Concordia University
Engineering and Computer Science

Comparative Study of Algorithms for Computing Augmenting Paths to Maximum Flow Problem

COMP 6651 – Algorithm Design Techniques

Fall 2023

Submitted To: Dr. Thomas Fevens

Submitted by:
Ms. Gurleen Kaur Pannu
(40289625)

December 11, 2023

GitHub: https://github.com/GurleenP06/COMP-6651-Project

TABLE OF CONTENTS

1. PROBLEM STATEMENT

The Max Flow problem is a central challenge in graph theory and optimization, particularly relevant in scenarios such as resource distribution, transportation, and communication networks. The problem is defined on a directed graph where each edge has a capacity limit, representing the maximum flow it can support. The essence of the problem is to determine the maximum amount of flow that can be transferred from the source to the sink while adhering to the capacity constraints of the edges. This flow optimization problem is not just theoretical; it has practical applications in optimizing network traffic, logistics, and resource management in various fields.

In addressing the Max Flow problem, various algorithms can be implemented, each with its unique approach to pathfinding in the directed graph. The project focuses on comparing four such algorithms: Random, DFS-like, Maximum Capacity (MaxCap), and Shortest Augmenting Path (SAP). Each algorithm is tested under different network conditions – varying numbers of nodes, edge weights, and capacity constraints – to evaluate their efficiency in finding augmenting paths. The performance metrics include the number of paths found, average path lengths, and the total edges visited. The aim is to understand the strengths and weaknesses of each algorithm in different scenarios.

Central to solving the Max Flow problem is the Ford-Fulkerson algorithm, which seeks to maximize the total flow from the source node to the sink node within the capacity constraints of the edges. The algorithm operates by initially starting with a feasible flow, usually zero, and then iteratively finding augmenting paths in the network. These paths are routes from the source to the sink with available capacity to increase the overall flow. The algorithm adjusts the flow along the augmenting path by the minimum capacity found on that path and subsequently updates the residual graph, reflecting the remaining capacity in the network. This process of finding and utilizing augmenting paths continues until no more paths can be found, signifying that the maximum flow has been achieved. This iterative approach provides a framework for understanding and optimizing flow in complex networks, making it a valuable tool in fields ranging from computer science to operations research.

2. IMPLEMENTATION DETAILS

This project harnesses the capabilities of the Python programming language, utilizing its robust libraries such as `heapq` for implementing priority queues and `csv` for managing data in ASCII format files. It is centered on the practical implementation of algorithms to determine maximum flows in networks with specified source and sink nodes. The cornerstone of this endeavor is the Ford-Fulkerson method, accompanied by four distinct algorithms for identifying augmenting paths: the Shortest Augmenting Path (SAP), DFS-like, Maximum Capacity (MaxCap), and Random algorithms. Each of these algorithms is critically examined for their efficiency and effectiveness in handling randomly generated source-sink graphs. This examination aims to provide insights into their comparative performance, revealing the strengths and potential limitations of each algorithm in diverse network scenarios. The use of Python and its libraries enables efficient handling of complex calculations and data management, making the study robust and scalable.

Below is a detailed outline of the implementation:

- **Sink-Source Graph Generator:**

  The Random Source-Sink Graph Generator function, ‘generate_sink_source_graph’, efficiently constructs random graphs for testing the max flow algorithms. It uses three parameters: ‘n’ for the number of vertices, ‘r’ for the maximum distance between connected nodes, and ‘upperCap’ for setting the edge capacity. The function places vertices randomly in a space and connects them based on their Euclidean distances, ensuring that the distance between connected nodes does not exceed ‘r’. Edge capacities are then randomly assigned integer values, with the maximum possible capacity being ‘upperCap’. This approach to graph generation offers a varied set of scenarios to rigorously test and compare the performance of the different algorithms under diverse network conditions.

- **Ford Fulkerson Algorithm:**

  The 'ford_fulkerson' function embodies the implementation of the Ford-Fulkerson algorithm in this project. This function operates by iteratively identifying augmenting paths using one of the chosen augmenting path algorithms. Upon each discovery, it updates the residual graph, which reflects the current capacity and flow status of the network. The process continues until the algorithm can no longer find any viable augmenting paths, indicating that the maximum flow from the source to the sink node has been reached. This methodical approach allows for a thorough exploration of the network's flow capacity, ensuring the efficient and accurate determination of maximum flow.

- **Augmenting Path Algorithms:**
  For this project, we have implemented four different kinds of algorithms for generating the augmenting path for the generated graphs, where each algorithm is a variation of the Dijkstra algorithm.

  - **Shortest Augmenting Path (SAP):**
    To update the distance (v.g) and predecessor (v.π) attributes for every node in the graph, the algorithm essentially performs a breadth-first search (BFS), beginning at the source node s. Finding the shortest routes between the source and every other node in the graph is the objective. The source node is used by the algorithm to initialise a priority queue. Nodes that are close to the source are found and added to the queue during the first iteration. In the following iterations, the algorithm searches its neighbouring nodes for the node with the lowest value, extracts it from the queue, and, in the event that v.g > u.g + 1, executes the RELAX operation. Until all pertinent nodes have been processed, this process keeps going.

    ShortestAugmentingPath(G, s, t)

    ```
    for each v ∈ G.E do
        v.g = 0
        v.π = NIL
    s.g = 0
    S = ∅
    Q = G.E
    while Q /= ∅ do
        u = EXTRACT -MIN(Q)
        S = S ∪ u
        for each v ∈ G.adj[u] do
            if v.g > u.g + 1 then
                v.g = u.g + 1
                v.π = u
    ```

  - **DFS-like:**
    The algorithm is a Dijkstra modification that includes a node exploration strategy akin to Depth-First Search (DFS). The source node and the algorithm start a priority queue. Nodes close to the source are discovered and added to the queue during the first iteration, with a decreasing counter value used to determine their priority. In the following iterations, the algorithm looks through the queue for the node with the lowest counter value, investigates the nodes that are next to it, and updates v.g when v.g = ∞. Until all relevant nodes are processed, this iterative procedure continues.

DFSLike(G, s, t)

```
1. for each v ∈ G.E do
2.     v.g = 0
3.     v.π = NIL
4. s.g = ∞
5. counter = 2 * |V|
6. S = ∅
7. Q = G.E
8. while Q /= ∅ do
9.     u = EXTRACT -MIN(Q)
10.    S = S ∪ u
11.    for each v ∈ G.adj[u] do
12.        if v.g = ∞ then
13.            v.g = counter - 1
14.            v.π = u
15.    counter = counter - 1
```

- **Maximum Capacity (MaxCap):**
  To implement the Ford-Fulkerson algorithm with the Maximum Capacity (MaxCap) strategy for finding augmenting paths, we will modify Dijkstra's algorithm to select paths based on the maximum capacity rather than the shortest distance. This approach focuses on maximizing the flow in each augmenting path by choosing the path with the highest minimum edge capacity.

MaximumCapacity(G, s, t)

```
1. for each v ∈ G.E do
2.     v.g = 0
3.     v.π = NIL
4. s.g = ∞
5. S = ∅
6. Q = G.E
7. while Q /= ∅ do
8.     u = EXTRACT -MAX(Q)
9.     S = S ∪ u
10.    for each v ∈ G.adj[u] do
11.        if u.g < res(u, v) then
12.            temp = u.g
13.        else
14.            temp = res(u, v)
15.        if v.g < temp then
16.            v.g = temp
17.            v.π = u
```

- **Random:**
  We'll alter the path-finding strategy to use random values as the key for prioritising nodes in the search in order to implement the Ford-Fulkerson algorithm with a Random strategy for finding augmenting paths. This approach, in contrast to more deterministic methods like DFS-like or Dijkstra's algorithm, introduces a degree of randomness in the order in which nodes are explored.

  Random(G, s, t)

  ```
  for each v ∈ G.E do
      v.g = 0
      v.π = NIL
  s.g = ∞
  S = ∅
  Q = G.E
  while Q /= ∅ do
      u = EXTRACT -MIN(Q)
      S = S ∪ u
      for each v ∈ G.adj[u] do
          temp = RANDOM - INT(1, 500)
          if v.g = ∞ then
              v.g = temp
              v.π = u
  ```

- **Simulation and Results:**
  In this project, a series of simulations are conducted across a spectrum of graph configurations to rigorously test the performance of the different algorithms. The results from these simulations are meticulously tabulated for comparative analysis. Key metrics are calculated for each algorithm in every simulation, providing a comprehensive view of their effectiveness. These metrics include the total number of edges traversed, the mean length of the augmenting paths, the mean proportional length (which offers insights into the efficiency of the paths relative to the graph size), and the number of augmenting pathways identified. This data-driven approach enables a nuanced understanding of each algorithm's strengths and limitations in various network scenarios, providing valuable insights into their practical applicability in solving the max flow problem.

- **Graph Input and Output:**
  To enhance the repeatability and persistence of the experiments, the project incorporates the ‘read_graph_from_csv’ and ‘write_graph_to_csv’ functions. These functions facilitate the saving and loading of graph configurations to and from CSV files, ensuring that experiments can be consistently replicated. This feature is

particularly valuable for comparative studies, as it allows for the exact same graph configurations to be used across different algorithm tests. By maintaining a record of the graphs in a universally accessible format like CSV, researchers can easily share their test scenarios, promoting collaborative verification and further experimentation. This approach not only ensures the integrity and consistency of the experiments but also significantly contributes to the ease of conducting extensive comparative analyses in the study of max flow algorithms.

- **Simulations:**
  In this study, two distinct simulation setups are employed to underscore the differences between the algorithms. Simulation I utilizes predefined settings, ensuring a standardized environment for initial algorithm testing. In contrast, Simulation II allows for custom configurations, enabling a more nuanced examination of how each algorithm performs under varied and specific conditions. The results from both simulations are presented in a tabular format, facilitating a clear and direct comparison.

- **Observations:**
  The outcomes of the simulations provide insightful observations on the performance of the methods based on several key indicators. These include the total number of pathways each method discovers, the mean length of these pathways, the mean proportional length which gives an insight into the efficiency of the paths relative to the overall size of the graph, and the total number of edges traversed during the process. Analyzing these metrics allows for a detailed understanding of each algorithm's efficiency and effectiveness in navigating and optimizing flow within the network. Such comprehensive data helps in determining not just the capability of each method in finding the maximum flow, but also in assessing their operational nuances in different graph configurations.

3. IMPLEMENTATION CORRECTNESS

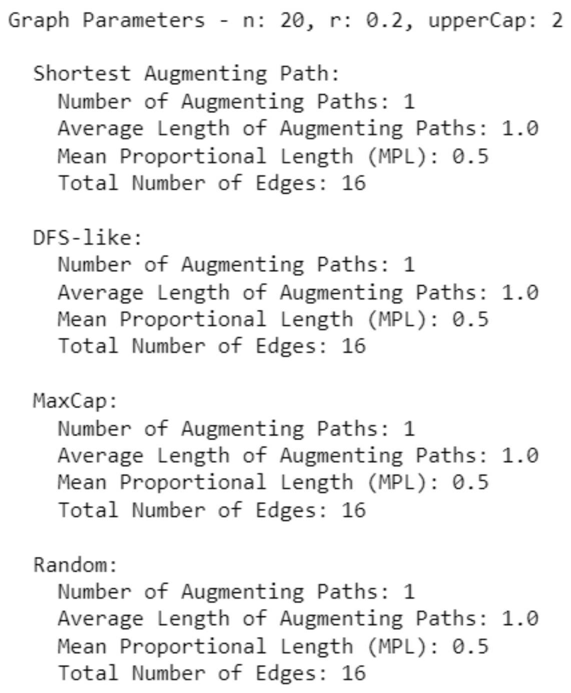
To ensure the correct implementation of the Ford-Fulkerson algorithm, a detailed and systematic testing process is conducted using a small, controlled graph. This process is designed to verify the accuracy and reliability of the algorithm across different augmenting path strategies.

A. **Small Graph Creation:** A graph with 20 vertices, a radius of 0.2, and upper capacity limit of 2 is generated. The small size of the graph facilitates easy manual verification of results.

B. **Algorithm Application:** The Ford-Fulkerson method is applied to this network using four augmenting path algorithms: Shortest Augmenting Path (SAP), DFS-like, Maximum Capacity (Max-Cap), and Random Path.

C. **Result Verification:** The outputs from each algorithm are compared with manually calculated expected results, focusing on metrics like the number of augmenting paths, average path length, mean proportional path length, and the total number of edges.

D. **Consistency Evaluation:** The consistency of results across multiple runs of each augmenting path algorithm is assessed. Consistent results indicate accurate and reliable algorithm implementation.


```
Graph Parameters - n: 20, r: 0.2, upperCap: 2

  Shortest Augmenting Path:
    Number of Augmenting Paths: 1
    Average Length of Augmenting Paths: 1.0
    Mean Proportional Length (MPL): 0.5
    Total Number of Edges: 16

  DFS-like:
    Number of Augmenting Paths: 1
    Average Length of Augmenting Paths: 1.0
    Mean Proportional Length (MPL): 0.5
    Total Number of Edges: 16

  MaxCap:
    Number of Augmenting Paths: 1
    Average Length of Augmenting Paths: 1.0
    Mean Proportional Length (MPL): 0.5
    Total Number of Edges: 16

  Random:
    Number of Augmenting Paths: 1
    Average Length of Augmenting Paths: 1.0
    Mean Proportional Length (MPL): 0.5
    Total Number of Edges: 16
```

4. RESULTS

Parameter Selection:

The chosen parameters (n, r, upperCap) are designed to encompass a variety of situations, allowing for a comprehensive assessment of the algorithm's effectiveness in diverse environments.

Parameters for Simulation I:

- (100, 0.2, 2): Tests the algorithm on a small, sparsely connected graph.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	100	0.2	2	5	8.6	0.95	513
DFS-like	100	0.2	2	5	16	1.78	513
MaxCap	100	0.2	2	4	15.75	1.75	513
Random	100	0.2	2	5	48	5.33	513

- (200, 0.2, 2): Expands the number of vertices from the first scenario to explore how the algorithm scales.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	200	0.2	2	3	8	0.8	2114
DFS-like	200	0.2	2	3	18.67	1.87	2114
MaxCap	200	0.2	2	2	20	2	2114
Random	200	0.2	2	3	97	9.7	2114

- (100, 0.3, 2): Increases the connection radius to create a denser network on a small graph.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	100	0.3	2	2	5	0.71	1056
DFS-like	100	0.3	2	2	12.5	1.78	1056
MaxCap	100	0.3	2	2	7	1	1056
Random	100	0.3	2	2	54	7.71	1056

- (200, 0.3, 2): A larger, denser graph to evaluate scalability in more intricate networks.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	200	0.3	2	20	3.85	0.77	4053
DFS-like	200	0.3	2	21	18.28	3.66	4053
MaxCap	200	0.3	2	13	6.69	1.34	4053
Random	200	0.3	2	21	128.28	25.66	4053

- (100, 0.2, 50): A small graph with an increased upper capacity limit to test the algorithm with higher-capacity edges.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	100	0.2	50	15	9.73	1.08	549
DFS-like	100	0.2	50	38	17.55	1.95	549
MaxCap	100	0.2	50	5	15	1.67	549
Random	100	0.2	50	80	57.81	6.42	549

- (200, 0.2, 50): A larger version of the previous scenario, focusing on larger graph size.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	200	0.2	50	6	7.83	0.87	2097
DFS-like	200	0.2	50	14	22.78	2.53	2097
MaxCap	200	0.2	50	4	17	1.89	2097
Random	200	0.2	50	33	124.48	13.83	2097

- (100, 0.3, 50): Combines higher density with higher capacity for a more challenging test.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	100	0.3	50	25	5.72	0.82	1017
DFS-like	100	0.3	50	66	14.04	2	1017
MaxCap	100	0.3	50	10	10.5	1.5	1017
Random	100	0.3	50	192	67.88	9.70	1017

- (200, 0.3, 50): A large, complex graph with both increased density and higher capacity edges.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	200	0.3	50	40	5.45	0.78	4155
DFS-like	200	0.3	50	123	18.02	2.57	4155
MaxCap	200	0.3	50	13	10.61	1.52	4155
Random	200	0.3	50	280	139.46	19.92	4155

Simulation II - Tailored Scenarios:

- (50, 0.2, 5): A small graph with moderate connectivity is chosen.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	50	0.2	5	2	7	0.78	122
DFS-like	50	0.2	5	1	8	0.89	122
MaxCap	50	0.2	5	1	8	0.89	122
Random	50	0.2	5	1	9	1	122

- (150, 0.3, 50): Focuses on a larger graph with enhanced connectivity.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	150	0.3	50	40	4.62	0.77	2508
DFS-like	150	0.3	50	113	16.53	2.75	2508
MaxCap	150	0.3	50	12	7.58	1.26	2508
Random	150	0.3	50	250	95.22	15.87	2508

- (100, 0.15, 25): A medium-sized graph with a reduced connection radius, testing the algorithm in a network with more localized connections.

Algorithm	n	r	upperCap	Paths	ML	MPL	Total Edges
SAP	100	0.15	25	2	8.5	0.85	302
DFS-like	100	0.15	25	5	10.8	1.08	302
MaxCap	100	0.15	25	1	10	1	302
Random	100	0.15	25	8	17.75	1.77	302

5. CONCLUSION

Analyzing the results of the Simulations can provide insights into how these factors influence the performance and behavior of different augmenting path strategies in the Ford-Fulkerson algorithm. Here's an analysis based on the data:

A. Impact of Graph Size (n):

- As ‘n’ increases from 100 to 200, there's a noticeable increase in the number of edges in the graph, which is expected given the larger number of possible connections.
- The number of paths required and the mean length of paths generally increase with the graph size. This is more pronounced in the Random strategy, suggesting that larger graphs lead to more complex augmenting paths.
- The ‘SAP’ strategy consistently requires fewer paths and has a shorter mean path length, indicating its efficiency in both small and large graphs.

B. Impact of Connection Radius (r):

- Increasing ‘r’ from 0.2 to 0.3 significantly increases the graph’s density, as seen in the ‘Total Edges’. A denser graph typically provides more augmenting path options.
- In denser graphs, ‘SAP’ and ‘MaxCap’ strategies tend to find shorter paths indicating these strategies are more efficient in utilizing the additional connections between the graph.

C. Impact of Upper Capacity Limit (upperCap):

- Increasing ‘upperCap’ from 2 to 50 doesn't significantly change the ‘Total Edges’ but affects the capacity of these edges.
- Higher capacities lead to fewer required paths in ‘MaxCap’ and ‘SAP’, suggesting these strategies are more effective in utilizing high-capacity edges.
- The ‘Random’ strategy shows a substantial increase in the mean length of paths with higher capacities, possibly due to the randomness leading to less efficient use of high-capacity edges.

D. Mean Proportional Length (MPL):

- ‘MPL’ is relatively stable across different configurations for ‘SAP’ and ‘MaxCap’, suggesting that these strategies maintain a proportionate path length despite changes in graph size or density.
- The ‘Random’ and ‘DFS-like’ strategies exhibit a higher ‘MPL’ in more complex graphs, indicating less efficiency in the path selection (w.r.t. MPL) relative to the longest acyclic path.

E. **Comparison of Strategies:**

- ‘SAP’ generally performs consistently across different graph configurations, indicating its robustness.
- ‘DFS-like’ tends to require more paths and longer paths on average, especially as graph complexity increases.
- ‘MaxCap’ performs well in high-capacity scenarios, suggesting its effectiveness in such contexts.
- The ‘Random’ strategy shows the most variability, with significantly longer paths in larger and more complex graphs, reflecting the inherent unpredictability of this approach.

To conclude, these simulations show that the choice of augmenting path strategy in the Ford-Fulkerson algorithm can significantly affect its performance, especially in terms of the number of paths required and the efficiency of these paths. The \`SAP\` and \`MaxCap\` strategies appear to be more robust and efficient across various graph configurations, while the \`Random\` strategy, although interesting for its unpredictability, may not be as efficient, particularly in larger and more complex networks.

6. FUTURE SCOPE

Moving ahead in the future we can expect to implement the following:

- **Efficiency Improvements:** Refine current algorithms to enhance their performance with larger and more complex graph structures.

- **Algorithm Development:** Investigate innovative algorithms or modify existing ones to boost both precision and processing speed.

- **Practical Implementation:** Utilize these algorithms in real-world contexts to evaluate their practical utility.

- **Comparative Studies:** Perform comprehensive comparisons with other network flow algorithms to understand relative strengths and weaknesses.

- **Hybrid Modeling Approaches:** Combine these algorithms with different modeling techniques to tackle multifaceted challenges.

- **Incorporating Machine Learning:** Investigate the potential of merging machine learning techniques for dynamic adaptation of algorithm parameters.

7. REFERENCES

1. Lecture Materials for Algorithm Reference and Learning.
2. “Lecture #11: Network Flows I 1 The Maximum Network Flow Problem,” 2023.
   Available: https://www.cs.cmu.edu/~15451-s23/lectures/lec11-flow1.pdf
3. K. Wayne, “Lecture slides.”
   Available: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/07NetworkFlowI.pdf
4. “Lecture 10,” 2011. Accessed: Dec. 12, 2023. [Online].
   Available: https://theory.stanford.edu/~trevisan/cs261/lecture10.pdf
5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). The MIT Press.