Concordia University

Engineering and Computer Science

COMP 6651

Algorithm Design Techniques

Project Report

Submitted To: Dr. Thomas Fevens

Student Name : Bhanu Prakash Vattikuti

Table of Contents:

Problem description:

The primary goal is to implement and thoroughly assess four distinct augmenting path algorithms: Shortest Augmenting Path (SAP), DFS-like, Maximum Capacity (MaxCap), and Random, all associated with the Ford-Fulkerson Method. Drawing inspiration from Dijkstra's approach, these algorithms introduce variations to identify the most efficient augmenting paths in residual networks. The key challenge involves a comprehensive evaluation of their performance across diverse random source-sink graphs, generated using coordinates, distances, and edge capacities. By closely examining metrics such as path counts, average path lengths, and total edges, the project aims to illuminate the comparative strengths and weaknesses of these algorithms. This effort ultimately contributes to refining and optimizing maximum flow computations.

Implementation details:

Graph Generation:

- The process of generating a graph involves creating a source-sink network for simulations. It begins with generating 'n' nodes.
- Each node is assigned a random (x, y) coordinate between 0 and 1, creating a spatial distribution.
- Directed edges connect ordered pairs of nodes to form the network, ensuring there are no parallel edges.
- A breadth-first search identifies the longest acyclic path from the source to the sink, designating them as 'source (s)' and 'sink (t)' nodes and establishing flow directionality within the network.
- Integer-value capacities from the range [1..upperCap] are randomly assigned to each edge. The entire graph and its edge capacities are saved in an ASCII-readable format like an adjacency list, CSV.

Graph Initialization:

- To begin, select a random node from the set of vertices 'V' to be the source node 's'.
- Conduct a breadth-first search (BFS) operation starting from the source node 's' to find the longest acyclic path in the graph.
- This path concludes at a specific node, designated as the sink node 't' in the source-sink network.
- This process ensures that source and sink nodes are identified in a way that promotes flow directionality in the graph.

Augmenting Path Algorithms:

1. Shortest Augmenting Path (SAP): Initialization:
    - Initialize distance values for each node to infinity, except for the source node, set to 0.
    - Create a 'previous' dictionary to store parent nodes for each node.
    - Initialize a priority queue (min_heap) with tuples containing distance and source node.

SAP Traversal:

- While the priority queue is not empty, repeatedly pop the node with the minimum distance.

- If the popped node is the sink node, reconstruct the augmenting path using the 'previous' dictionary and return it.

Path Exploration:

- Skip visited nodes.
- For each neighbor with available edge capacity, calculate the total distance to reach the neighbor through the current path.
- Update the distance and set the current node as the parent of the neighbor if the calculated distance is shorter.

Result:

- If the sink node is reached, return the augmenting path as a list of nodes. Otherwise, return None.

2. DFS-like: Initialization:
    - Initialize distance values for each node to infinity, except for the source node, set to 0.
    - Create a priority queue (priority_queue) with tuples containing distance and the source node.
    - Initialize a counter to 0 for decreasing key values.

DFS-like Traversal:

- While the priority queue is not empty, repeatedly pop the node with the minimum distance.
- Update distances and explore neighbors to find shorter paths.

DFS-like Behavior:

- Assign decreasing key values to unvisited nodes, resembling a depth-first search aspect.

Result:

- Return a dictionary (distances) with final distance values from the source node to all other nodes.

3. Maximum Capacity (MaxCap): Initialization:
    - Initialize distance values for each node to negative infinity, except for the source node, set to positive infinity.
    - Create a 'previous' dictionary to store parent nodes.
    - Initialize a priority queue (min_heap) with tuples containing negative infinity and the source node.

MaxCap Traversal:

- While the priority queue is not empty, repeatedly pop the node with the maximum capacity (negative value).
- If the popped node is the sink node, reconstruct the augmenting path using the 'previous' dictionary and return it along with the critical edge capacity.

Path Exploration:

- Skip visited nodes.
- Calculate the minimum capacity along the path to reach neighbors and update distances.

Result:

- If the sink node is reached, return the augmenting path and critical edge capacity. Otherwise, return None.

4. Random: Initialization:
    - Initialize distance values for each node to infinity, except for the source node, set to 0.
    - Create a 'previous' dictionary to store parent nodes.

- Initialize a priority queue (min_heap) with tuples containing distance, a random value, and the source node.

Random Traversal:

- While the priority queue is not empty, repeatedly pop the node with the minimum distance (and random value).
- If the popped node is the sink node, reconstruct the augmenting path using the 'previous' dictionary and return it.

Path Exploration:

- Skip visited nodes.
- Calculate the total distance to reach neighbors through the current path and update distances.

Result:

- If the sink node is reached, return the augmenting path as a list of nodes. Otherwise, return None.

Simulation and Analysis:

- Simulations with given network configurations (n, r, upperCap) are executed.
- Each augmenting path algorithm runs the Ford-Fulkerson algorithm and records metrics such as the number of augmenting paths, mean length, mean proportional length, and total edges.
- Results are presented in a text format for easy analysis.

Implementation correctness:

We can analyse correctness of this code I am submitting in 2 sub blocks.

1. Correctness of Graph generation:
   I have checked correctness of the generated graph using visualization of graph and saving it to CSV and retrieving it from CSV with randomly generate source-sink. They are in my submitted code.
2. Correctness of Individual Algorithms:
   For testing the implementation correctness of the Algorithm I have tested using following parameters.

a. Tested for Simple Graph: Created a simple graph manually and test the shortest path calculation.

b. Tested with No Path: Created a graph where no path exists between the source and the sink.

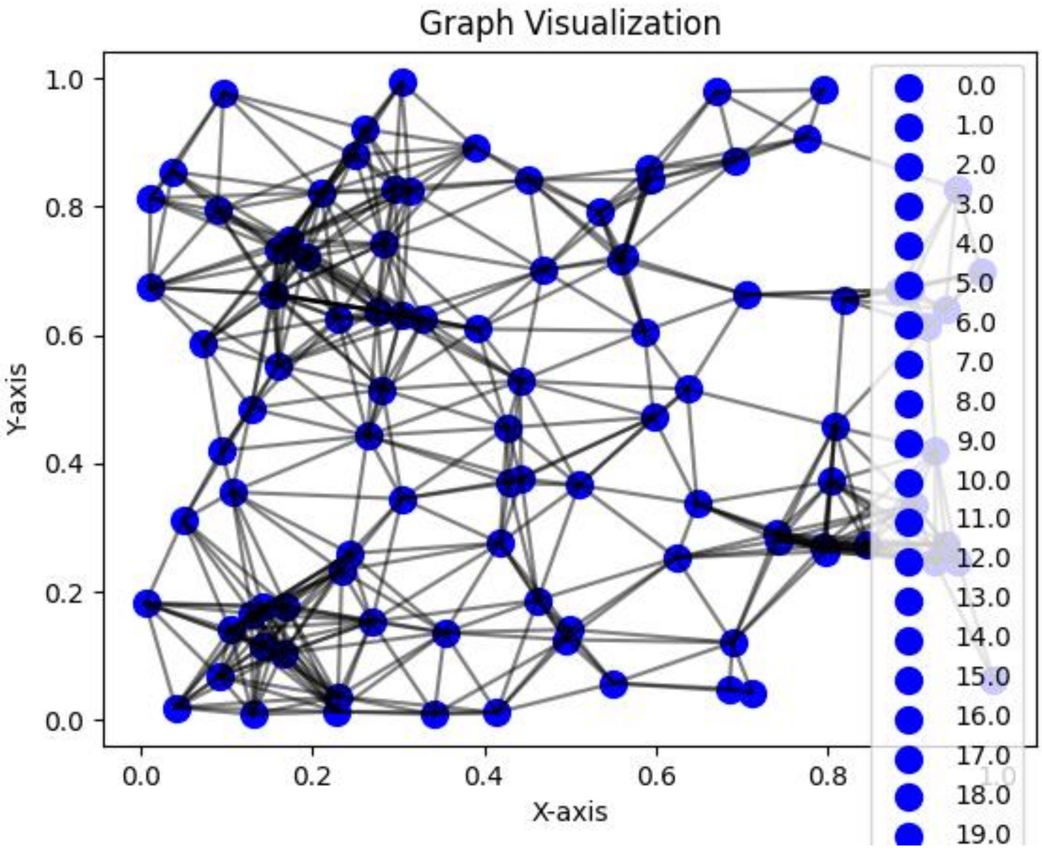
c. Tested with Randomly Generated Graph: Utilize graph generation logic to create a more complex graph than manual and test the algorithm.

```
class TestRandonDij(unittest.TestCase):

    def setUp(self):
        self.simple_graph = Graph()
        self.simple_graph.add_node(0, 0, 0)
        self.simple_graph.add_node(1, 1, 1)
        self.simple_graph.add_node(2, 2, 2)
        self.simple_graph.add_edge(0, 1, 1)
        self.simple_graph.add_edge(1, 2, 2)
        self.simple_graph.set_source(0)
        self.simple_graph.set_sink(2)

    def test_simple_path(self):
        augmenting_algorithms = [SAP, dfs_like, MaxCap, Randon_Dij]
        for algorithm in augmenting_algorithms:
            path = augmenting_algorithms(self.simple_graph, self.simple_graph.get_source(),
self.simple_graph.get_sink())
            self.assertIn(path, [[0, 1, 2]])

    def test_no_path(self):
        isolated_graph = Graph()
        isolated_graph.add_node(0, 0, 0)
        isolated_graph.add_node(1, 1, 1)
        isolated_graph.set_source(0)
        isolated_graph.set_sink(1)
        augmenting_algorithms = [SAP, dfs_like, MaxCap, Randon_Dij]
        for algorithm in augmenting_algorithms:
            path = augmenting_algorithms(isolated_graph, isolated_graph.get_source(),
isolated_graph.get_sink())
            self.assertIsNone(path)

    def test_random_generated_graph(self):
        random_graph = generate_graph(10, 0.5, 5)
        augmenting_algorithms = [SAP, dfs_like, MaxCap, Randon_Dij]
        for algorithm in augmenting_algorithms:
            path = augmenting_algorithms(random_graph, random_graph.get_source(),
random_graph.get_sink())
            self.assertIsNotNone(path)
```

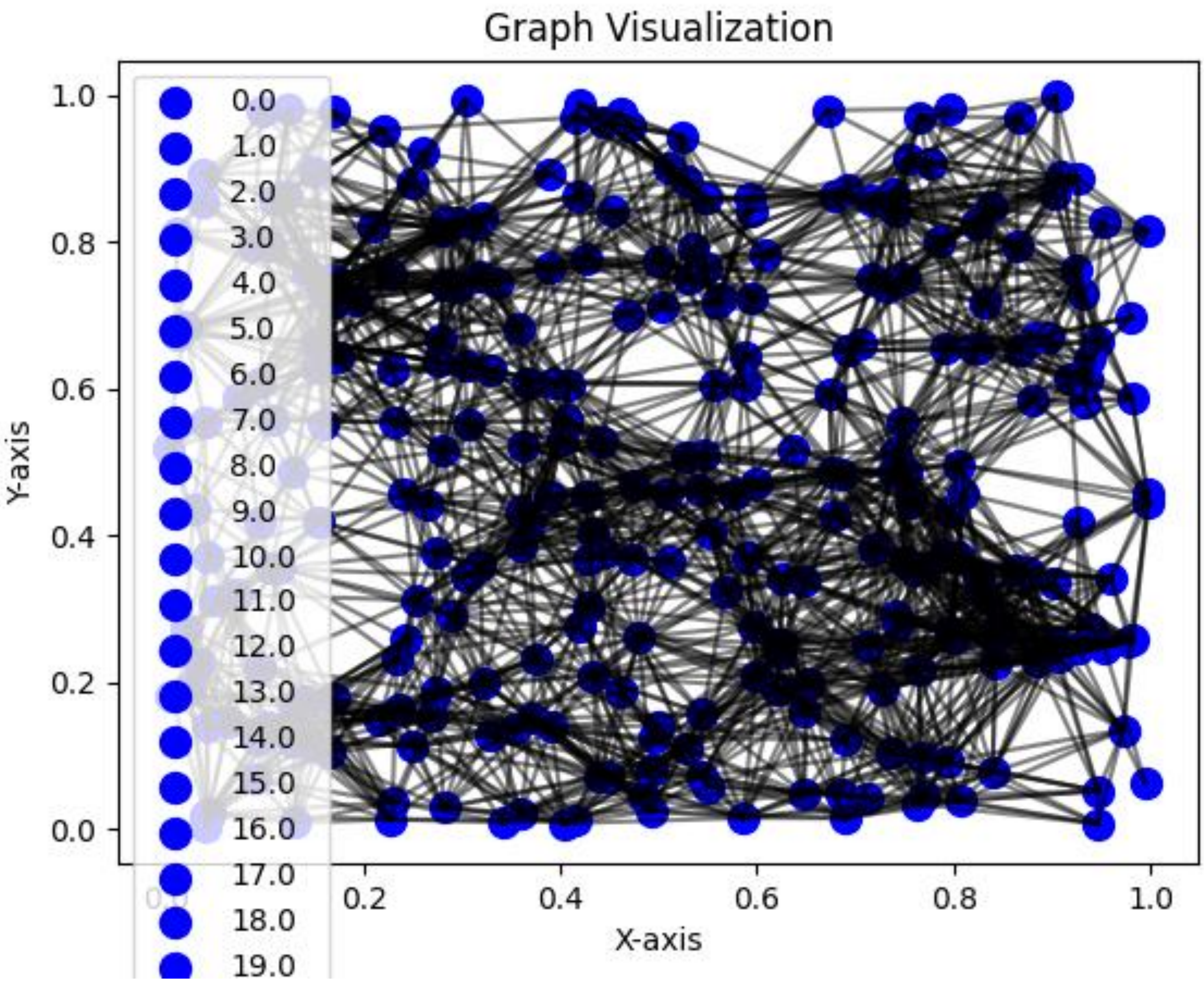
Results:

Simulations I:

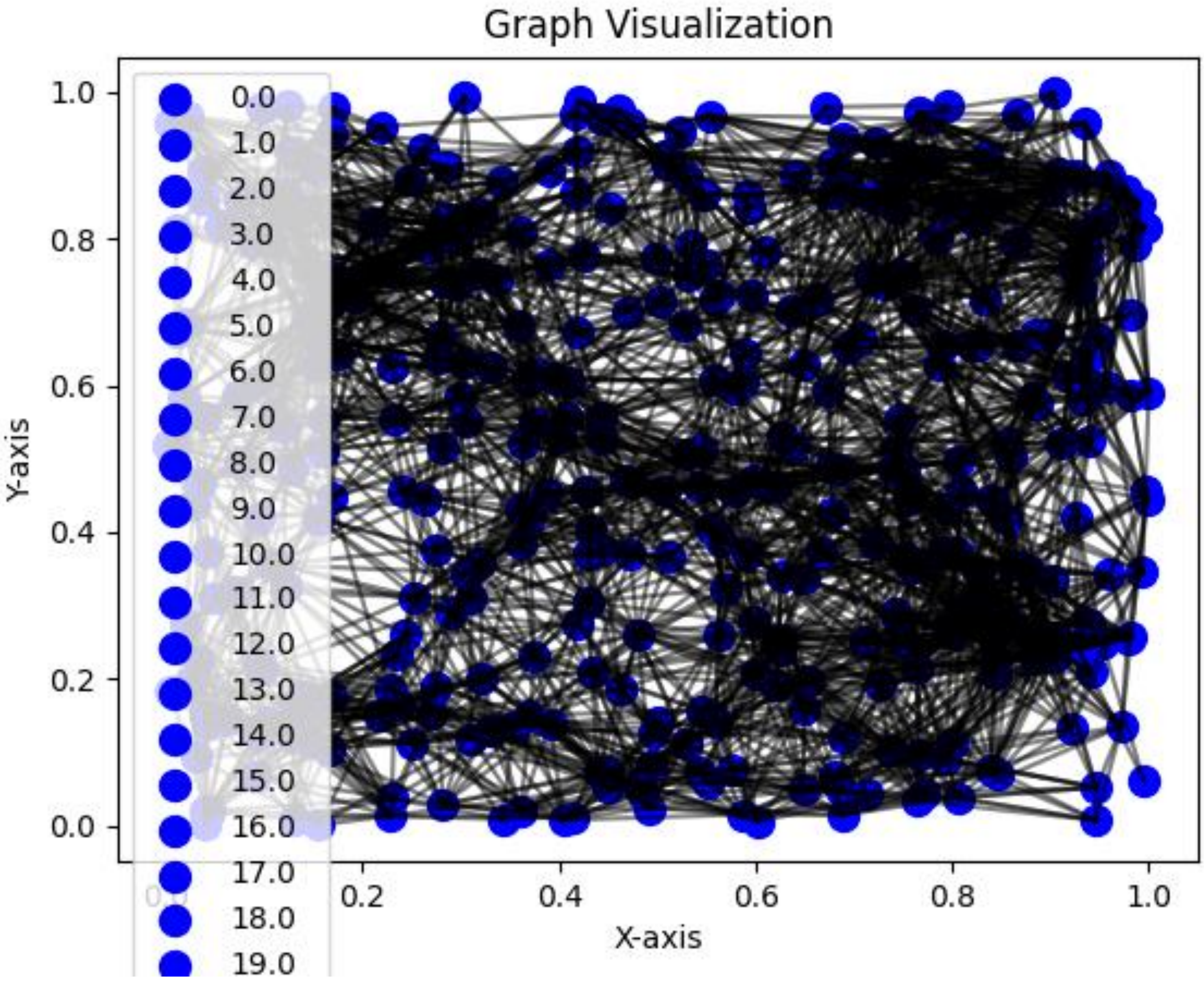
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	100	0.2	2	4	9.5	1	533
DFS	100	0.2	2	4	36.25	1	533
MAXCAP	100	0.2	2	4	17.75	1	533
RANDOM	100	0.2	2	4	9.5	1	533



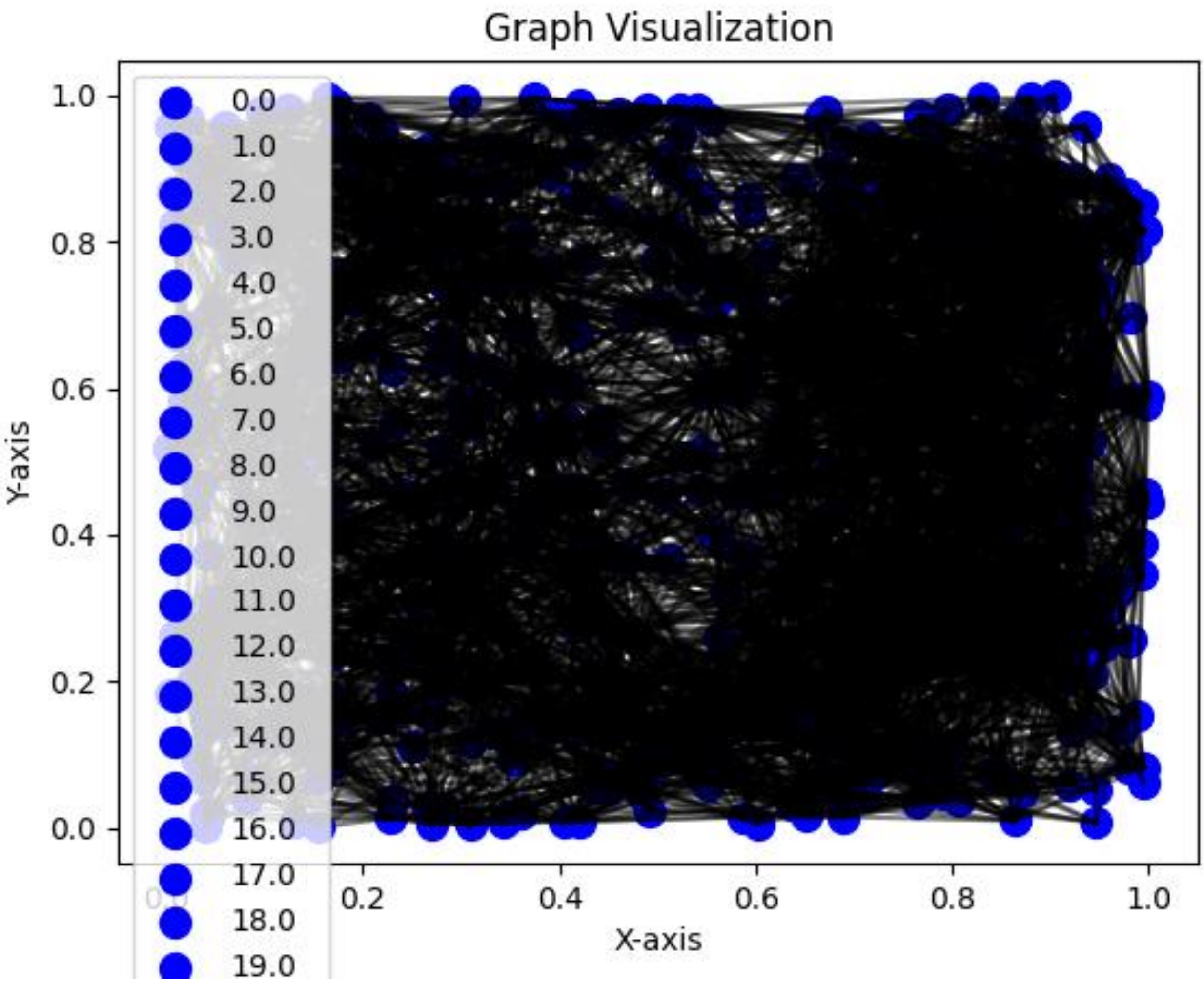
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	200	0.2	2	11	8.45	1	2039
DFS	200	0.2	2	11	90.81	1	2039
MAXCAP	200	0.2	2	11	13.45	1	2039
RANDOM	200	0.2	2	11	9.18	1.02	2039



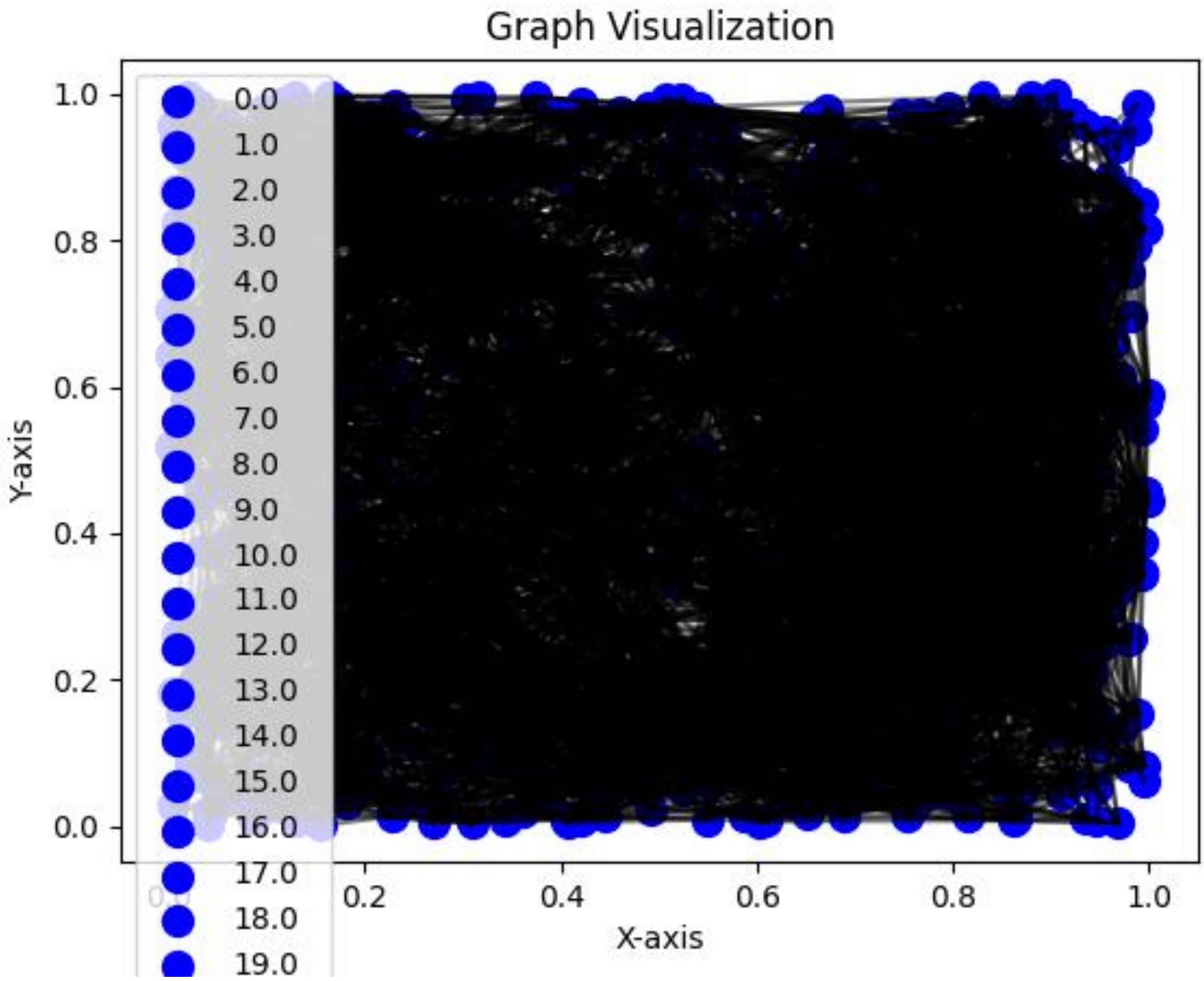
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	100	0.3	2	4	7.5	1	988
DFS	100	0.3	2	4	56	1	988
MAXCAP	100	0.3	2	4	13.5	1	988
RANDOM	100	0.3	2	4	8.5	1	988



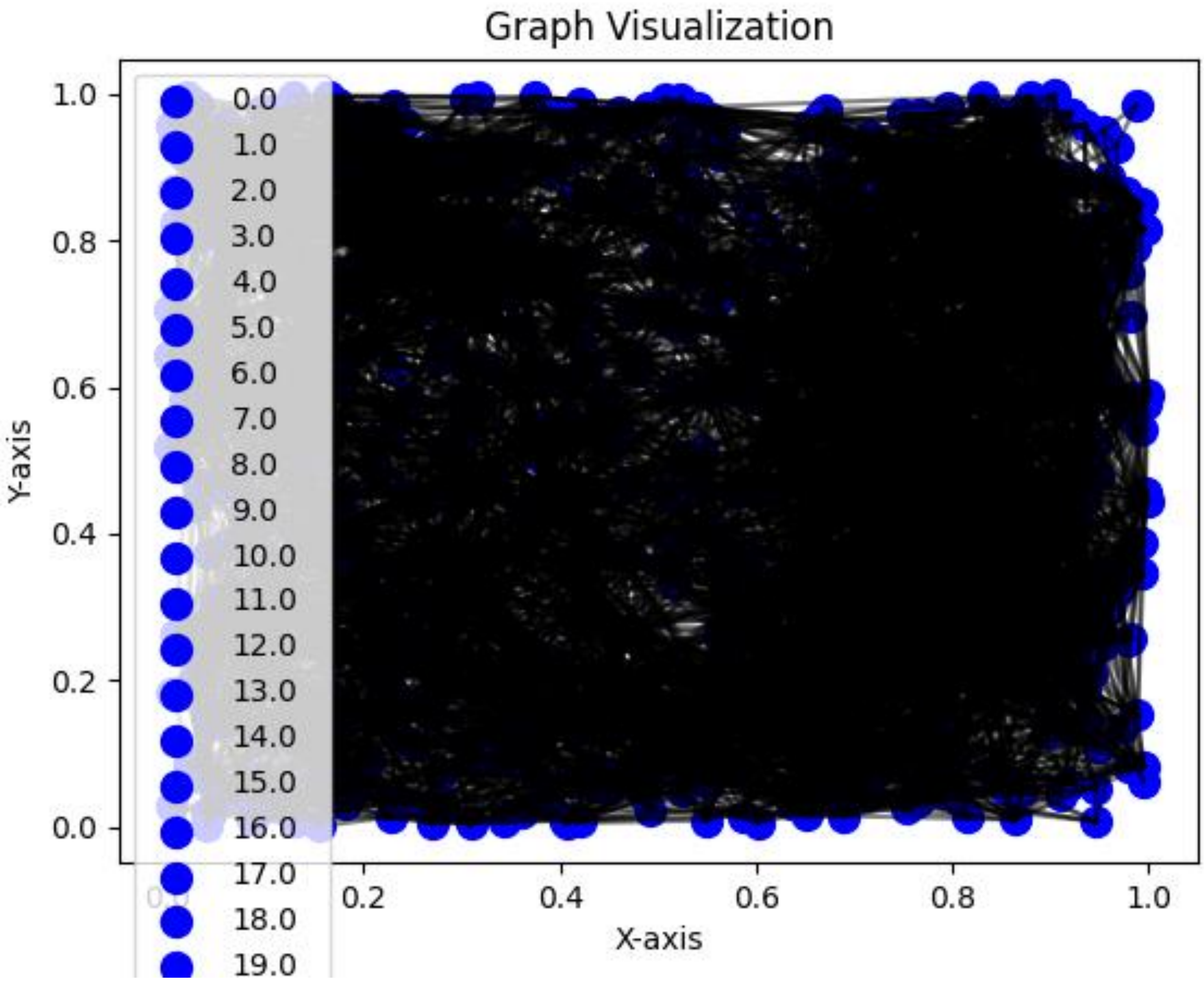
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	200	0.3	2	20	5.55	1	4060
DFS	200	0.3	2	20	103.85	1	4060
MAXCAP	200	0.3	2	20	9.7	1	4060
RANDOM	200	0.3	2	20	6	1.015	4060



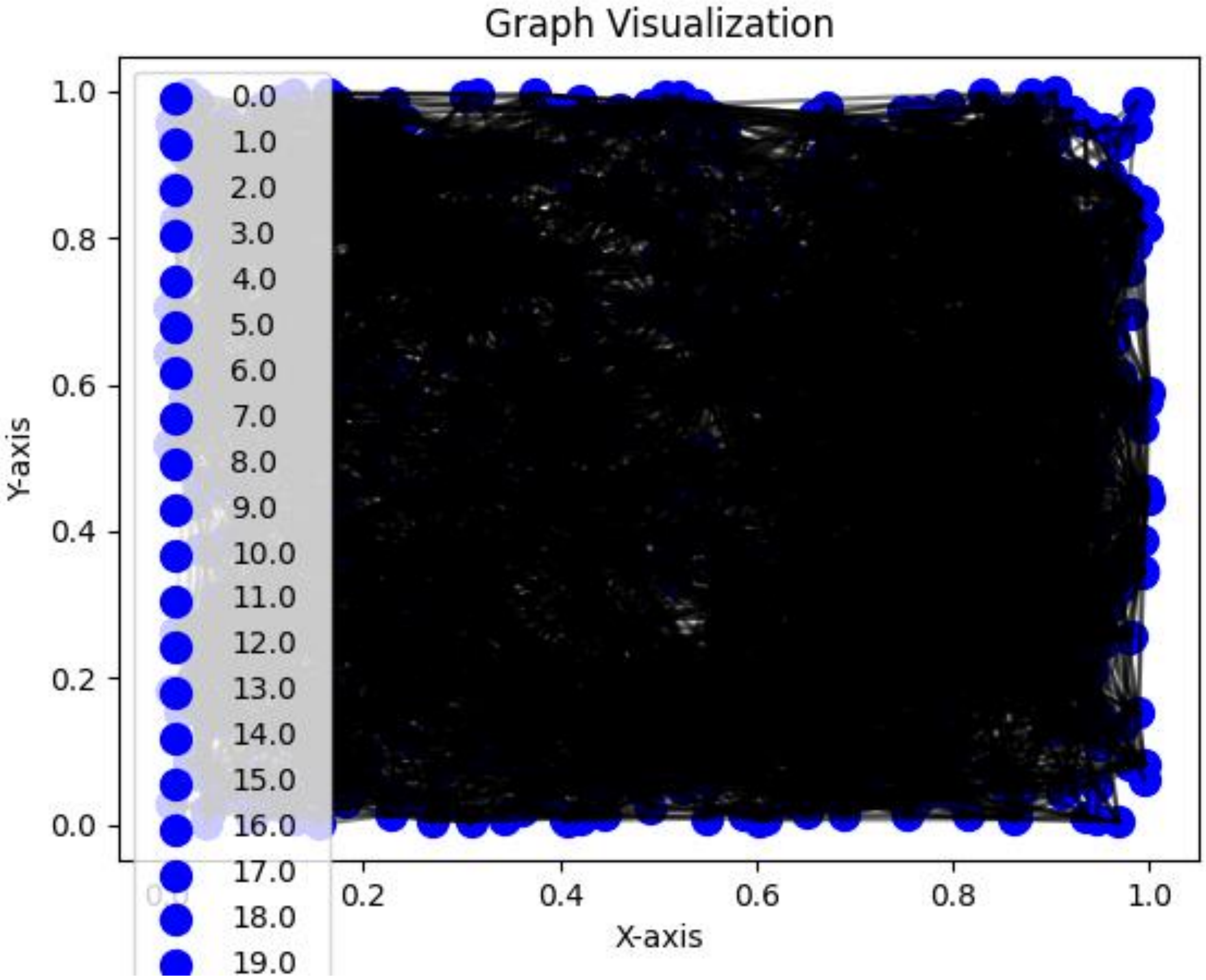
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	100	0.2	50	44	10.9	1	464
DFS	100	0.2	50	41	27.82	1	464
MAXCAP	100	0.2	50	44	15.13	1	464
RANDOM	100	0.2	50	43	11.9	1	464



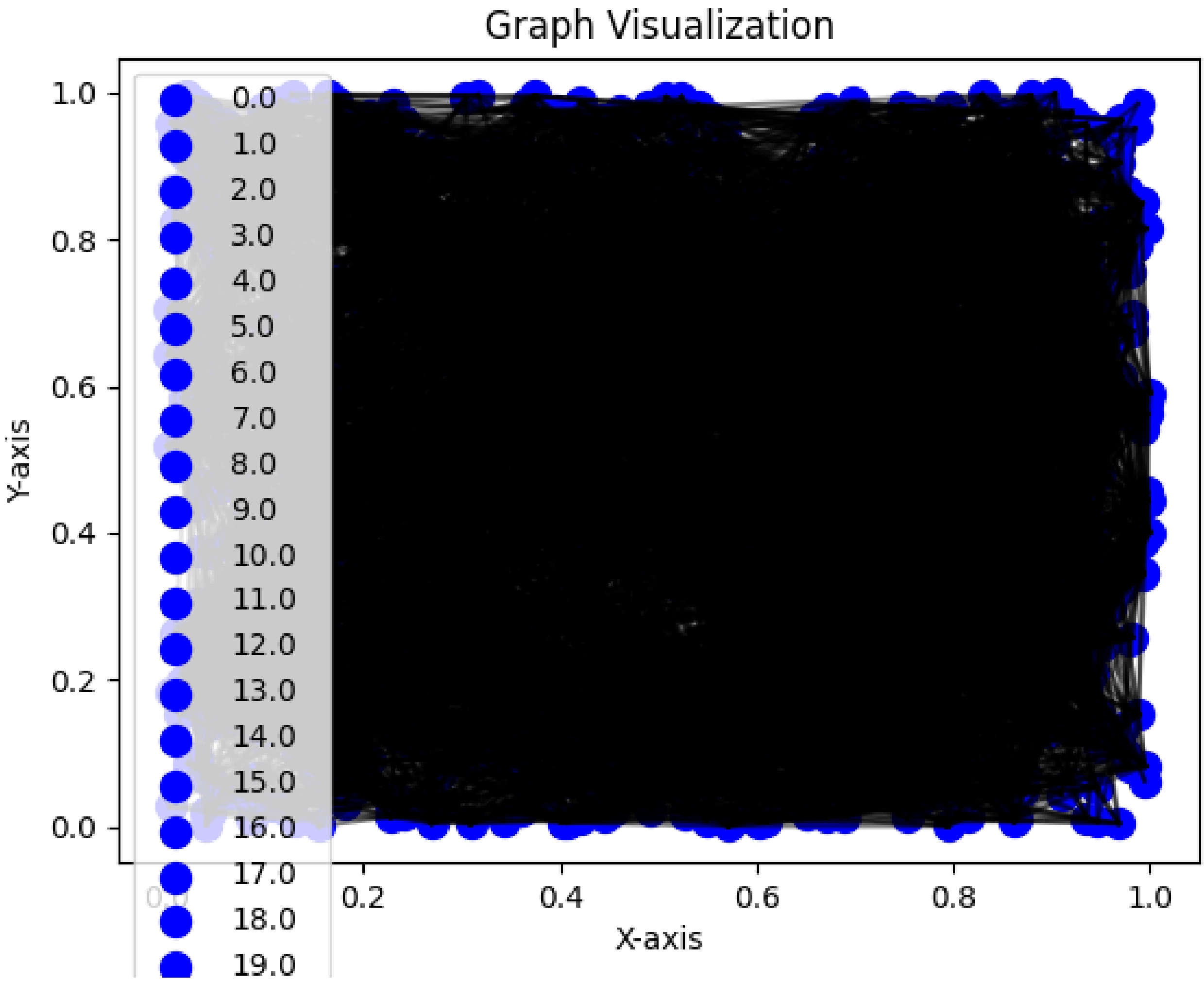
Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	200	0.2	50	414	9.62	1	2062
DFS	200	0.2	50	247	114.83	1	2062
MAXCAP	200	0.2	50	414	16.98	1	2062
RANDOM	200	0.2	50	414	12.75	1	2062



Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	100	0.3	50	36	6.33	1	982
DFS	100	0.3	50	36	37.58	1	982
MAXCAP	100	0.3	50	36	13.83	1	982
RANDOM	100	0.3	50	36	8.05	1	982



Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	200	0.3	50	442	5.35	1	4644
DFS	200	0.3	50	442	122.13	1	4644
MAXCAP	200	0.3	50	442	9.17	1	4644
RANDOM	200	0.3	50	442	7.76	1	4644



Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	150	0.2	100	118	9.31	1	1149
DFS	150	0.2	100	118	63.33	1	1149
MAXCAP	150	0.2	100	118	13.11	1	1149
RANDOM	150	0.2	100	118	12.355	1	1149

Simulations II

Based on the results you obtained above and the differences observed in the performance of the augmenting path algorithms, I have selected 3 sets of values for **n**, **r**, and **upperCap** that highlight the differences:

Set 1: Emphasizing Large Graph Size and High Capacities

- **n**: 300
- **r**: 0.4
- **upperCap**: 50
- **Justification:** This set emphasizes a larger graph size with a higher probability of nodes being connected (higher **r**) and higher capacities (**upperCap**). It can highlight how the algorithms scale with increased graph complexity and larger capacities.

Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	300	0.4	50	751	4.54	1	15169
DFS	300	0.4	50	751	230.31	1	15169
MAXCAP	300	0.4	50	751	7.23	1	15169
RANDOM	300	0.4	50	751	7.4	1	15169

Set 2: Emphasizing Moderate Graph Size and Varied Capacities

- **n**: 150
- **r**: 0.3
- **upperCap**: 20
- **Justification:** This set strikes a balance with a moderate graph size, moderate probability of nodes being connected, and varied capacities. It aims to provide insights into how the algorithms perform in scenarios with a mix of graph connectivity and capacity values.

Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	150	0.3	20	120	6.95	1	2461
DFS	150	0.3	20	118	89.44	1	2461
MAXCAP	150	0.3	20	120	10.575	1	2461
RANDOM	150	0.3	20	120	9.325	0.99	2461

Set 3: Emphasizing Small Graph size and Small Capacities

- **n**: 50
- **r**: 0.1
- **upperCap**: 2
- **Justification:** This set emphasizes a small graph size with a low probability of nodes being connected and Small capacities (**upperCap**). It can highlight how the algorithms fits and serve small configuration graph.

Algorithms	n	r	UpperCap	Paths	ML	MPL	Total Edges
SAP	50	0.1	2	2	3	1	38
DFS	50	0.1	2	2	3	1	38
MAXCAP	50	0.1	2	2	3	1	38
RANDOM	50	0.1	2	2	3	1	38

By running simulations with these sets, I have observed how the different augmenting path algorithms respond to changes in graph size, connectivity, and capacity. These sets are chosen to provide a diverse range of scenarios for evaluation.

Conclusions:

1. **Algorithm Performance with Different Parameters:**
   - **SAP (Shortest Augmenting Path):** Generally performs well and finds a small number of paths with relatively low mean path lengths. It tends to be efficient for smaller and less dense networks.
   - **DFS (Depth-First Search):** Demonstrates the highest mean path length among all algorithms, making it less efficient in finding augmenting paths. It may not be suitable for larger networks or denser graphs.
   - **MAXCAP (Maximum Capacity):** Offers a balance between efficiency and path length, making it a reasonable choice for various scenarios.
   - **RANDOM:** Shows competitive performance in terms of paths and mean path length, making it a viable alternative in scenarios where randomness can be beneficial.
2. **Impact of Network Size (n) and Density (r):**
   - As the network size (n) increases, the path lengths tend to increase for all algorithms. This suggests that finding augmenting paths becomes more challenging in larger networks.
   - Higher values of r (density) lead to shorter paths in general, as nodes are more likely to be interconnected. However, for extremely dense graphs, the algorithms may struggle due to the complexity of the network.
3. **Considerations for Large and Dense Networks:**
   - For much larger networks (higher n), algorithms like RANDOM and MAXCAP may be more suitable due to their adaptability and ability to handle complex graphs.
   - In scenarios with very dense networks (higher r), RANDOM and MAXCAP algorithms may still perform well, but careful experimentation and analysis are needed to choose the most appropriate algorithm.

References:

https://github.com/williamfiset/Algorithms/tree/master
https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
https://favtutor.com/blogs/ford-fulkerson-algorithm