CONCORDIA UNIVERSITY
COMP 6651: ALGORITHM DESIGN AND ANALYSIS

REPORT BY: SIMRAN KAUR
STUDENT ID: 40221666

PROBLEM DESCRIPTION

The Ford Fulkerson algorithm is a widely used method to solve maximum flow problems.The algorithm is based on the concept of augmenting paths, which are the paths in the network along which additional flow can be sent.The problem focuses on finding the maximum flow in a source sink network using the Ford Fulkerson algorithm. The algorithm iterates through the augmenting paths and adds the flow to the maximum flow based on the capacity of the path.In this project we aim to use four different augmenting path algorithms and study their performance on randomly generated source-sink graph

IMPLEMENTATION DETAILS

To create a graph, 'n' number of vertices are created with random x and y coordinates. These vertices are then picked at random and are connected with one another to work a directed graph. One node out of the n vertices is selected as the source vertex. From this source vertex, at the end of the longest acyclic path is the sink vertex . The sink vertex is found using the BFS algorithm.

PSEUDO CODE:

```
function main_code(n, r, upperCap, fname):
    vertices = generateVertices(n)
    edges = generateEdges(vertices, r, upperCap)
    graph = generateGraph(vertices, edges)
    findSourceSink(vertices, edges, graph, size(edges), fname)

function generateVertices(n):
    vertices = []
    rand = Random()
    for i in range(n):
        x = rand.nextDouble()
        y = rand.nextDouble()
        vertices.add(Vertex(i, x, y))
    return vertices

function generateEdges(vertices, r, upperCap):
    edges = []
    rand = Random()
```

```
    for i in range(size(vertices)):
        for j in range(size(vertices)):
            u = vertices[i]
            v = vertices[j]
            if u != v and distance(u, v) <= r:
                val = rand.nextDouble()
                if val < 0.5 and not hasEdge(edges, u, v):
                    edges.add(Edge(u, v, rand.nextInt(upperCap - 1) + 1))
                else if not hasEdge(edges, v, u):
                    edges.add(Edge(v, u, rand.nextInt(upperCap - 1) + 1))
    return edges

function hasEdge(edges, u, v):
    for edge in edges:
        if (edge.u == u and edge.v == v) or (edge.u == v and edge.v == u):
            return true
    return false

function generateGraph(vertices, edges):
    graph = []
    for i in range(size(vertices)):
        v = [vertices[i]]
        e = []
        for j in range(size(edges)):
            if edges[j].u == vertices[i]:
                e.add(edges[j])
        v.add(e)
        graph.add(v)
    return graph
```

The graph created is stored in an ASCII file which is used in the augmenting path algorithms. The four augmenting paths implemented are variations of the Dijkstra algorithm.

```
 function Dijkstra(graph, start):
    // Initialization
    distance = initializeDistance(graph, start)
    visited = set()
    priorityQueue = createPriorityQueue(graph, start)
```

```
// Main loop
while priorityQueue is not empty:
    currentVertex = dequeueMin(priorityQueue)
    visited.add(currentVertex)

    // Relaxation step
    for neighbor in neighbors(currentVertex):
        if neighbor is not in visited:
            newDistance = distance[currentVertex] + edgeWeight(currentVertex, neighbor)
            if newDistance < distance[neighbor]:
                distance[neighbor] = newDistance
                updatePriorityQueue(priorityQueue, neighbor, newDistance)

return distance
```

1. Shortest Augmenting path: Just like the dijkstra algorithm if the distance of the current node is greater than the parent node plus the weight, the edge is relaxed and the updated node is put in the priority queue. Once the sink was found , control would break and would call the add_flow function where the flow of the graph is updated and then the update_graph generates the residual graph to further explore the graph.
1. DFS like: To implement this in case the v.d = INFINITY then it was set to a decrementing counter value so that it could be explored at a later stage and would appear as a DFS algorithm.
2. Maximum capacity was the modification of dijkstra which was to find out the maximum capacity instead of the minimum and explore the nodes with maximum capacity first.
3. Random: In this the edges to be explored were selected at random.

ALGORITHM CORRECTNESS

Correctness of an algorithm is a critical aspect of an algorithm, ensuring that it produces accurate and valid results across different sets of inputs.The correctness of the algorithm is typically assessed through systematic testing and verification.

- Manual Testing:For a small set of nodes, the graph was manually tested, which provided a basic sense of validation. This manual testing involved creating specific scenarios with known inputs, running the algorithm and comparing the results against the expected outcome.

- Random Testing: To expand the testing coverage and further test the reliability of the algorithm, random testing was used. Random Testing involves continuously testing the algorithm with various set of nodes , capacities. The behavior of the algorithm was

observed across a broader set of nodes and capacities, helping to identify potential inconsistencies.

RESULTS:

N=100, R=0.2,CAPACITY=2

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	29	2	5.14E-4	4838	29.0
DFS like	29	2	5.14E-4	4838	29.0
MaxCap	29	2	5.14E-4	4838	29.0
Random	29	2	5.14E-4	4838	29.0

N=100, R=0.2,CAPACITY=50

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	73	2	0.001035	4811.0	73.0
DFS like	376.0	4	0.002069	4811.0	1685.0
MaxCap	120.0	2	0.001035	4811.0	1685.0
Random	145.0	2	0.001035	4811.0	1685.0

N=100, R=0.3,CAPACITY=2

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	37.0	2	3.65E-4	4729.0	37.0
DFS like	37.0	2	3.65E-4	4729.0	37.0
MaxCap	37.0	2	3.65E-4	4729.0	37.0
Random	37.0	2	3.65E-4	4729.0	37.0

N=100, R=0.3,CAPACITY=50

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	30.0	2	3.37E-4	4698.0	30.0
DFS like	270.0	4	6.74E-4	4698.0	904.0
MaxCap	53.0	2	3.37E-4	4698.0	904.0
Random	83.0	2	3.37E-4	4698.0	904.0

N=100, R=0.15,CAPACITY=2

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	85.0	2	0.002049	4864.0	85.0
DFS like	85.0	2	0.002049	4864.0	85.0

MaxCap	85.0	2	0.002049	4864.0	85.0
Random	85.0	2	0.002049	4864.0	85.0

N=200, R=0.2,CAPACITY=2

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	117.0	2	1.33E-4	19391.0	117.0
DFS like	117.0	2	1.33E-4	19391.0	117.0
MaxCap	117.0	2	1.33E-4	19391.0	117.0
Random	117.0	2	1.33E-4	19391.0	117.0

N=200, R=0.2,CAPACITY=50

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	114.0	2	1.39E-4	19383.0	114.0
DFS like	1030.0	5	3.47E-4	19383.0	3072.0
MaxCap	213.0	3	2.08E-4	19383.0	3072.0
Random	297.0	2	1.39E-4	19383.0	3072.0

N=200, R=0.3,CAPACITY=2

	Number of Augmenting Paths	Average Length of augmenting	Mean proportional length	Total Edges	Max flow

		path			
Shortest Augmenting Path	109.0	2	8.0E-5	18836.0	109.0
DFS like	109.0	2	8.0E-5	18836.0	109.0
MaxCap	109.0	2	8.0E-5	18836.0	109.0
Random	109.0	2	8.0E-5	18836.0	109.0

N=200, R=0.3,CAPACITY=50

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	146.0	2	8.4E-5	18840.0	146.0
DFS like	1287.0	4	1.69E-4	18840.0	3463.0
MaxCap	281.0	4	1.69E-4	18840.0	3463.0
Random	404.0	2	8.4E-5	18840.0	3463.0

To evaluate the performance of DFS like algorithm(which seems to be inefficient) and MaxCap (which seems to be an efficient algorithm) the two sets of input introduced are:
N: 10 R: 0.2 CAPACITY: 10: To see if DFS-like performs equivalent to MaxCap or Random for a smaller capacity.
N=100, R=0.15,CAPACITY=10

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	179.0	2	5.71E-4	19609.0	179.0

DFS like	635.0	3	8.57E-4	19609.0	853.0
MaxCap	322.0	3	8.57E-4	19609.0	853.0
Random	393.0	2	5.71E-4	19609.0	853.0

N=200, R=0.15,CAPACITY=50

	Number of Augmenting Paths	Average Length of augmenting path	Mean proportional length	Total Edges	Max flow
Shortest Augmenting Path	130	2	2.11E-4	19569.0	130.0
DFS like	1054	4	4.22E-4	19569.0	3286.0
MaxCap	251	3	3.16E-4	19569.0	3286.0
Random	329	2	2.11E-4	19569.0	3286.0

CONCLUSION

- On keeping the value of cap=2, (N=100, R=0.2,CAPACITY=2) all the algorithms seem to produce the same result, in terms of number of augmenting paths, average length of augmenting path, mean proportional length, total edges and max flow.
- On increasing the capacities, the DFS like, MaxCap and Random shows better max flow results as compared to Shortest Augmenting path.
- In smaller numbers of augmenting paths and smaller average length,Max Cap is able to provide better Max flow than DFS like a Random algorithm.
- DFS-like algorithms have been seen to provide the same Maximum Flow for N=200, R=0.2,CAPACITY=50 in comparison to MaxCap and Random. However, the augmenting paths used to produce the result is quite high. Thereby, making the algorithm inefficient.
- The DFS-like algorithm in N=100, R=0.15,CAPACITY=10 is as usual seen to have higher number of augmenting paths, however when the capacity is increased for the same density parameter, it is seen that the number of paths DFS explores is almost five times as explored by MAXCAP.

The algorithm that would work the best for more nodes or for more density is MaxCap. It has been observed that after increasing the value of ‘n’ or ‘r’. The algorithm is able to find the maximum flow in least number of augmenting paths.Thereby making it an efficient algorithm.

REFERENCES

1.https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

2.https://www.programiz.com/dsa/ford-fulkerson-algorithm

3. https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

4.https://cs.stackexchange.com/questions/55041/residual-graph-in-maximum-flow

5.chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.cmu.edu/afs/cs/academic/class/15750-s18/ScribeNotes/lecture18.pdf