Max Flow: Exploring and Analyzing Flow Problem using Algorithms

Mahwish Ahmed
ma07174@st.habib.edu.pk
Habib University
Karachi, Sindh, Pakistan

Owais Waheed
ow07611@st.habib.edu.pk
Habib University
Karachi, Sindh, Pakistan

Abdullah Junejo
aj0154@st.habib.edu.pk
Habib University
Karachi, Sindh, Pakistan

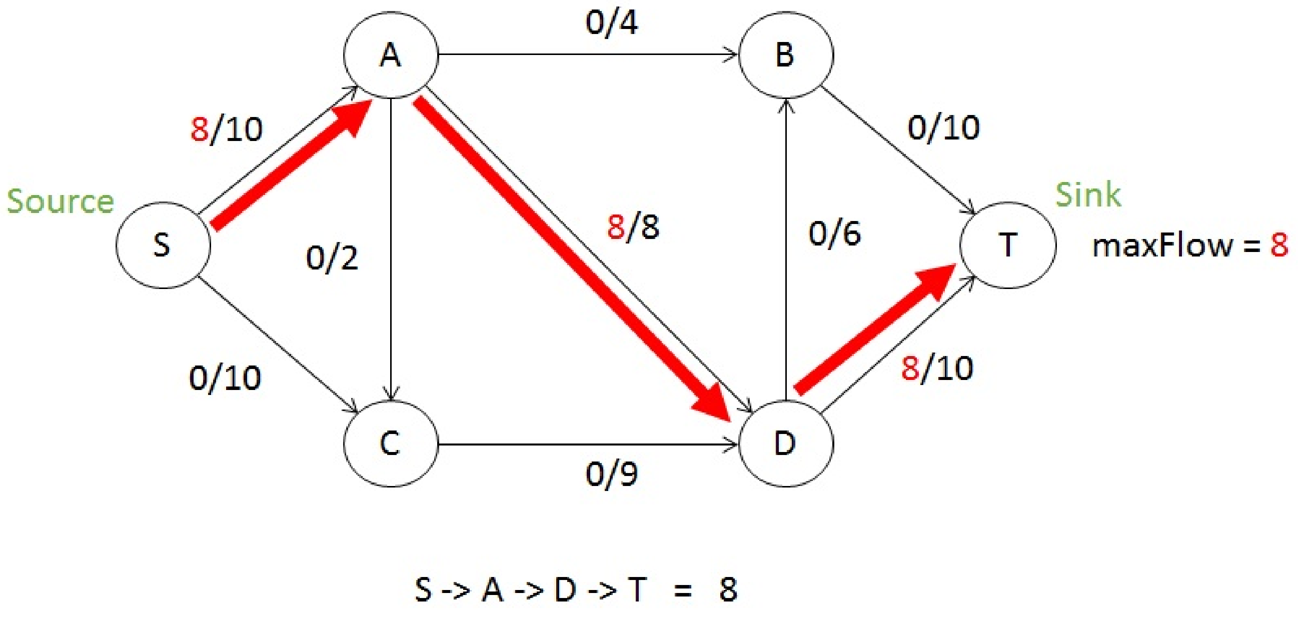


Figure 1: Example of Max Flow in a Graph

ABSTRACT

The Maximum Flow Problem is a fundamental challenge in computer science and graph theory, with applications in network flow optimization. This report presents a comprehensive study and implementation of three prominent algorithms for solving this problem: Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm. These algorithms were implemented in a common programming language, and their performances were evaluated based on theoretical time complexity analysis, empirical measurements, and visual representations. Through rigorous experimentation and analysis, the report compares the strengths and weaknesses of these algorithms, considering their design techniques, time complexities, and practical performance. Visual implementations offer insights into the algorithms' workflows and behaviors. The findings contribute to a deeper understanding of the trade-offs involved in solving the Maximum Flow Problem and aid in selecting the most suitable algorithm based on specific requirements and constraints. This comprehensive analysis serves as a valuable resource for researchers, practitioners, and students working with network flow optimization and related graph theory problems.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Maximum Flow Problem, Network Flow, Graph Theory, Ford-Fulkerson Algorithm, Edmonds-Karp Algorithm, Dinic's Algorithm, Flow Networks, Augmenting Paths, Residual Graphs, Time Complexity Analysis, Visual Algorithms, Optimization

ACM Reference Format:
Mahwish Ahmed, Owais Waheed, and Abdullah Junejo. 2024. Max Flow: Exploring and Analyzing Flow Problem using Algorithms. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Maximum Flow Problem is a fundamental problem in computer science and graph theory, with numerous applications in network flow optimization, transportation planning, and resource allocation. It involves determining the maximum amount of flow that can be sent from a designated source vertex to a designated sink vertex in a flow network, subject to capacity constraints on the edges.

This report presents a comprehensive study and implementation of three prominent algorithms for solving the Maximum Flow Problem: Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm. These algorithms employ different approaches and techniques to find the maximum flow in a given flow network efficiently.

2 MOTIVATION

The Maximum Flow Problem has wide-ranging applications in various domains, making it a problem of significant practical importance. Some motivations for studying and implementing efficient algorithms for solving the Maximum Flow Problem include:

- **Network Optimization**: In communication networks, the Maximum Flow Problem can be used to determine the maximum data transfer rate between nodes, optimizing bandwidth utilization and improving network efficiency.

3 IMPLEMENTATION & METHODOLOGY

3.1 Algorithm Implementations

We implemented three algorithms in Python for solving maximum flow problems: Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm. Each algorithm offers a different approach to finding augmenting paths and computing the maximum flow in a network.

3.2 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm is a method for computing the maximum flow in a flow network. It repeatedly finds augmenting paths using Depth-First Search (DFS) and augments the flow along those paths until no more augmenting paths exist. This section presents the DFS implementation for finding shortest paths and the overall implementation of the Ford-Fulkerson algorithm.

3.2.1 *Depth-First Search (DFS) for Finding Shortest Paths.*

The Depth-First Search (DFS) algorithm is utilized to find augmenting paths in the residual graph during each iteration of the Ford-Fulkerson algorithm. This DFS implementation explores the graph in a depth-first manner, starting from the source node and traversing along edges until it reaches the target node or a dead end. It marks nodes as visited and updates their parent pointers to trace back the path found.

```

def dfs_shortest_path(Gf, s, t, parent):
    visited = [False] * len(Gf)
    stack = [s]
    visited[s] = True
    parent[s] = -1

    while stack:
        u = stack.pop()
        for v, capacity in enumerate(Gf[u]):
            if not visited[v] and capacity >
                 0:
                stack.append(v)
                parent[v] = u
                visited[v] = True
    return visited[t]
```

3.2.2 *Ford-Fulkerson Algorithm Implementation.*

The Ford-Fulkerson algorithm iteratively finds augmenting paths using DFS and updates the flow in the network until no more augmenting paths can be found. This implementation maintains a residual graph to keep track of remaining capacity and flow. It terminates when no more augmenting paths can be found from the source to the sink nodes.

```
def ford_fulkerson(G, s, t):
    n = len(G)
    f = [[0] * n for _ in range(n)]
    Gf = [row[:] for row in G]

    while True:
        parent = [-1] * n
        if not dfs_shortest_path(Gf, s, t,
            parent):
            break

        path_flow = float("inf")
        v = t
        while v != s:
            u = parent[v]
            path_flow = min(path_flow, Gf[u
                ][v])
            v = u

        v = t
        while v != s:
            u = parent[v]
            f[u][v] += path_flow
            f[v][u] -= path_flow
            v = u

        Gf = [[G[u][v] - f[u][v] for v in
            range(n)] for
        u in range(n)]

    return sum(f[s])
```

3.3 Edmonds-Karp Algorithm

The Edmonds-Karp algorithm is a specific implementation of the Ford-Fulkerson algorithm that uses Breadth-First Search (BFS) to find augmenting paths. This section presents the BFS implementation for finding shortest paths and the overall implementation of the Edmonds-Karp algorithm.

3.3.1 *Breadth-First Search (BFS) for Finding Shortest Paths.*

The Breadth-First Search (BFS) algorithm is employed to find augmenting paths in the residual graph during each iteration of the Edmonds-Karp algorithm. This BFS implementation explores the graph in a breadth-first manner, starting from the source node and traversing level by level until it reaches the target node or a dead end. It marks nodes as visited and updates their parent pointers to trace back the path found.

```
def bfs_shortest_path(Gf, s, t, parent):
    visited = [False] * len(Gf)
    queue = deque([s])
    visited[s] = True
    parent[s] = -1

    while queue:
        u = queue.popleft()
        for v, capacity in enumerate(Gf[u]):
            if not visited[v] and capacity >
                 0:
                queue.append(v)
                visited[v] = True
                parent[v] = u
    return visited[t]
```

3.3.2 ***Edmonds-Karp Algorithm Implementation.*** The Edmonds-Karp algorithm is a variation of the Ford-Fulkerson algorithm that uses BFS to find augmenting paths. It iteratively finds augmenting paths using BFS and updates the flow in the network until no more augmenting paths can be found. This implementation maintains a residual graph to keep track of remaining capacity and flow. It terminates when no more augmenting paths can be found from the source to the sink nodes.

```
def edmonds_karp(G, s, t):
    n = len(G)
    f = [[0] * n for _ in range(n)]
    Gf = [row[:] for row in G]

    while True:
        parent = [-1] * n
        if not bfs_shortest_path(Gf, s, t,
            parent):
            break

        path_flow = float("inf")
        v = t
        while v != s:
            u = parent[v]
            path_flow = min(path_flow, Gf[u
                ][v])
            v = u

        v = t
        while v != s:
            u = parent[v]
            f[u][v] += path_flow
            f[v][u] -= path_flow
            v = u

        Gf = [[G[u][v] - f[u][v] for v in
            range(n)] for
        u in range(n)]

    return sum(f[s])
```

3.4 Dinic's Algorithm

Dinic's algorithm is another variation of the Ford-Fulkerson algorithm that employs a different approach to finding augmenting paths. This section presents the DFS implementation for finding augmenting paths and the overall implementation of Dinic's algorithm.

3.4.1 ***Depth-First Search (DFS) for Finding Shortest Paths.*** The Depth-First Search (DFS) algorithm is utilized to find augmenting paths in the residual graph during each iteration of Dinic's algorithm. This DFS implementation explores the graph in a depth-first manner, starting from the source node and traversing as far as possible along each branch before backtracking. It marks nodes as visited and updates their parent pointers to trace back the path found.

Click here to refer to the Depth-First Search (DFS) section in the Ford-Fulkerson Algorithm(3.2.1)

3.4.2 ***Dinic's Algorithm Implementation.*** Dinic's algorithm is a faster variant of the Ford-Fulkerson algorithm that utilizes the concept of level graphs and blocking flow to efficiently find augmenting paths. It iteratively constructs level graphs and finds blocking flows until no more blocking flows can be found. This implementation maintains a residual graph to keep track of remaining capacity and flow. It terminates when no more augmenting paths can be found from the source to the sink nodes.

```

def DinicMaxflow(self, source, sink):
    parent = [-1] * self.V
    max_flow = 0

    while self.BFS(source, sink, parent):
        path_flow = float('Inf')
        s = sink

        while s != source:
            u, index = parent[s]
            path_flow = min(path_flow, self.
                adj[u][index][1])
            s = u

        s = sink
        while s != source:
            u, index = parent[s]
            self.adj[u][index][1] -=
                path_flow
            self.adj[s][self.adj[u][index
                ][2]][1] += path_flow
            s = u

        max_flow += path_flow

    return max_flow
```

4 DESIGN OF EXPERIMENTS

4.1 Experimental Objectives

The primary objective of this experiment was to empirically analyze and compare the performance of three maximum flow algorithms—Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm—under varying graph sizes. The key focus was on measuring and comparing their execution times and associated computational complexities using randomized graphs with controlled parameters.

4.2 Experimental Setup

4.2.1 *Input Sizes.*

- **Number of Vertices (V)**:
  The experiment was conducted using random graphs with varying numbers of vertices. For the first set of experiments, Vertices were systematically increased in increments of 100, ranging from 10 to 10,000 vertices. and for the second set of experiments they were kept constant at 1000.
- **Edge Factor**:
  Edge factor is defined as number of edges per vertex. It is used to measure the density of graph. The number of edges (E) was computed using edge factor as $E = \text{edge factor} \times V$, ensuring a consistent graph density across different input sizes.
  For the first set of experiments, the edge factor was kept constant at 10. Later it was systematically increased in increments of 10 from 10 to 100 for ford fulkerson and for comparison of dinic and Edmond it was increased in increment of 100 from 100 to 1000.

4.2.2 *Programming Language and Tools.*

- **Programming Language**:
  - Python (version 3.x) was selected as the primary programming language for implementing the algorithms and conducting the experiments.
  - Python's extensive libraries, including `matplotlib` for visualization and `time` for performance measurement, facilitated efficient experimentation and data analysis.

4.2.3 *Hardware Environment.*

- **Machine Specifications**:
  The experiments were conducted on a machine equipped with the following specifications:
  - Processor: Intel Core i7 (8th Gen)
  - RAM: 16 GB
  - Operating System: Windows 10

4.2.4 *Experimental Procedure.*

(1) **Graph Generation**:
Random graphs were generated using the following function, specifying the number of vertices and computing the corresponding number of edges based on the edge factor.Each graph was then saved to a file for subsequent loading and analysis.

```
def random_graph(nodes, edges, capacity):
    with open(file, 'w') as f:
f.write(f"{nodes} {edges}\n")
for _ in range(num_edges):
 source = random.randint(0, num_nodes - 1)
 target = random.randint(0, num_nodes - 1)
     while target == source:
   target = random.randint(0, num_nodes - 1)
capacity = random.randint(1, max_capacity)
f.write(f"{source} {target} {capacity}\n")
```

(2) **Algorithm Execution**:
- Graphs were loaded into separate instances of the FordGraph, EdmondsGraph, and DinicsGraph classes.
- The execution time for each algorithm was measured using the `time` module to compute the maximum flow and associated performance metrics (e.g., number of residual graphs, layered networks).

(3) **Data Collection and Analysis**:
- Execution times and performance metrics were collected for each algorithm across multiple graph sizes.
- Data analysis involved computing and comparing the average execution times, plotting graphs to visualize the results, and assessing the scalability and efficiency of each algorithm with increasing input sizes.

4.3 Experimental Constraints and Considerations

- **Resource Limitations**:
  The primary resource constraints included memory limitations for large graph sizes and processing power constraints for computationally intensive algorithms. Due to lack of resources the maximum number of edges used was restricted to 100,000.

5 ANALYSIS

5.1 Theoretical Analysis

This section examines the time complexities associated with the Ford-Fulkerson, Edmonds-Karp, and Dinic's Algorithm for maximum flow. Understanding these complexities helps us evaluate how each algorithm might perform under different computational conditions.

Ford-Fulkerson Algorithm
The time complexity of the Ford-Fulkerson algorithm is $O(E|f|)$, where E is the number of edges in the graph and $|f|$ is the maximum flow. This complexity arises from the iterative path-finding process in the residual graph. The algorithm will keep finding path till the maximum flow is reached. In worst case, the flow will increase by a unit in every iteration hence it's complexity is $O(E|f|)$.

Edmonds-Karp Algorithm
The Edmonds-Karp algorithm, a variant of Ford-Fulkerson using BFS to find augmenting paths, has a time complexity of $O(VE^2)$, where V is the number of vertices and E is the number of edges. In comparison to Ford-Fulkerson, the number of iterations are not dependent upon maxflow but there are VE iterations max as proved

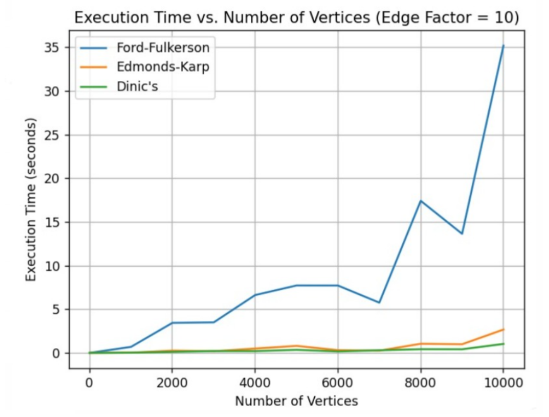
in CLRS [2]. Since each iteration takes $O(E)$ time the complexity is $O(VE^2)$.

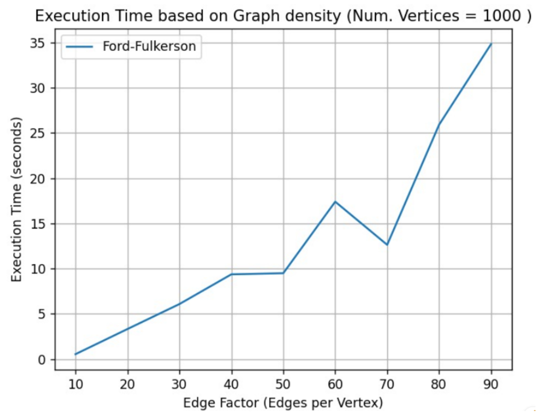
Dinic's Algorithm

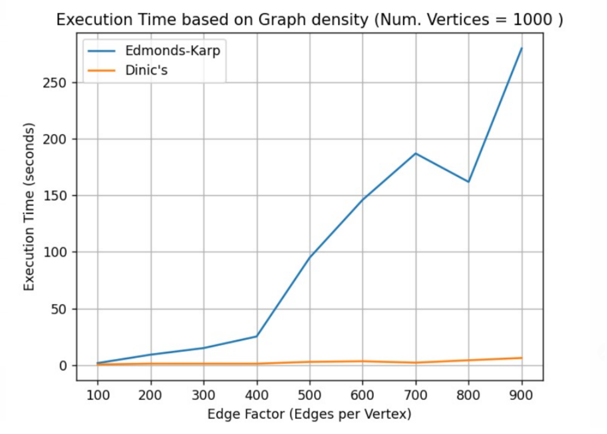
Dinic's algorithm is a more efficient approach to finding the maximum flow, with a time complexity of $O(EV^2)$. This complexity stems from the layered graph construction and the use of blocking flows to increase flow incrementally. Dinic's algorithm ensures faster convergence compared to Ford-Fulkerson and Edmonds-Karp because its outer loop runs only for $O(V)$ times and in every iteration it takes $O(VE)$ in sending multiple more flows until a blocking flow is reached. Therefore, its complexity sums up to $O(EV^2)$ and will perform way better in dense graphs.

5.2 Empirical Analysis

5.2.1 ***Execution Time Analysis.*** The execution time analysis compares the performance of each algorithm in terms of time complexity, measured in seconds against two different variables. i.e increasing number of vertices , and increasing number of edges when vertices are set constant at 1000.





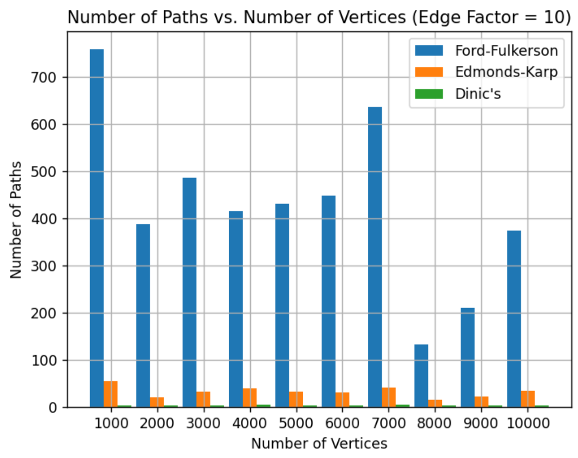


Here are some key observations derived from the above graphical analysis:

- **Ford-Fulkerson Algorithm** The execution time for the Ford-Fulkerson algorithm increases notably as the number of vertices V and edge factor increases. The observed execution times suggest a potential increase in time complexity with larger and denser graph sizes, aligning with the theoretical understanding of the algorithm's behavior. one significant observation here is that even though ford-fulkerson was able to find max flow in sparse graph in reasonable time but for graphs with higher density, it was taking significantly huge time and we were unable to calculate the flow due to limited resources. That's why its execution time against edge factor is plotted separately on less dense graphs to get the idea.
- **Edmonds-Karp Algorithm**
  The Edmonds-Karp algorithm consistently shows faster execution times compared to the Ford-Fulkerson algorithm across varying graph sizes. This efficiency can be attributed to the use of Breadth-First Search (BFS) to find augmenting paths. The execution time increases gradually with larger graphs, demonstrating the algorithm's linear relationship with the number of vertices ($O(VE^2)$). Despite the efficient performance, the algorithm exhibited an increased execution times with highly dense graphs due to the $O(VE^2)$ time complexity. Therefore it is not suitable to use with dense graphs.
- **Dinic's Algorithm**
  Dinic's algorithm consistently outperformed Ford-Fulkerson algorithms in terms of execution time across all graph sizes. This superior performance is attributed to its sophisticated level graph construction and efficient blocking flow identification. Incase of comparison with Edmond's Karp, it's performance was relatively similar when the graph was sparse but as the density of graph increased, it outperformed the algorithm by huge margins. The execution times for Dinic's algorithm remain relatively stable and efficient as compared to other maximum flow algorithms for larger graphs, highlighting its scalability and suitability for real-world applications.

5.2.2 ***Number of Residual Graphs and Layered Networks.*** This section examines the behavior of the algorithms in terms of the number of residual graphs generated and the number of layered networks constructed during execution. The experiment measured the number of residual graphs / layered networks constructed as the number of vertices in the graph increased.

Number of Residual/Layered Graphs made



The result shows that Ford-Fulkerson generates the most number of residual graphs and the number of graphs generated is equivalent to the maxflow from the graph. Whereas in comparison to the Ford-Fulkerson, both Dinic and Edmond-Karp algorithm uses significantly less graphs as they use BFS to update the residual graph.

6 CONCLUSION

In our empirical analysis comparing Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm for maximum flow computation, Dinic's algorithm emerges as the preferred choice due to its scalability, efficient time complexity and robust performance across diverse graph complexities. While Ford-Fulkerson exhibits increasing execution times with larger and denser graphs, especially struggling with highly dense networks, and Edmonds-Karp demonstrates improved efficiency over Ford-Fulkerson in less dense graphs but is limited in denser scenarios, Dinic's algorithm consistently outperforms both alternatives. Our analysis of residual graphs and layered networks confirms Dinic's efficiency, utilizing fewer graph constructions compared to Ford-Fulkerson and Edmonds-Karp. Hence, this study concludes by highlighting Dinic's algorithm as the most versatile and efficient choice for maximum flow computations.

REFERENCES

[1] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. 1997. Computational investigations of maximum flow algorithms. *European Journal of Operational Research* 97, 3 (1997), 509–542. https://doi.org/10.1016/S0377-2217(96)00269-X

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4th ed.). MIT Press.

[3] L. R. Ford and D. R. Fulkerson. 1956. Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8 (1956), 399–404. https://doi.org/10.4153/CJM-1956-045-5

[4] Antonio Sedeño-Noda, M. González-Sierra, and C. González-Martín. 2000. An algorithmic study of the Maximum Flow problem: A comparative statistical analysis. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research* 8 (02 2000), 135–162. https://doi.org/10.1007/BF02564832

[5] Owais Waheed, Mahwish Ahmed, and Abdullah Junejo. 2024. Comparative-Analysis-Of-MaxFlow-Algorithms. https://github.com/Owais-Waheed/Comparative-Analysis-Of-MaxFlow-Algorithms.