAugmentingPath
    GraphFiles
      Graph_1.csv
      Graph_2.csv
      Graph_3.csv
      Graph_4.csv
      Graph_5.csv
      Graph_6.csv
      Graph_7.csv
      Graph_8.csv
      Graph_9.csv
    InputFiles
      InputFile.csv
    OutputFiles
      Graph_1_results.csv
      Graph_2_results.csv
      Graph_3_results.csv
      Graph_4_results.csv
      Graph_5_results.csv
      Graph_6_results.csv
      Graph_7_results.csv
      Graph_8_results.csv
      Graph_9_results.csv
      Graph_Without_Cycle_results.csv
    Tests
      TestGraphWithCycle
      TestGraphWithoutCycle
    util
      AlgoIOUtils
      FileConstants
      GenerateOutputFile
      GraphFilesUtil
      Metrics
      ReadInputFile
      ResidualGraphCommonMethods
    COMP_6651_PROJ_40232370.iml
    readme
External Libraries
Scratches and Consoles
```

3. Implementation Correctness

3.1 Testing with tool

I have used available online tool [1] to run ford Fulkerson on the given graph. Please find the below screenshots for reference.

Input ➔ Sample Graph: taken from TestGraphWithoutCycle.java file.

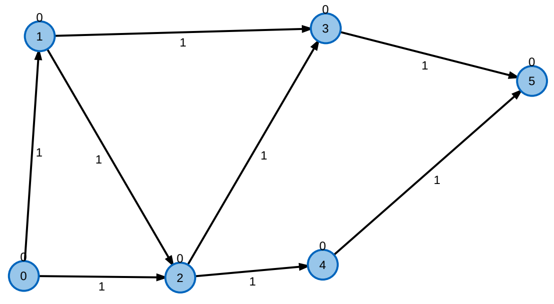


Fig1: Graph with unit distance edge capacities

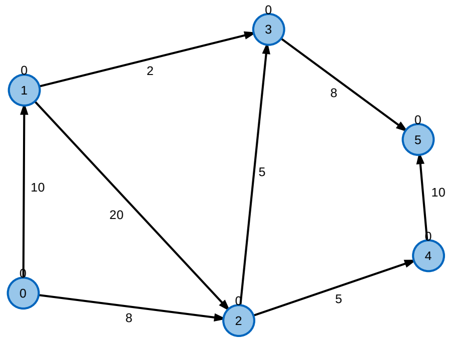
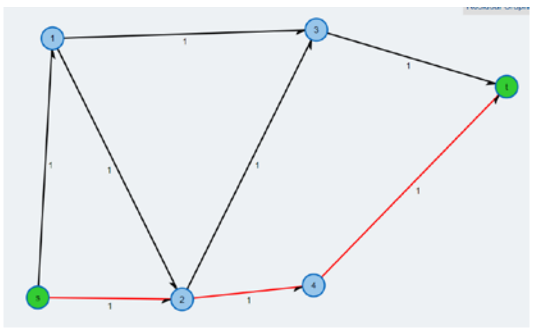


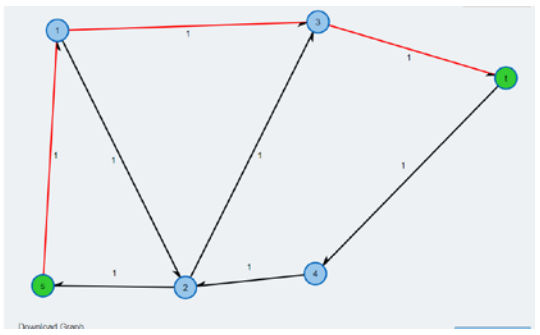
Fig2: Graph with edge capacities

Simulations:

For Fig1,

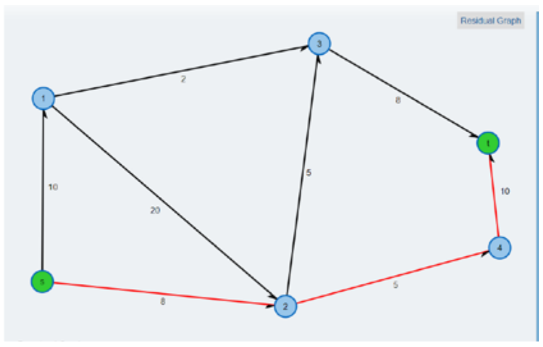


First iteration: [0,2,4,5]

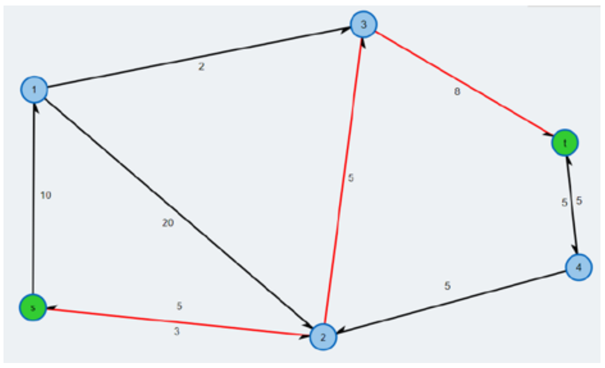


Second iteration: [0,1,3,5]

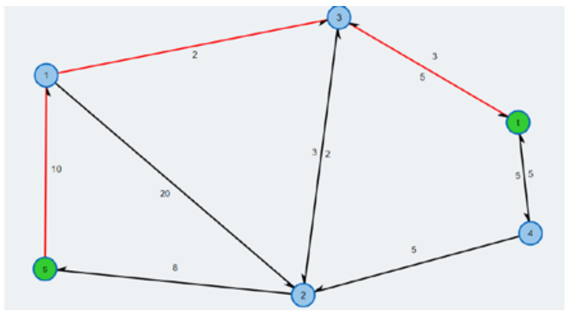
For Fig2,



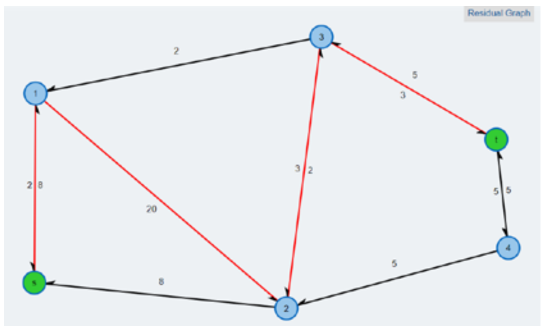
First iteration: [0,2,4,5]



Second iteration: [0,2,3,5]

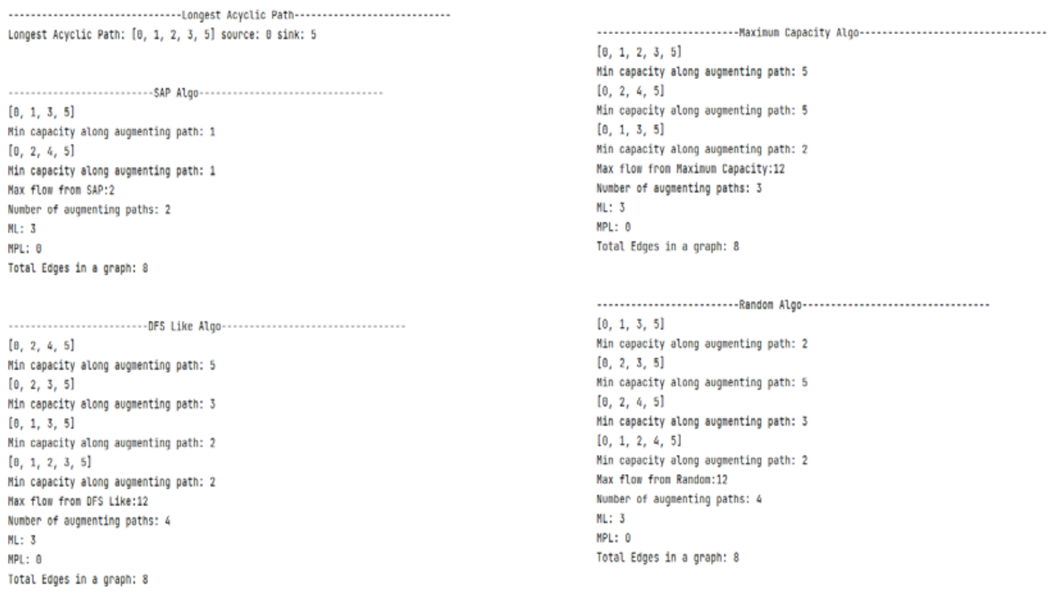


First iteration: [0,1,3,5]



Second iteration: [0,1,2,3,5]

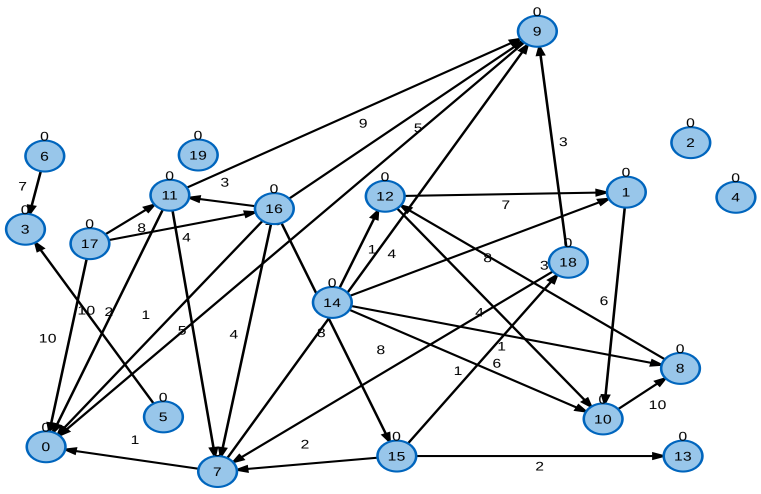
These results of augmenting paths are matching with the program execution. Please find the below screenshot for reference.

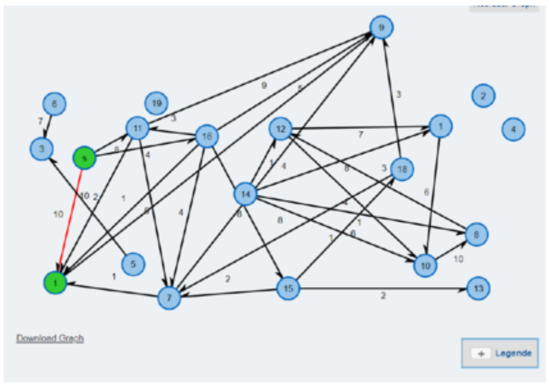


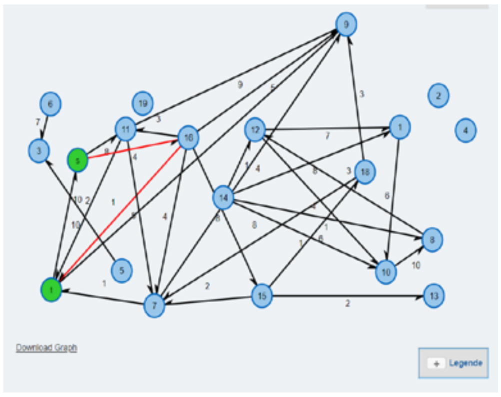
```
------------------------------Longest Acyclic Path---------------------------
Longest Acyclic Path: [0, 1, 2, 3, 5] source: 0 sink: 5


------------------------SAP Algo--------------------------------
[0, 1, 3, 5]
Min capacity along augmenting path: 1
[0, 2, 4, 5]
Min capacity along augmenting path: 1
Max flow from SAP:2
Number of augmenting paths: 2
ML: 3
MPL: 0
Total Edges in a graph: 8


-----------------------DFS Like Algo--------------------------------
[0, 2, 4, 5]
Min capacity along augmenting path: 5
[0, 2, 3, 5]
Min capacity along augmenting path: 3
[0, 1, 3, 5]
Min capacity along augmenting path: 2
[0, 1, 2, 3, 5]
Min capacity along augmenting path: 2
Max flow from DFS Like:12
Number of augmenting paths: 4
ML: 3
MPL: 0
Total Edges in a graph: 8
```

```
------------------------Maximum Capacity Algo--------------------------------
[0, 1, 2, 3, 5]
Min capacity along augmenting path: 5
[0, 2, 4, 5]
Min capacity along augmenting path: 5
[0, 1, 3, 5]
Min capacity along augmenting path: 2
Max flow from Maximum Capacity:12
Number of augmenting paths: 3
ML: 3
MPL: 0
Total Edges in a graph: 8


------------------------Random Algo--------------------------------
[0, 1, 3, 5]
Min capacity along augmenting path: 2
[0, 2, 3, 5]
Min capacity along augmenting path: 5
[0, 2, 4, 5]
Min capacity along augmenting path: 3
[0, 1, 2, 4, 5]
Min capacity along augmenting path: 2
Max flow from Random:12
Number of augmenting paths: 4
ML: 3
MPL: 0
Total Edges in a graph: 8
```

Similarly, I performed the execution on graph with 20 nodes, with values n=20, r=0.5 and upperCap=10.

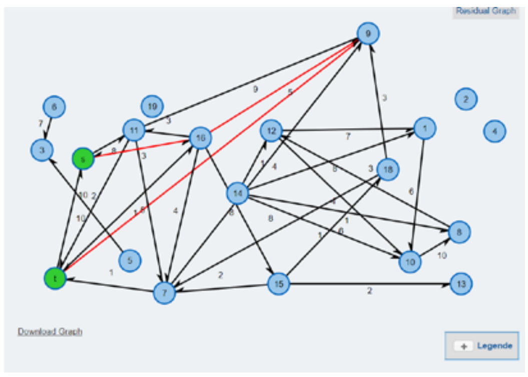


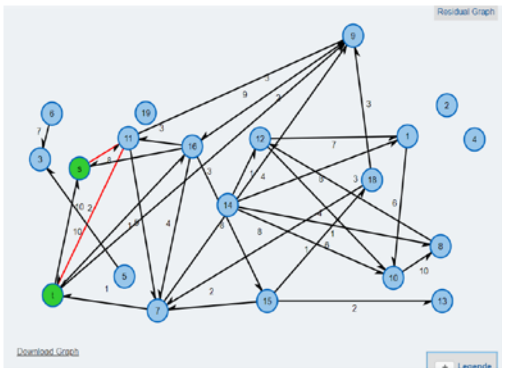




First iteration: [17,0]

Second iteration: [17,16,0]



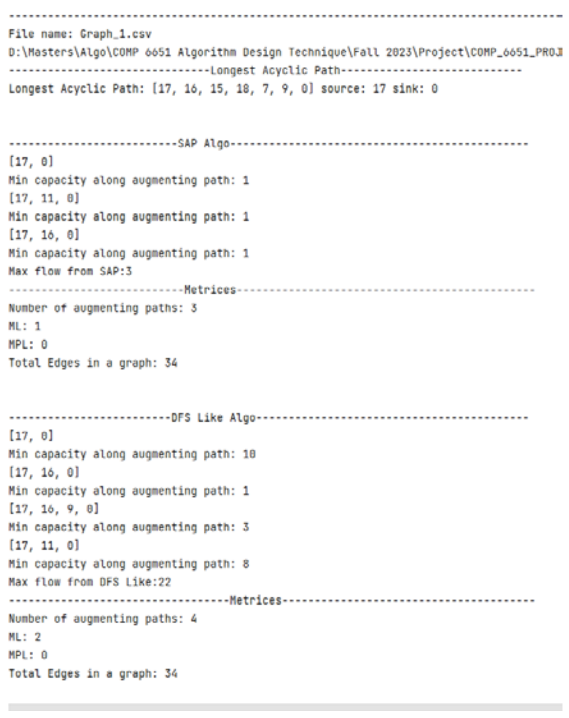


Third iteration: [17,16,9,0]

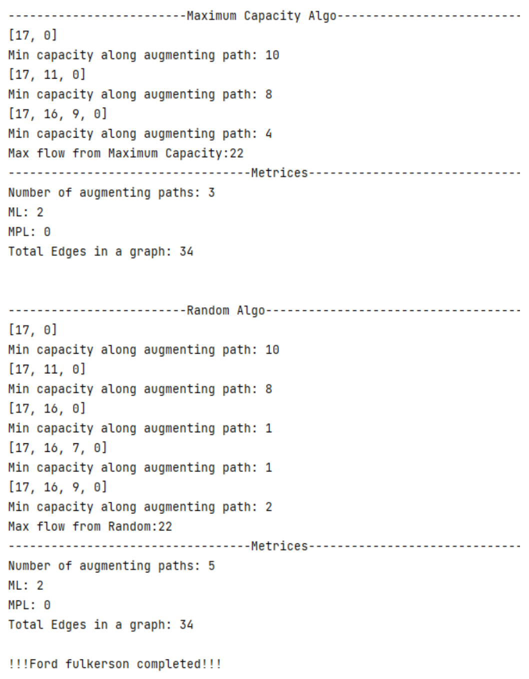
Fourth iteration: [17,11,0]

Random Algo gives one more additional path: [17, 16, 7, 0] which is also a valid path

These results of augmenting paths are matching with the program execution. Please find the below screenshot for reference.



```
-------------------------------------------------------------------------------------
File name: Graph_1.csv
D:\Masters\Algo\COMP 6651 Algorithm Design Technique\Fall 2023\Project\COMP_6651_PROJ
------------------------------Longest Acyclic Path---------------------------
Longest Acyclic Path: [17, 16, 15, 18, 7, 9, 0] source: 17 sink: 0


-------------------------SAP Algo---------------------------------------------
[17, 0]
Min capacity along augmenting path: 1
[17, 11, 0]
Min capacity along augmenting path: 1
[17, 16, 0]
Min capacity along augmenting path: 1
Max flow from SAP:3
--------------------------Metrices---------------------------------------------
Number of augmenting paths: 3
ML: 1
MPL: 0
Total Edges in a graph: 34


------------------------DFS Like Algo-----------------------------------------
[17, 0]
Min capacity along augmenting path: 10
[17, 16, 0]
Min capacity along augmenting path: 1
[17, 16, 9, 0]
Min capacity along augmenting path: 3
[17, 11, 0]
Min capacity along augmenting path: 8
Max flow from DFS Like:22
---------------------------------Metrices-------------------------------------
Number of augmenting paths: 4
ML: 2
MPL: 0
Total Edges in a graph: 34
```



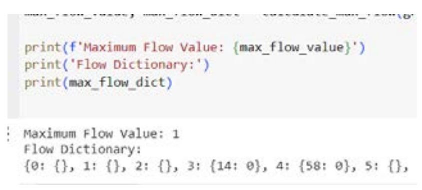
```
-------------------------Maximum Capacity Algo--------------------------
[17, 0]
Min capacity along augmenting path: 10
[17, 11, 0]
Min capacity along augmenting path: 8
[17, 16, 9, 0]
Min capacity along augmenting path: 4
Max flow from Maximum Capacity:22
---------------------------------Metrices-----------------------------
Number of augmenting paths: 3
ML: 2
MPL: 0
Total Edges in a graph: 34


------------------------Random Algo-----------------------------------
[17, 0]
Min capacity along augmenting path: 10
[17, 11, 0]
Min capacity along augmenting path: 8
[17, 16, 0]
Min capacity along augmenting path: 1
[17, 16, 7, 0]
Min capacity along augmenting path: 1
[17, 16, 9, 0]
Min capacity along augmenting path: 2
Max flow from Random:22
---------------------------------Metrices-----------------------------
Number of augmenting paths: 5
ML: 2
MPL: 0
Total Edges in a graph: 34

!!!Ford fulkerson completed!!!
```

3.2 Testing MaxFlow with NetworkX library

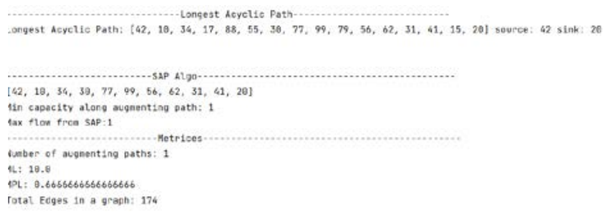
I have done this to check maxflow value. The program is returning max flow as given by networkX in python. Please find few screenshots below:

NetworkX results

My program execution results

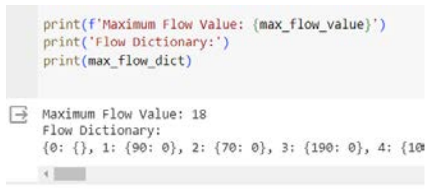


```
print(f'Maximum Flow Value: {max_flow_value}')
print('Flow Dictionary:')
print(max_flow_dict)

Maximum Flow Value: 1
Flow Dictionary:
{0: {}, 1: {}, 2: {}, 3: {14: 0}, 4: {58: 0}, 5: {},
```

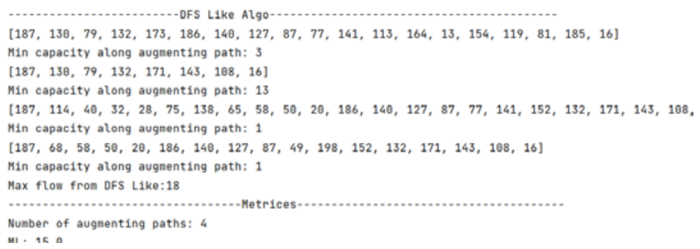


```
------------------------------Longest Acyclic Path---------------------------
.ongest Acyclic Path: [42, 10, 34, 17, 88, 55, 30, 77, 99, 79, 56, 62, 31, 41, 15, 20] source: 42 sink: 20


-------------------------SAP Algo---------------------------------------------
[42, 10, 34, 30, 77, 99, 56, 62, 31, 41, 20]
Min capacity along augmenting path: 1
Max flow from SAP:1
--------------------------Metrices---------------------------------------------
Number of augmenting paths: 1
ML: 10.0
MPL: 0.6666666666666666
Total Edges in a graph: 174
```

Graph_1.csv



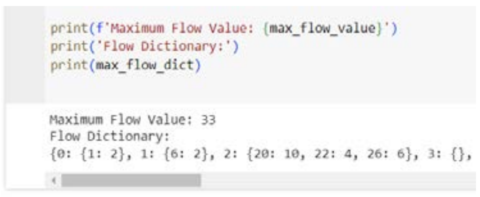
```
print(f'Maximum Flow Value: {max_flow_value}')
print('Flow Dictionary:')
print(max_flow_dict)

Maximum Flow Value: 18
Flow Dictionary:
{0: {}, 1: {90: 0}, 2: {70: 0}, 3: {190: 0}, 4: {10
```



```
-------------------------DFS Like Algo-----------------------------------------
[187, 130, 79, 132, 173, 186, 140, 127, 87, 77, 141, 113, 164, 13, 154, 119, 81, 185, 16]
Min capacity along augmenting path: 3
[187, 130, 79, 132, 171, 143, 108, 16]
Min capacity along augmenting path: 13
[187, 114, 40, 32, 28, 75, 138, 65, 58, 50, 20, 186, 140, 127, 87, 77, 141, 152, 132, 171, 143, 108,
Min capacity along augmenting path: 1
[187, 68, 58, 50, 20, 186, 140, 127, 87, 49, 198, 152, 132, 171, 143, 108, 16]
Min capacity along augmenting path: 1
Max flow from DFS Like:18
----------------------------------Metrices-------------------------------------
Number of augmenting paths: 4
ML: 15.0
```

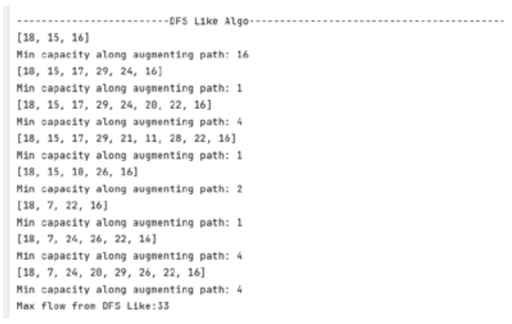
Graph_6.csv

With Proposed set of values, taken as test



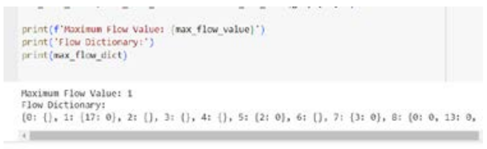
```
print(f'Maximum Flow Value: {max_flow_value}')
print('Flow Dictionary:')
print(max_flow_dict)
```

```
Maximum Flow Value: 33
Flow Dictionary:
{0: {1: 2}, 1: {6: 2}, 2: {20: 10, 22: 4, 26: 6}, 3: {},
```



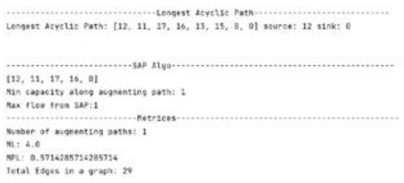
```
------------------------DFS Like Algo------------------------------------------
[18, 15, 16]
Min capacity along augmenting path: 16
[18, 15, 17, 29, 24, 16]
Min capacity along augmenting path: 1
[18, 15, 17, 29, 24, 20, 22, 16]
Min capacity along augmenting path: 4
[18, 15, 17, 29, 21, 11, 28, 22, 16]
Min capacity along augmenting path: 1
[18, 15, 10, 26, 16]
Min capacity along augmenting path: 2
[18, 7, 22, 16]
Min capacity along augmenting path: 1
[18, 7, 24, 26, 22, 16]
Min capacity along augmenting path: 4
[18, 7, 24, 20, 29, 26, 22, 16]
Min capacity along augmenting path: 4
Max flow from DFS Like:33
```

Graph 14.csv



```
print(f'Maximum Flow Value: {max_flow_value}')
print('Flow Dictionary:')
print(max_flow_dict)
```

```
Maximum Flow Value: 1
Flow Dictionary:
{0: {}, 1: {17: 0}, 2: {}, 3: {}, 4: {}, 5: {2: 0}, 6: {}, 7: {3: 0}, 8: {0: 0, 13: 0,
```



```
-------------------------------Longest Acyclic Path----------------------------
Longest Acyclic Path: [12, 11, 17, 16, 13, 15, 8, 0] source: 12 sink: 0

-------------------------SAP Algo----------------------------------------------
[12, 11, 17, 16, 0]
Min capacity along augmenting path: 1
Max flow from SAP:1
--------------------------Metrices---------------------------------------------
Number of augmenting paths: 1
ML: 4.0
MPL: 0.5714285714285714
Total Edges in a graph: 29
```

Graph 13.csv

4. Results

Simulations I:

Graph_1_results.csv:

n = 100, r = 0.2, upperCap = 2

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	100	0.2	2	1	10.0	0.6666	174
DFS	100	0.2	2	1	12.0	0.8	174
MaxCap	100	0.2	2	1	11.0	0.7333	174
Random	100	0.2	2	1	10.0	0.6666	174

Graph_2_results.csv:

n = 200, r = 0.2, upperCap = 2

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	200	0.2	2	2	4.0	0.1538	611
DFS	200	0.2	2	3	6.0	0.2307	611
MaxCap	200	0.2	2	3	4.0	0.1538	611
Random	200	0.2	2	3	5.0	0.1923	611

Graph_3_results.csv:

n = 100, r = 0.3, upperCap = 2

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	100	0.3	2	1	11.0	0.3666	321
DFS	100	0.3	2	1	16.0	0.5333	321
MaxCap	100	0.3	2	1	11.0	0.3666	321
Random	100	0.3	2	1	11.0	0.3666	321

<u>*Graph_4_results.csv:*</u>

n = 200, r = 0.3, upperCap = 2

Algorithm	**n**	**r**	**upperCap**	**paths**	**ML**	**MPL**	**total edges**
SAP	200	0.3	2	5	10.0	0.1162	1183
DFS	200	0.3	2	5	20.0	0.2325	1183
MaxCap	200	0.3	2	5	11.0	0.1279	1183
Random	200	0.3	2	5	11.0	0.1279	1183

<u>*Graph_5_results.csv:*</u>

n = 100, r = 0.2, upperCap = 50

Algorithm	**n**	**r**	**upperCap**	**paths**	**ML**	**MPL**	**total edges**
SAP	100	0.2	50	1	3.0	0.375	164
DFS	100	0.2	50	1	3.0	0.375	164
MaxCap	100	0.2	50	1	3.0	0.375	164
Random	100	0.2	50	1	3.0	0.375	164

<u>*Graph_6_results.csv:*</u>

n = 200, r = 0.2, upperCap = 50

Algorithm	**n**	**r**	**upperCap**	**paths**	**ML**	**MPL**	**total edges**
SAP	200	0.2	50	2	11.0	0.4782	571
DFS	200	0.2	50	4	15.0	0.6521	571
MaxCap	200	0.2	50	2	9.0	0.3913	571
Random	200	0.2	50	3	8.0	0.3478	571

<u>*Graph_7_results.csv:*</u>

n = 100, r = 0.3, upperCap = 50

Algorithm	**n**	**r**	**upperCap**	**paths**	**ML**	**MPL**	**total edges**
SAP	100	0.3	50	1	14.0	0.6086	301
DFS	100	0.3	50	3	17.0	0.7391	301
MaxCap	100	0.3	50	1	16.0	0.6956	301
Random	100	0.3	50	2	14.0	0.6086	301

<u>*Graph_8_results.csv:*</u>

n = 200, r = 0.3, upperCap = 50

Algorithm	**n**	**r**	**upperCap**	**paths**	**ML**	**MPL**	**total edges**
SAP	200	0.3	50	4	3.0	0.0337	1300
DFS	200	0.3	50	4	3.0	0.0337	1300
MaxCap	200	0.3	50	4	4.0	0.0449	1300
Random	200	0.3	50	4	3.0	0.0337	1300

Simulations II:

<u>*Graph_9_results.csv:*</u>

I have increased the distance value(r) to get more denser graph as I was expecting that algorithms should perform better and produce more augmenting paths which results into augmentation of maxflow through the edges.

n = 100, r = 0.5, upperCap = 70

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	100	0.5	70	5	6.0	0.075	780
DFS	100	0.5	70	33	12.0	0.15	780
MaxCap	100	0.5	70	6	10.0	0.125	780
Random	100	0.5	70	22	9.0	0.1125	780

Graph_10_results.csv:

I have increased the number of vertices(n) and distance value(r) for graph to compare with previous simulation results. To understand the combination of larger and denser graphs, affecting the maxflow through the graph.

n = 200, r = 0.5, upperCap = 70

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	200	0.5	70	6	6.0	0.0344	3181
DFS	200	0.5	70	99	20.0	0.1149	3181
MaxCap	200	0.5	70	13	11.0	0.0632	3181
Random	200	0.5	70	54	10.0	0.0574	3181

Tests:

Graph_13_results.csv:

n = 20, r = 0.4, upperCap = 10

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	20	0.4	10	1	4.0	0.5714	29
DFS	20	0.4	10	1	4.0	0.5714	29
MaxCap	20	0.4	10	1	4.0	0.5714	29
Random	20	0.4	10	1	4.0	0.5714	29

Graph_14_results.csv:

n = 30, r = 0.8, upperCap = 25

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	30	0.2	25	4	3.0	0.1666	159
DFS	30	0.2	25	8	5.0	0.2777	159
MaxCap	30	0.2	25	4	5.0	0.2777	159
Random	30	0.2	25	7	3.0	0.1666	159

5. Conclusions

- As per the simulations, please find the below conclusions:
  We can use **Shortest Augmenting Path (SAP)** for both larger and denser graphs, because it finds the shortest augmenting paths which would result into faster execution of algorithm and calculation of maxflow. This algorithm has faster convergence to the max flow.

  DFS-like: It explores different augmenting paths which may increase the flow. So, as per the results in case of denser graphs, considerably more augmenting paths are produced by DFS which might result into faster convergence to the maximum flow.

  Maximum Capacity (MaxCap): It focuses on finding the augmenting path with maximum capacity, and in denser graphs it is increasing number of augmenting paths, as it might be a possibility to explore another path with same maximum capacity. This results into reduction in mean length. For larger graphs, number of augmenting paths produced are comparatively less so it can affect the convergence of the algorithm and may result in slower progress towards finding the maximum flow.

Random: It has different variations, as sometimes it produces the same number of augmenting paths as produced by one of the other algorithms or completely different existing path not traversed by any other algorithm. So, this less deterministic, may be few paths might be enough to increase the flow. Test iterations in test to prove correctness section shows this behavior.

- If we increase the distance (r) i.e., graphs are denser, then I would suggest DFS Like and SAP, MaxCap algorithms
- If we increase the number of nodes(n) i.e., larger graphs, then I would suggest Shortest Augmenting Path algorithm and DFSLike.

6. References

1. https://algorithms.discrete.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html

2. https://www.geeksforgeeks.org/longest-path-in-a-directed-acyclic-graph-dynamic-programming/

3. https://www.baeldung.com/cs/network-flow-edmonds-karp-algorithm

4. https://www.quora.com/How-can-one-modify-BFS-to-compute-longest-paths

5. https://stackoverflow.com/questions/61740126/find-longest-paths-for-each-node-in-a-dag-from-start-node

6. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

7. https://stackoverflow.com/questions/8922060/how-to-trace-the-path-in-a-breadth-first-search

8. https://www.quora.com/Can-I-get-all-the-shortest-paths-from-source-node-to-destination-in-graph-using-bfs

9.https://www.sciencedirect.com/science/article/pii/0166218X9390148H#:~:text=The%20maximum%20capacity%20augmentation%20algorithm%20is%20a%20variant%20of%20the,maximum%20increase%20in%20flow%20value.

10. https://networkx.org/documentation/stable/reference/index.html