Project Report
COMP 6651
Algorithm Design Techniques

Submitted by:

Sushant Sinha

Student ID: 40261753

INDEX

1. Problem Description

In this project we aim to implement the Ford-Fulkerson method of finding the maximum flow in a given flow network.

For implementing the Ford-Fulkerson algorithm, we will need to find the augmenting path(if exists) in the residual network. Instead of using a single method to find the augmenting, we will be varying the algorithm in order to test multiple metrics.

Below is the pseudocode of the Ford-Fulkerson algorithm as mentioned in the project description:

```
FORD-FULKERSON-METHOD($G, s, t$)
    1    initialize flow $f$ to 0
    2    while there exists an augmenting path $p$ in the residual network $G_f$
    3        augment flow $f$ along $p$
    4    return $f$
```

Here, we will be modifying the line 2 in the above pseudocode with the following 4 algorithms:

1) Shortest Augmenting Path (SAP)
2) DFS-like
3) Maximum Capacity (MaxCap) and
4) Random

The graphs will be kept constant while varying the algorithms so as to check their relative performance within the same complexity/density of the graph.

We'll then tabulate the observations to have a comparative study, and build up a conclusion from it.

P.T.O

2. Implementation Details

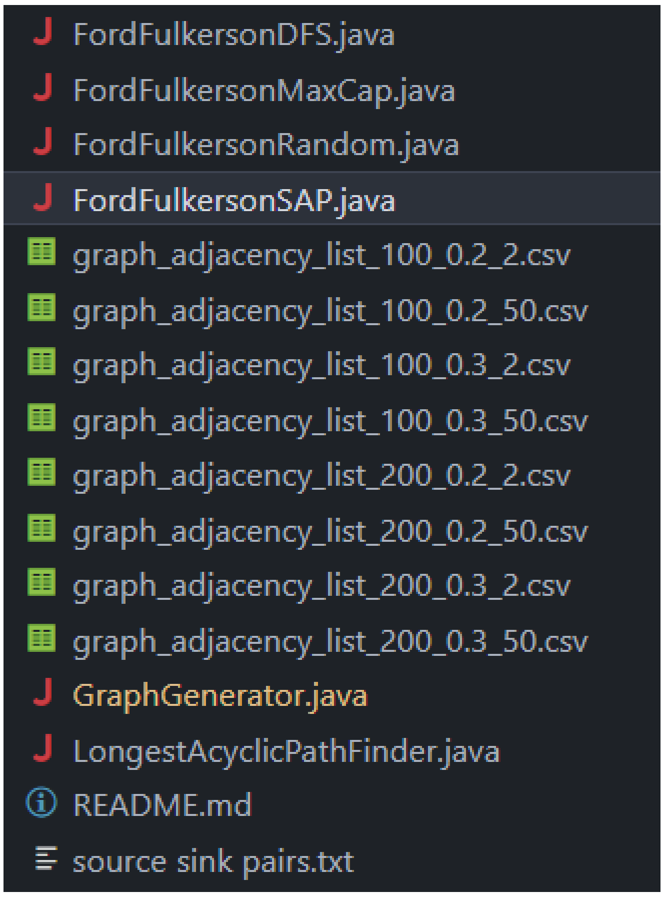
2.1) Technologies used

I’ve used Java as the programming language to implement the entire project ***without*** any external libraries.

The generated graphs are stored in a CSV file.

2.2) Directory structure

Below is the directory structure of the entire project repo:



```
FordFulkersonDFS.java
FordFulkersonMaxCap.java
FordFulkersonRandom.java
FordFulkersonSAP.java
graph_adjacency_list_100_0.2_2.csv
graph_adjacency_list_100_0.2_50.csv
graph_adjacency_list_100_0.3_2.csv
graph_adjacency_list_100_0.3_50.csv
graph_adjacency_list_200_0.2_2.csv
graph_adjacency_list_200_0.2_50.csv
graph_adjacency_list_200_0.3_2.csv
graph_adjacency_list_200_0.3_50.csv
GraphGenerator.java
LongestAcyclicPathFinder.java
README.md
source sink pairs.txt
```

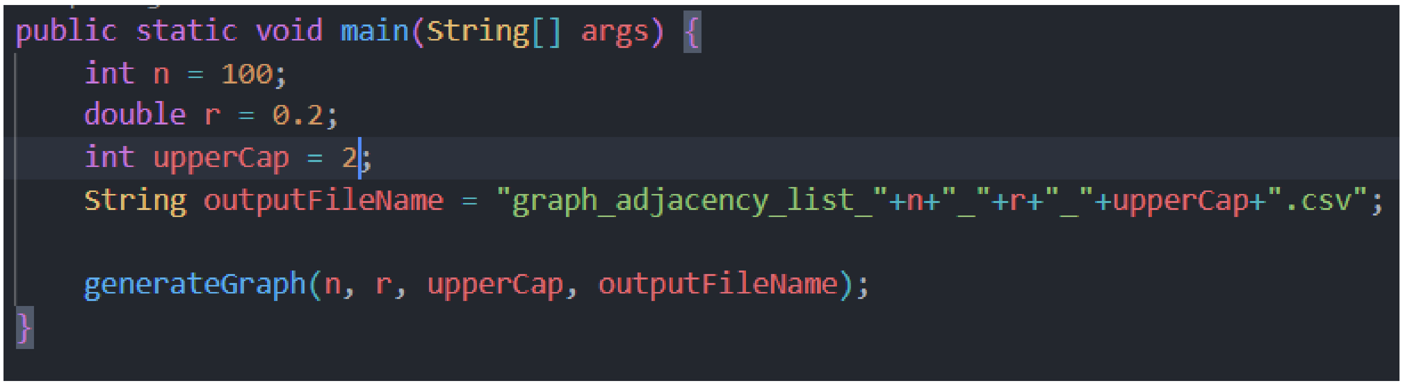
The nomenclature has been wisely to be self explanatory.

The files will be discussed in depth in the upcoming sections.

2.3) Files and their details

a) **GraphGenerator.java:**

This file is responsible for the generation of the Graph using the provided parameters and subsequently storing it in a csv file.



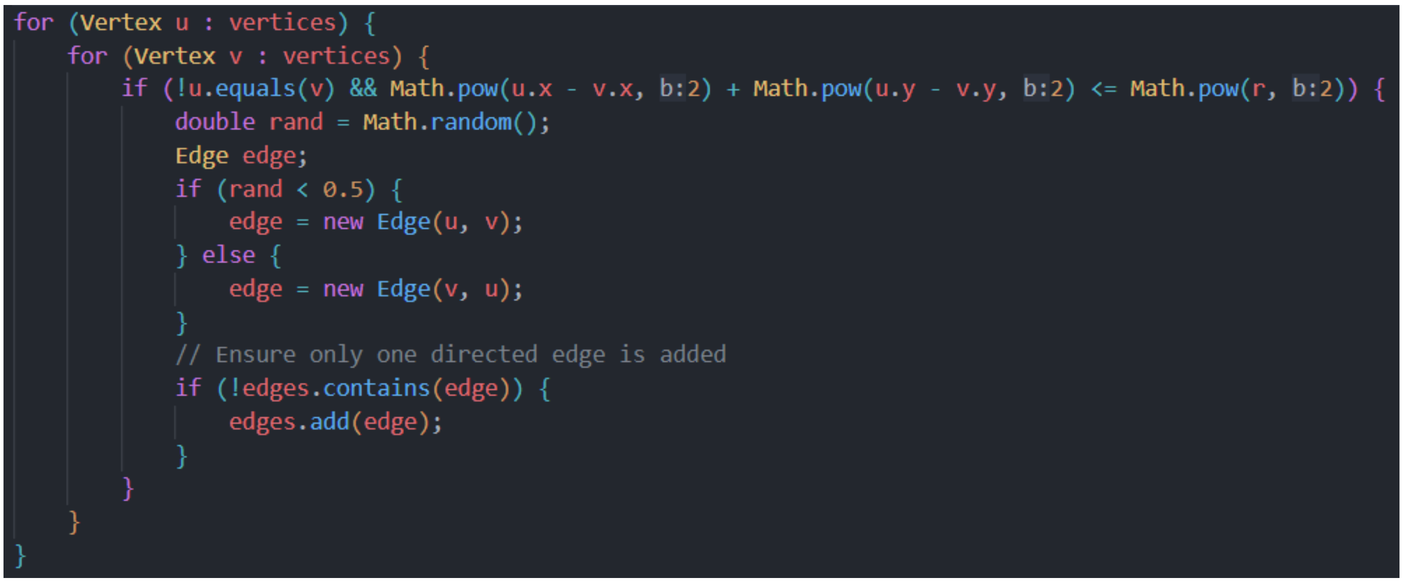
```java
public static void main(String[] args) {
    int n = 100;
    double r = 0.2;
    int upperCap = 2;
    String outputFileName = "graph_adjacency_list_"+n+"_"+r+"_"+upperCap+".csv";

    generateGraph(n, r, upperCap, outputFileName);
}
```

Input parameters are taken from the project description file, and stored in a csv file whose name is dependent on the graph properties.

Preprocessing: generate *n* vertices and provide them random x and y coordinates.

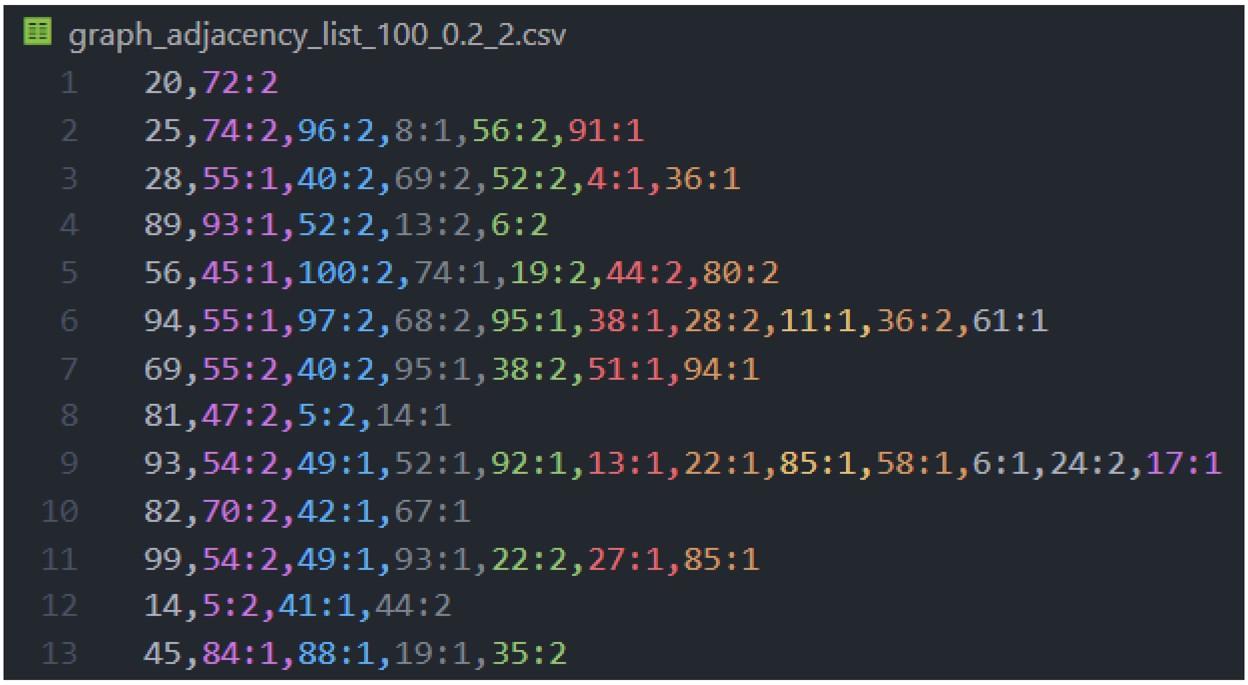
Now, for a pair of vertices, we will calculate the Euclidean distance between them and connect them with an edge using the condition mentioned in the project description.



```java
for (Vertex u : vertices) {
    for (Vertex v : vertices) {
        if (!u.equals(v) && Math.pow(u.x - v.x, b:2) + Math.pow(u.y - v.y, b:2) <= Math.pow(r, b:2)) {
            double rand = Math.random();
            Edge edge;
            if (rand < 0.5) {
                edge = new Edge(u, v);
            } else {
                edge = new Edge(v, u);
            }
            // Ensure only one directed edge is added
            if (!edges.contains(edge)) {
                edges.add(edge);
            }
        }
    }
}
```

Once the edges are connected, we can now assign capacities to each edge bounded by the upperCap.

We can now create an adjacency list using the above graph. Below is an example how we have stored the graph



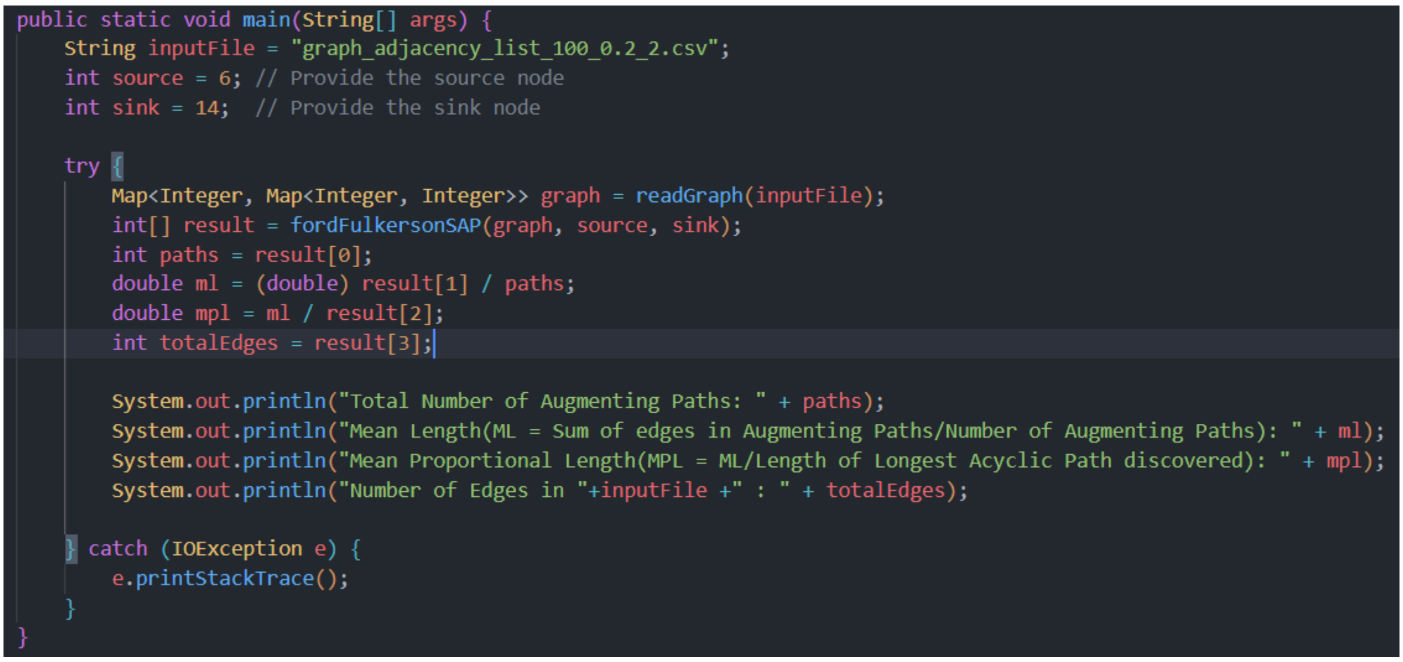
```
graph_adjacency_list_100_0.2_2.csv
1   20,72:2
2   25,74:2,96:2,8:1,56:2,91:1
3   28,55:1,40:2,69:2,52:2,4:1,36:1
4   89,93:1,52:2,13:2,6:2
5   56,45:1,100:2,74:1,19:2,44:2,80:2
6   94,55:1,97:2,68:2,95:1,38:1,28:2,11:1,36:2,61:1
7   69,55:2,40:2,95:1,38:2,51:1,94:1
8   81,47:2,5:2,14:1
9   93,54:2,49:1,52:1,92:1,13:1,22:1,85:1,58:1,6:1,24:2,17:1
10  82,70:2,42:1,67:1
11  99,54:2,49:1,93:1,22:2,27:1,85:1
12  14,5:2,41:1,44:2
13  45,84:1,88:1,19:1,35:2
```

Here, in line #1, the vertex 20 has an edge towards vertex 72. The capacity of this edge is 2.

b) **FordFulkersonSAP:**

This is the file which will implement the Ford-Fulkerson method while employing the SAP algorithm to find the augmenting paths.

Below are few important functions (skipping the helper functions) which define the implementation. These are similar in all the 3 implementations except the FordFulkersonMaxCap because that selects only the augmenting path with THE maximum(bottleneck) capacity and skips all other paths. This is because the description of this methods wants only one path and that should have THE highest capacity.



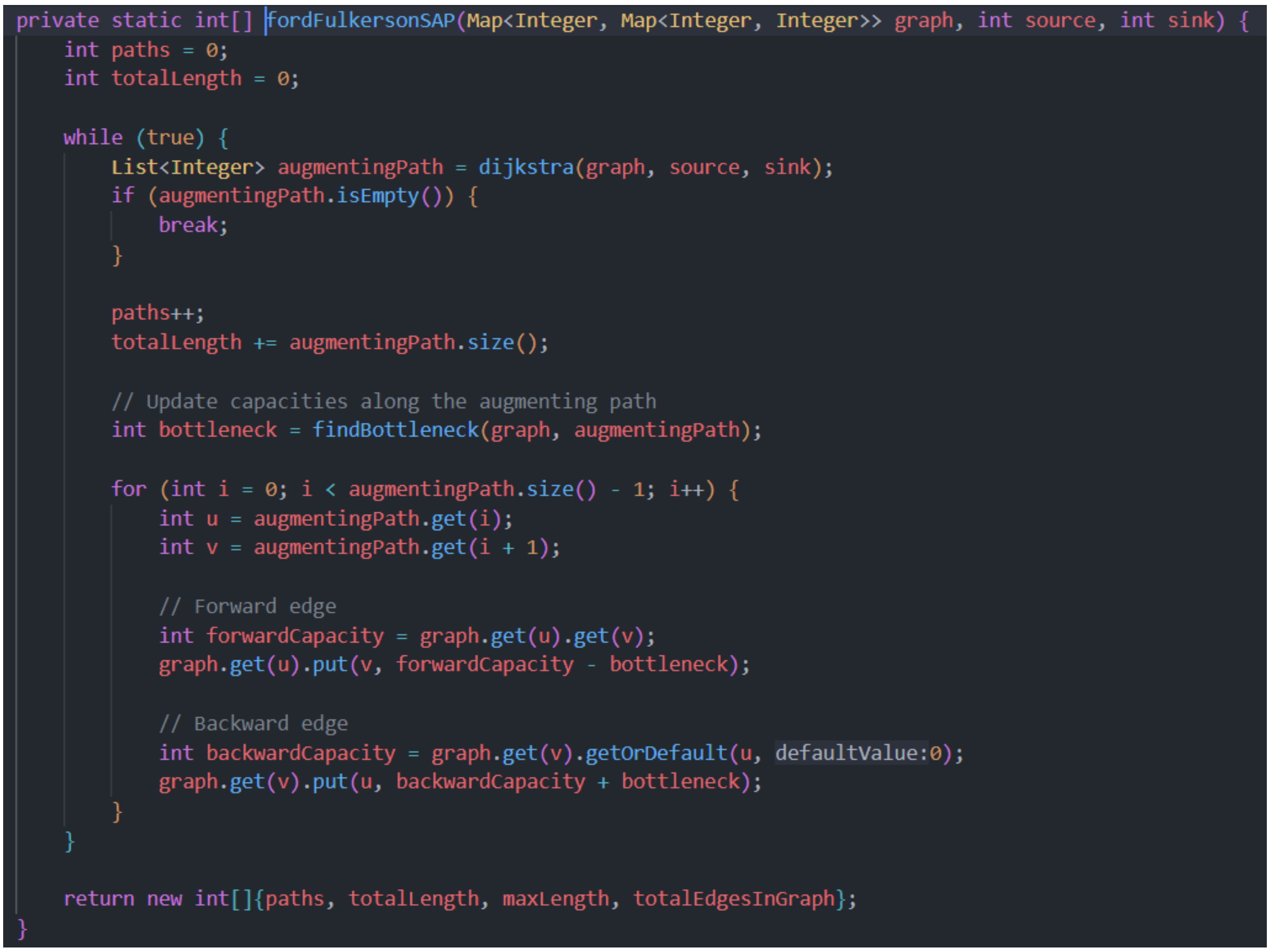
```java
public static void main(String[] args) {
    String inputFile = "graph_adjacency_list_100_0.2_2.csv";
    int source = 6; // Provide the source node
    int sink = 14;  // Provide the sink node

    try {
        Map<Integer, Map<Integer, Integer>> graph = readGraph(inputFile);
        int[] result = fordFulkersonSAP(graph, source, sink);
        int paths = result[0];
        double ml = (double) result[1] / paths;
        double mpl = ml / result[2];
        int totalEdges = result[3];

        System.out.println("Total Number of Augmenting Paths: " + paths);
        System.out.println("Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): " + ml);
        System.out.println("Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): " + mpl);
        System.out.println("Number of Edges in "+inputFile +" : " + totalEdges);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Above is the main function. All the raw metrices are calculated within the *fordfulkersonSap*.



```
private static int[] fordFulkersonSAP(Map<Integer, Map<Integer, Integer>> graph, int source, int sink) {
    int paths = 0;
    int totalLength = 0;

    while (true) {
        List<Integer> augmentingPath = dijkstra(graph, source, sink);
        if (augmentingPath.isEmpty()) {
            break;
        }

        paths++;
        totalLength += augmentingPath.size();

        // Update capacities along the augmenting path
        int bottleneck = findBottleneck(graph, augmentingPath);

        for (int i = 0; i < augmentingPath.size() - 1; i++) {
            int u = augmentingPath.get(i);
            int v = augmentingPath.get(i + 1);

            // Forward edge
            int forwardCapacity = graph.get(u).get(v);
            graph.get(u).put(v, forwardCapacity - bottleneck);

            // Backward edge
            int backwardCapacity = graph.get(v).getOrDefault(u, defaultValue:0);
            graph.get(v).put(u, backwardCapacity + bottleneck);
        }
    }

    return new int[]{paths, totalLength, maxLength, totalEdgesInGraph};
}
```

In the above function we will use a helper function d*ijkstra* to find the augmenting paths according to the project description. It will keep on generating the augmenting paths (inside an infinite loop). But when there is no augmenting path left, the loop will be broken.

c) **FordFulkersonDFS.java and FordFulkersonRandom.java**

Both of these programs follow the same structure as that of FordFulkersonSap. However, they implement their own modified version of d*ijkstra*, namely *dijkstraDFSLike* and *dijkstraRandom* respectively.

d) **FordFulkersonMaxCap.java**

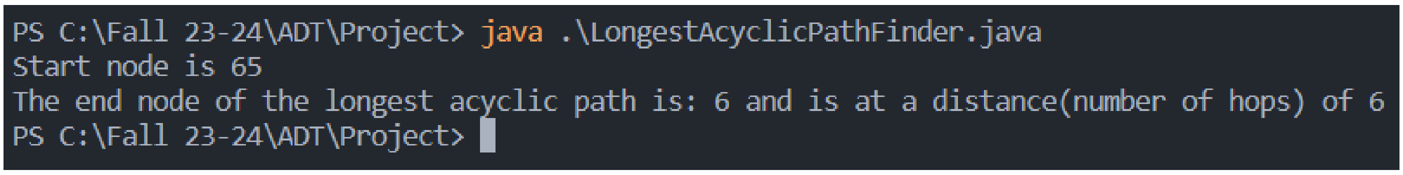
This program has a completely different approach. It will first search all the augmenting paths using the Dijkstra algorithm, and **select the augmenting path with the maximum bottleneck capacity.**

The metrics for this methods do stand out because of this selection pattern.

e) **LongestAcyclicPathFinder.java**:

This is a helper file which was used for finding the source sink pair for each graph. It takes the graph csv file as input. Then it'll randomly generate the source and find ***a*** longest acyclic path and mark the end as the sink.

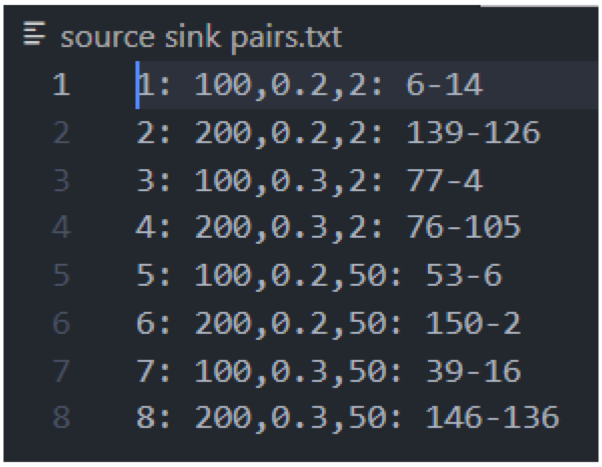
Below is an example of the output:



```
PS C:\Fall 23-24\ADT\Project> java .\LongestAcyclicPathFinder.java
Start node is 65
The end node of the longest acyclic path is: 6 and is at a distance(number of hops) of 6
PS C:\Fall 23-24\ADT\Project>
```

f) **source sink pairs.txt:**

This is another file created manually to record the source-sink pairs for each graph.



```
1: 100,0.2,2: 6-14
2: 200,0.2,2: 139-126
3: 100,0.3,2: 77-4
4: 200,0.3,2: 76-105
5: 100,0.2,50: 53-6
6: 200,0.2,50: 150-2
7: 100,0.3,50: 39-16
8: 200,0.3,50: 146-136
```

2.4) Execution procedure and Output format

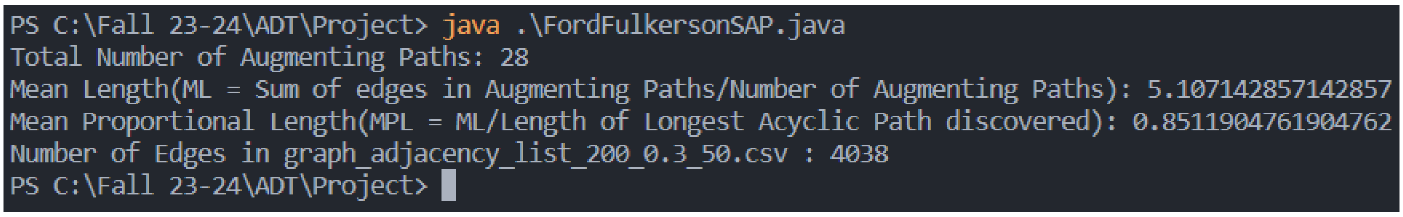
*** (Please compile all the files before running the programs) ***

Please follow the below sequence:

1) **GraphGenerator.java:** update the 3 parameters and run. Program will take care of the rest.

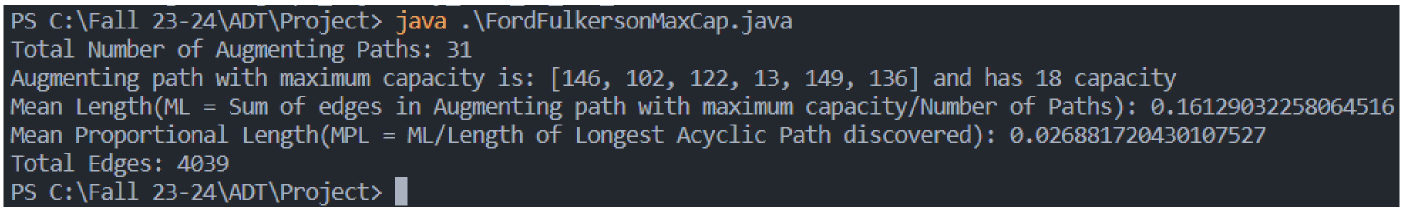
2) **LongestAcyclicPathFinder.java:** get the source-sink pair

3) Edit the **FordFurlkersonx** files for the graph file, source and sink. An output example is given below for **FordFulkersonSAP.java**.



```
PS C:\Fall 23-24\ADT\Project> java .\FordFulkersonSAP.java
Total Number of Augmenting Paths: 28
Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 5.107142857142857
Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.8511904761904762
Number of Edges in graph_adjacency_list_200_0.3_50.csv : 4038
PS C:\Fall 23-24\ADT\Project>
```

4) However, the output format of **FordFulkersonMaxCap.java** is completely different. Below is an example:



```
PS C:\Fall 23-24\ADT\Project> java .\FordFulkersonMaxCap.java
Total Number of Augmenting Paths: 31
Augmenting path with maximum capacity is: [146, 102, 122, 13, 149, 136] and has 18 capacity
Mean Length(ML = Sum of edges in Augmenting path with maximum capacity/Number of Paths): 0.16129032258064516
Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.026881720430107527
Total Edges: 4039
PS C:\Fall 23-24\ADT\Project>
```

We will additionally print the augmenting path with the maximum(bottleneck) capacity and choose the flow from this path only so that the sink will have flow equal to that of the bottleneck capacity of this path.

Therefore there will be multiple candidate paths, but we'll choose only the one which had the maximum capacity.

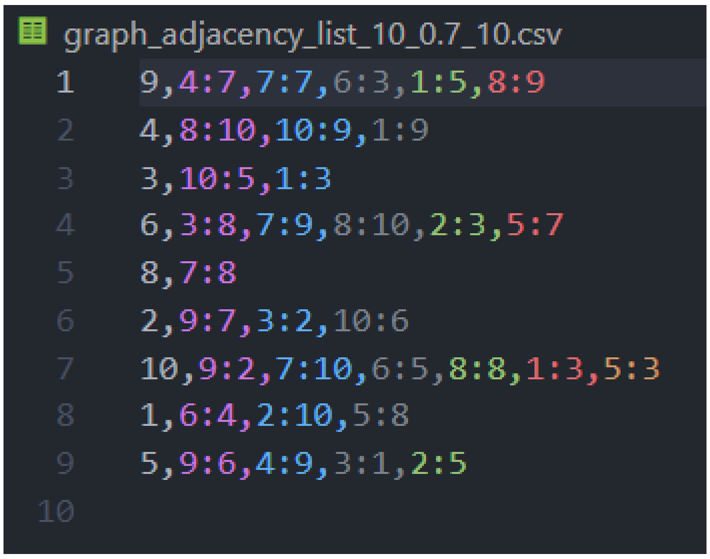
P.T.O

3. Implementation Correctness

To prove the correctness of my implementation, I will simulate the above process for a relatively small graph so as to verify the components manually.

I'll choose a graph with the following parameters:

Below is the csv file generated by GraphGenerator.java representing the graph for n=10, r=0.7 and upperCap=10:

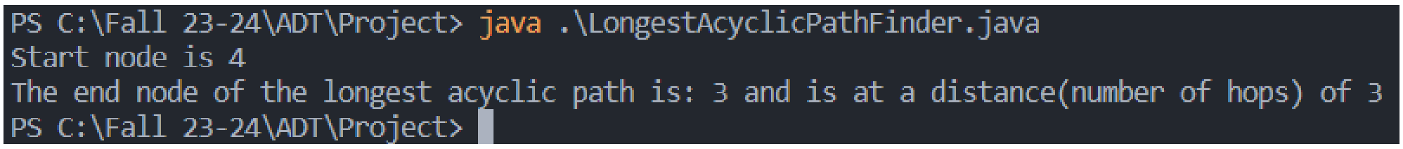


graph_adjacency_list_10_0.7_10.csv

```
9,4:7,7:7,6:3,1:5,8:9
4,8:10,10:9,1:9
3,10:5,1:3
6,3:8,7:9,8:10,2:3,5:7
8,7:8
2,9:7,3:2,10:6
10,9:2,7:10,6:5,8:8,1:3,5:3
1,6:4,2:10,5:8
5,9:6,4:9,3:1,2:5
```

Clearly, there is no cycle in this graph.

Now, using the LongestAcyclicPathFinder.java, I'll find the source-sink pair:



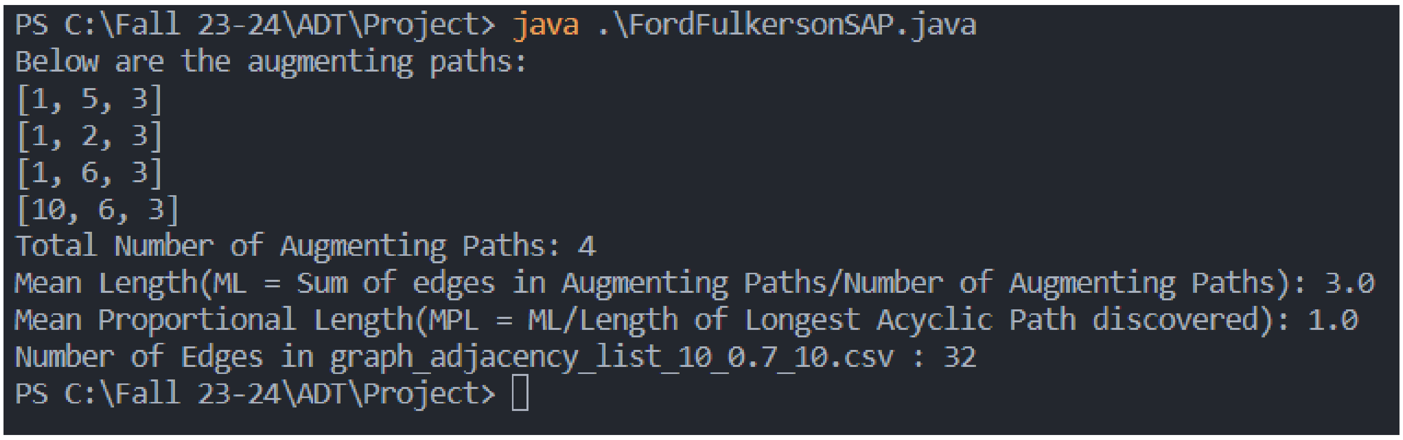
```
PS C:\Fall 23-24\ADT\Project> java .\LongestAcyclicPathFinder.java
Start node is 4
The end node of the longest acyclic path is: 3 and is at a distance(number of hops) of 3
PS C:\Fall 23-24\ADT\Project>
```

The pair is: source=4 and sink=3.

From the graph, we can see that, there is indeed a path: 4 > 10 > 6 > 3.

Now, I'll prove the correctness of ForFulkersonSAP.java. For that, I'll update the input file, source and sink. Also, print the augmenting paths to verify the correctness.

Below is the output:



```
PS C:\Fall 23-24\ADT\Project> java .\FordFulkersonSAP.java
Below are the augmenting paths:
[1, 5, 3]
[1, 2, 3]
[1, 6, 3]
[10, 6, 3]
Total Number of Augmenting Paths: 4
Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 3.0
Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 1.0
Number of Edges in graph_adjacency_list_10_0.7_10.csv : 32
PS C:\Fall 23-24\ADT\Project> 
```

The augmenting paths are shown without the source (showing only the next hops).

We can also show that there is not other augmenting path possible by performing an exhaustive search to the sink from the source. However this is just needed for the verification and is beyond the scope of this project's implementation.

Verifying the metrics:

1) **Total Number of Augmenting Paths: 4**

   These we printed above and can be easily cross verified

2) **Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 3.0**

   Total number of edges in augmenting paths = 3+3+3+3 = 12

   Number of augmenting paths = 4

   Therefore, mean length = 12/4 = 3

3) **Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 1.0**

   ML = 3

   Length of longest acyclic path = 3 (can be verified from the LongestAcyclicPathFinder.java output. It was anyways recalculated in FordFulkersonSAP.java)

   MPL = 3/3 = 1

4) **Number of Edges in graph_adjacency_list_10_0.7_10.csv : 32**

   This can be easily verified by counting the number of edges in the csv file.

Therefore, we were able to successfully verify the program's output and thus proving the correctness of the implementation.

4. Results

Simulation I

Below is the output in tabular format for the given 8 graphs:

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	100	0.2	2	1	9	1	542
	200	0.2	2	4	8.5	0.944	2165
	100	0.3	2	7	4.714	0.7857	1041
	200	0.3	2	20	5.25	0.875	4346
	100	0.2	50	5	8.6	0.86	537
	200	0.2	50	50	8.74	0.847	1924
	100	0.3	50	10	4.3	0.86	1092
	200	0.3	50	28	5.107	0.851	4039
DFS	100	0.2	2	1	14	1	542
	200	0.2	2	4	24.5	0.875	2165
	100	0.3	2	8	14.75	0.7023	1041
	200	0.3	2	21	20.095	0.744	4346
	100	0.2	50	5	14.4	0.8	537
	200	0.2	50	124	28.943	0.6158	1924
	100	0.3	50	33	14.363	0.683	1092

	200	0.3	50	89	17.719	0.681	4039
Algorithm	n	r	upperCap	paths	ML	MPL	total edges
MaxCap	100	0.2	2	1	9	1	542
	200	0.2	2	4	2	0.222	2165
	100	0.3	2	8	0.5	0.0833	1041
	200	0.3	2	21	0.238	0.039	4349
	100	0.2	50	5	1.8	0.18	537
	200	0.2	50	56	0.142	0.012	1924
	100	0.3	50	10	0.4	0.08	1092
	200	0.3	50	31	0.161	0.026	4039
Random	100	0.2	2	1	9	1	542
	200	0.2	2	4	9.5	0.836	2165
	100	0.3	2	8	5.625	0.703	1041
	200	0.3	2	21	6.190	0.619	4346
	100	0.2	50	3	9.333	0.848	537
	200	0.2	50	59	10.627	0.559	1924
	100	0.3	50	10	4.7	0.783	1092
	200	0.3	50	33	5.606	0.700	4039

Simulation II

To check which strategy performs well, I will generate two graphs of different complexities. One being very big (number of nodes) and the other being very dense (very large r).

This will effectively test the efficiency and strength of every method.

I actually want to observe the behavior of DFS, as it tends to find all the paths, its performance will take a huge hit when the graph gets very long and dense.

Also, the MaxCap has a limitation of choosing only one path that has *the* maximum bottleneck capacity, but can be very useful when we want to select only one path and THE path if we have only one option. This seems to be the most suited version for real life cases.

Lets run a few examples and observe the behavior of the 4 methods:

➔ For a very big network, I'll choose n = 1,000 nodes. The upperCap is set to 100 and r=0.2.

For SAP:

Total Number of Augmenting Paths: 75

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 6.493333333333333

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.9276190476190476

Number of Edges in graph_adjacency_list_1000_0.2_100.csv : 52466

For DFS:

Total Number of Augmenting Paths: 621

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 41.44927536231884

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.5921325051759835

Number of Edges in graph_adjacency_list_1000_0.2_100.csv : 52466

For MaxCap:

Total Number of Augmenting Paths: 69

Augmenting path with maximum capacity is: [486, 189, 843, 769, 423, 360, 237, 23] and has 44 capacity

Mean Length(ML = Sum of edges in Augmenting path with maximum capacity/Number of Paths): 0.10144927536231885

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.014492753623188406

Total Edges: 52466

For Random:

Total Number of Augmenting Paths: 70

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 7.071428571428571

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.4419642857142857

Number of Edges in graph_adjacency_list_1000_0.2_100.csv : 52466

- ➔ Now, I'll use a very dense network: with n=500, r=0.5 and upperCap=100:

For SAP:

Total Number of Augmenting Paths: 173

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 3.5433526011560694

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.8858381502890174

Number of Edges in graph_adjacency_list_500_0.5_100.csv : 58336

For DFS:

Total Number of Augmenting Paths: 817

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 13.151774785801713

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.5718162950348571

Number of Edges in graph_adjacency_list_500_0.5_100.csv : 58336

For MaxCap:

Total Number of Augmenting Paths: 162

Augmenting path with maximum capacity is: [492, 166, 295, 23] and has 92 capacity

Mean Length(ML = Sum of edges in Augmenting path with maximum capacity/Number of Paths): 0.018518518518518517

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.004629629629629629

Total Edges: 58336

For Random:

Total Number of Augmenting Paths: 142

Mean Length(ML = Sum of edges in Augmenting Paths/Number of Augmenting Paths): 3.535211267605634

Mean Proportional Length(MPL = ML/Length of Longest Acyclic Path discovered): 0.5892018779342724

Number of Edges in graph_adjacency_list_500_0.5_100.csv : 58336

Tabulating the above outputs generated using the computation.encs.concordia.ca server:

Algorithm	n	r	upperCap	paths	ML	MPL	total edges
SAP	1000	0.2	100	75	6.493	0.927	52466
	500	0.3	100	173	3.543	0.0.88	58336
DFS	1000	0.2	100	621	41.449	0.592	52466
	500	0.3	100	817	13.151	0.571	58336
MaxCap	1000	0.2	100	69	0.014	0.0144	52466
	500	0.3	100	162	0.018	0.004	58336
Random	1000	0.2	100	70	7.071	0.441	52466
	500	0.3	100	142	3.532	0.589	58336

5. Conclusion

From the above combination of more nodes and dense nodes, we can conclude that the DFS method was able to explore the maximum number of augmenting paths and thus providing the maximum flow.

Also, SAP is capable of finding the set of paths that result in the least Mean Length because it chooses the shortest paths.

These different heuristics can be used to solve different problem cases.

Lets say we want to create a single pipeline between two cities and want the maximum flow in that pipeline. We will go with the MaxCap as it is capable of finding the path with the maximum bottleneck capacity.

Or, lets say we want to find to connect two cities with by multiple highways, while minimising the material used per highway. In that case, we will choose the SAP method because it gives us the least Mean Length.

From the above, we can conclude that there is no *best* algorithm and each one of them come up with their own set of advantages and drawbacks.

It boils down to the user and the constraints which will define the choice of the method to be used.

The zip file is also uploaded with a well described Readme file.

The entire project's GitHub repo can be reached here at a later date GitHub - sushant-sinha/ADT-Project

P.T.O

6. References

1) CLRS, 4th edition

2) FileWriter (Java Platform SE 8) (oracle.com)

3) Java Method Overriding (programiz.com)

4) Java BufferedReader Class - javatpoint

5) Implement PriorityQueue through Comparator in Java - GeeksforGeeks

6) How to connect to a GCS Public Lab computer from outside of the Concordia network - Concordia University

7) How do I securely connect to a GCS server? - Concordia University