UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO HOMMERDING AMORIM

Assignment 2
Ford-Fulkerson: Max Flow Analysis

Porto Alegre
2025

1.1 Task

Implement and compare the following three variants of the Ford-Fulkerson algorithm to compute the maximum flow between a source and sink node in a given flow graph:

- **Randomized DFS:** Uses **randomized Depth-First Search (DFS)** to find augmenting paths. The theoretical time complexity is $O(mC)$, where C is an upper bound on the flow and m is the number of edges in the **residual graph**.
- **Edmonds-Karp:** Uses **Breadth-First Search (BFS)** to find the shortest augmenting path in terms of the number of edges. The theoretical time complexity of this algorithm is $O(nm^2)$, where n is the number of nodes and m is the number of edges in the **residual graph**.
- **Fattest Path:** Selects augmenting paths based on the maximum bottleneck capacity, using a **Dijkstra-like algorithm with a max-priority queue**. The theoretical time complexity is $O(((n+m)\log n) \cdot m \log C)$, assuming a k-ary heap is used for the priority queue. Here, C is an upper bound on the flow, and n and m denote the number of nodes and edges in the **residual graph**, respectively.

The goal of this implementation is to empirically validate the theoretical time complexities by running these algorithms on benchmark instances and analyzing their performance. Specifically, we aim to compare the empirical complexities of the algorithms with their respective worst-case time complexities and understand the relationship between them.

1.2 Solution

The Ford-Fulkerson algorithm was implemented using a customized graph data structure. Each edge maintains a reference to its reverse edge, which is created with zero capacity if it does not already exist, and a criticality counter. During each iteration, the residual graph is updated by decreasing the capacities of forward edges and increasing those of the corresponding reverse edges.

Edges that reach zero residual capacity—termed critical edges—are tracked, and their criticality counters are updated accordingly. To reduce memory usage, the same parent edge vector used to reconstruct the augmenting path is also reused to track visited vertices during the search. About the specific variants:

- **Randomized DFS:** Implemented using a stack and employing a random number generator to shuffle the outgoing edges of each vertex during exploration.
- **Edmonds-Karp:** Implemented using a queue to ensure that the shortest augmenting path (by number of edges) is always selected.
- **Fattest Path:** Implemented using an 8-ary heap. A vector maps vertices to their positions in the heap, enabling constant-time $O(1)$ lookup for efficient updates. The optimal k-value was determined through prior empirical analysis of Dijkstra's algorithm [7].

1.3 Test environment

The results were obtained using a *Lenovo Ideapad S145-15IWL* notebook with an **Intel(R) Core(TM) i7-8565U** processor running at **1.80 GHz** and **20 GB of RAM**. The system was running **Ubuntu 22.04 LTS** with Linux kernel 6.8.0. The code was compiled using **g++ 13.1.0** with optimization flags `-O3`.

1.4 Results

In this section, the results of the experiments are presented, divided into three subsections: datasets, complexity analysis by counting operations, and complexity analysis by measuring execution time.

1.4.1 Data Sets

The first subsection provides details on the 3 families of graphs chosen for this experiment, with each family containing 27 different graphs. The corresponding properties, including parameters for graph construction as well as the number of vertices and edges, are displayed in the following tables. All graphs were generated with a maximum capacity of 1000. These datasets serve as the foundation for the complexity analysis and performance evaluation in the following sections. For a more detailed description of each graph family, please refer to the README file in the repository [8].

1.4.1.1 Meshes

The meshes were generated by incrementally increasing the number of rows r and columns c, which in turn increased the total number of vertices $n = rc$. For each value of n, three types of graphs were created to explore different structural layouts:

1. Square Meshes, where $r = c$.
2. Vertical Meshes, where $r = 4c$.
3. Horizontal Meshes, where $4r = c$.

These mesh types were used to explore different graph structures for the analysis.

Table 1.1 – Dataset 1: Mesh Graphs

G	n	m	r	c	G	n	m	r	c	G	n	m	r	c
1	18	32	2	8	10	578	1680	48	12	19	4098	12160	128	32
2	18	40	8	2	11	578	1704	24	24	20	4098	12224	64	64
3	18	44	4	4	12	578	1716	12	48	21	4098	12256	32	128
4	66	176	16	4	13	1026	3008	64	16	22	8102	24120	180	45
5	66	184	8	8	14	1026	3040	32	32	23	8102	24210	90	90
6	66	188	4	16	15	1026	3056	16	64	24	8102	24255	45	180
7	256	1472	32	8	16	2118	6256	92	23	25	16386	48896	256	64
8	256	1504	16	16	17	2118	6302	46	46	26	16386	49024	128	128
9	258	1520	8	32	18	2118	6325	23	92	27	16386	49088	64	256

1.4.1.2 Random 2-Leveled Mesh Graphs

To introduce variability in the connectivity while preserving the general mesh structure, a second dataset was created. These graphs follow the same row/column structure as the previous meshes but include randomized connections between two distinct levels. This introduces unpredictability to the topology while keeping similar size and layout constraints, helping to analyze performance under more irregular but still structured conditions.

Table 1.2 – Dataset 2: Randomized Mesh Graphs

G	n	m	r	c	G	n	m	r	c	G	n	m	r	c
1	18	40	8	2	10	578	1680	48	12	19	4098	12160	128	32
2	18	44	4	4	11	578	1704	24	24	20	4098	12224	64	64
3	26	69	3	8	12	578	1716	12	48	21	4098	12256	32	128
4	66	176	16	4	13	1026	3008	64	16	22	8102	24120	180	45
5	66	184	8	8	14	1026	3040	32	32	23	8102	24210	90	90
6	66	188	4	16	15	1026	3056	16	64	24	8102	24255	45	180
7	146	408	24	6	16	2118	6256	92	23	25	16386	48896	256	64
8	146	426	6	24	17	2118	6302	46	46	26	16386	49024	128	128
9	258	752	16	16	18	2118	6325	23	92	27	16386	49088	64	256

1.4.1.3 Matching Graphs

To construct this dataset of matching graphs, the parameter n' — representing the number of nodes on one side of the bipartite graph — was incrementally increased. For each value of n', the total number of vertices $n = 2n' + 2$ was determined, including the source and sink nodes. Then, for each fixed n', the degree d of the nodes was varied among 2, 16, and 32 (excluding the smallest graph with $n = 18$, where d took values 2, 4, and 8 due to its smaller size). This variation directly affects the number of edges $m = n'(d+2)$. This approach ensures that for each graph size, multiple instances with different densities are generated, allowing for meaningful comparisons of algorithm performance under varied conditions.

Table 1.3 – Dataset 3: Matching Graphs

G	n	m	d	G	n	m	d	G	n	m	d
1	18	32	2	10	578	1152	2	19	4098	8192	2
2	18	48	4	11	578	5184	16	20	4098	36864	16
3	18	80	8	12	578	9792	32	21	4098	69632	32
4	66	128	2	13	1026	2048	2	22	8102	16200	2
5	66	576	16	14	1026	9216	16	23	8102	72900	16
6	66	1088	32	15	1026	17408	32	24	8102	137700	32
7	258	512	2	16	2118	4232	2	25	16386	32768	2
8	258	2304	16	17	2118	19044	16	26	16386	147456	16
9	258	4352	32	18	2118	35972	32	27	16386	278528	32

1.4.2 Complexity by Counting Operations

This section presents the results obtained by counting the number of operations at both the algorithm and path-finding levels.

1.4.2.1 Algorithm Level

To evaluate the relative iteration cost across algorithms, the following metric is defined: $r = \frac{i}{I}$, where i is the number of iterations executed by the algorithm on a given instance, and I is an upper bound on the number of iterations required by that algorithm in the worst case. To estimate the upper bound C on the maximum flow, the algorithm computes the minimum between the sum of the capacities of the edges leaving the source and the sum of the capacities of the edges entering the sink. The expressions used for I

depend on the algorithm:

- **Randomized DFS**: $I = C$, assuming integral capacities bounded by C.
- **Edmonds-Karp**: $I = \frac{nm}{2}$, where n is the number of vertices and m is the number of edges in the residual graph.
- **Fattest Path**: $I = m \log C$, where C is the upper bound on the maximum flow value and m is the number of edges in the residual graph.

For each input instance, the actual number of iterations i was recorded. Then, the ratio r was computed, and the average $\bar{r}$ was calculated over each dataset-algorithm pair. These results are summarized in Table 1.4.

Table 1.4 – Average r (in %) for each algorithm-dataset pair

Algorithm	matching	mesh	rand-m
Randomized DFS	1.47%	17.19%	15.58%
Edmonds-Karp	0.26%	0.42%	0.38%
Fattest Path	0.45%	0.35%	0.30%

1.4.2.1.1 Edmonds-Karp: Critical Arc Behavior

To understand the structural implications of the Edmonds-Karp algorithm, the behavior of critical arcs across the different datasets is analyzed. Two specific metrics are considered: the fraction of critical arcs among all arcs, and the average criticality per critical arc.

Let C be the set of critical arcs after Edmonds-Karp computes the maximum flow, and let m be the total number of arcs in the graph. The metrics are defined as follows:

- **Critical Arc Fraction**:
$$C = \frac{1}{m} \sum_{a \in A} \mathbf{1}[C_a > 0]$$
This metric indicates the fraction of arcs that were critical at least once during the algorithm execution.

- **Average Criticality per Critical Arc**:
$$\bar{r} = \frac{1}{Cm} \sum_{a \in A} r_a$$
where $r_a = \frac{C_a}{n/2}$ is the criticality ratio for arc a, C is the number of critical arcs, and n is the total number of vertices. This measures how many times a critical arc was critical on average during the execution of the algorithm.

For each dataset, for each graph instance triple with a fixed (or closely the same) value of n (the number of vertices), the average values of C and $\bar{r}$ were computed. These average values were then used to obtain the corresponding plot results. Additionally, the density of the graphs, when averaged over the triples, remains relatively constant, with variations from density being negligible. This suggests that the observed differences in the plot results are primarily due to the scaling effects of the graph size and structure, rather than density variations.

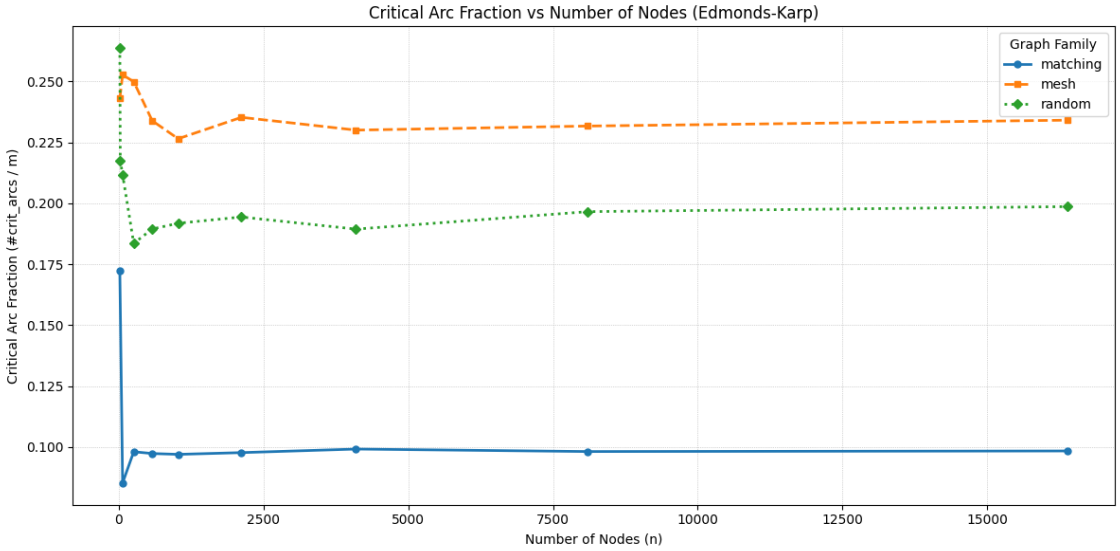

Figure 1.1 – Average.

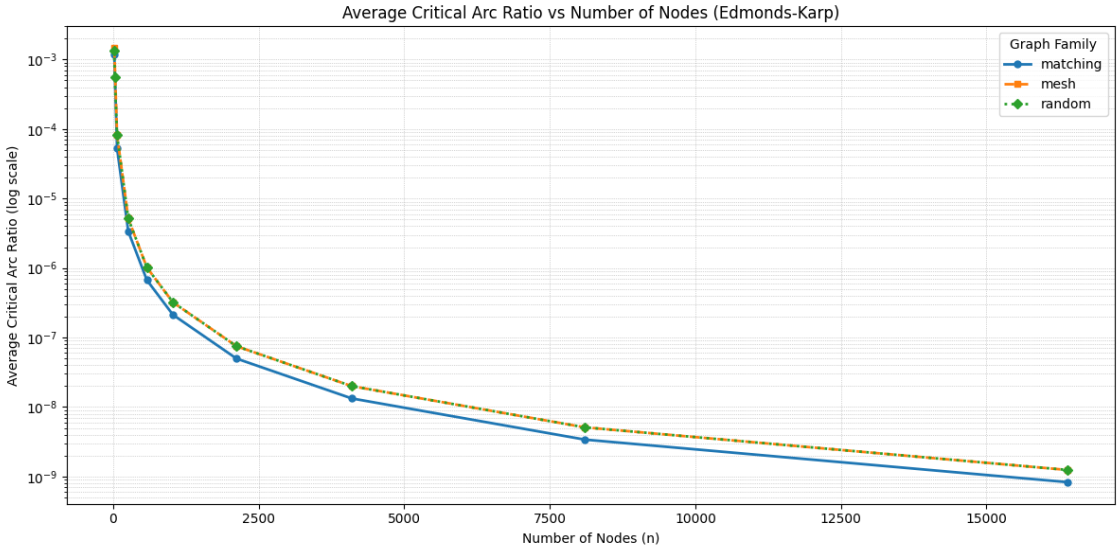

Figure 1.2 – Average criticality per critical arc for each dataset.

1.4.2.2 Path-finding Level

This section analyzes the complexity of the algorithms at the level of path-finding operations. To analyze the search effort per iteration for the Randomized DFS and Edmonds-Karp methods, the fraction of visited vertices $s_i = \frac{n'_i}{n}$ and arcs $t_i = \frac{m'_i}{m}$ are stored during each iteration, where n'_i and m'_i represent the number of visited vertices and arcs in iteration i, respectively. We then compute the averages of these fractions over all iterations (k):

$$\bar{s} = \frac{1}{k}\sum_{i=1}^{k} s_i \quad \text{and} \quad \bar{t} = \frac{1}{k}\sum_{i=1}^{k} t_i$$

These averages, $\bar{s}$ and $\bar{t}$, indicate the typical fraction of the graph explored per iteration for both of the algorithms.

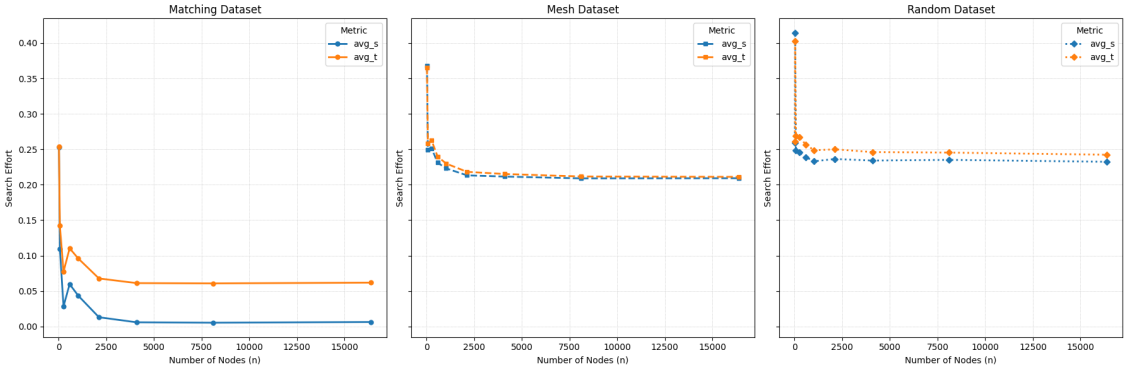

Figure 1.3 – Average fraction of visited vertices ($\bar{s}$) and arcs ($\bar{t}$) for Randomized DFS.

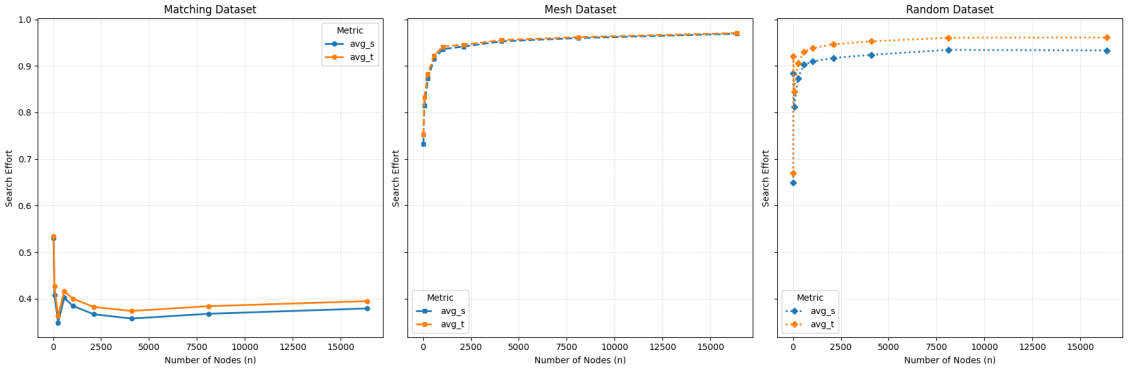

Figure 1.4 – Average fraction of visited vertices ($\bar{s}$) and arcs ($\bar{t}$) for Edmonds-Karp.

For the Fattest Path algorithm, the number of `insert`, `deletemin`, and `update` operations in the priority queue is counted. These counts are used to compute the metrics:

$$i = \frac{\#\text{inserts}}{n}, \quad d = \frac{\#\text{deletemins}}{n}, \quad u = \frac{\#\text{updates}}{m}$$

where:

- $\#$ denotes the total count of the respective operation,
- n is the number of vertices of the graph,
- m is the number of edges of the graph.

The averages $\bar{i}$, $\bar{d}$, and $\bar{u}$ were calculated for each graph.

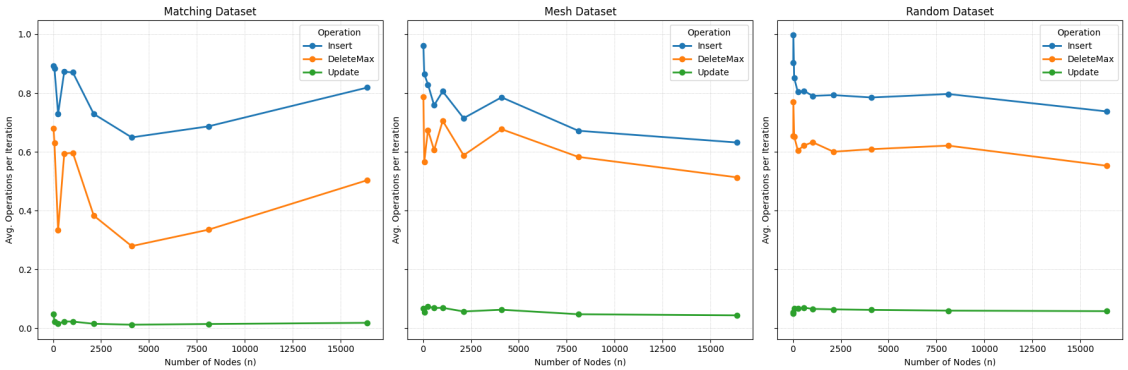

Figure 1.5 – $\bar{i}$, $\bar{d}$ and $\bar{u}$ for incremental graphs using Fattest Path.

1.4.3 Complexity by Measuring Time

This section presents the results obtained by computing the average runtime of each Ford-Fulkerson variant algorithm across the 27 graphs in each of the 3 datasets, using 10 repetitions for each graph. The time plots display the normalized time $\frac{T}{nm(n+m)}$, where T represents the execution time of the algorithm in microseconds. It is important to note that the reported execution time **does not** include the time required to read, parse, and store the graph into memory; it solely reflects the time spent executing the algorithm.

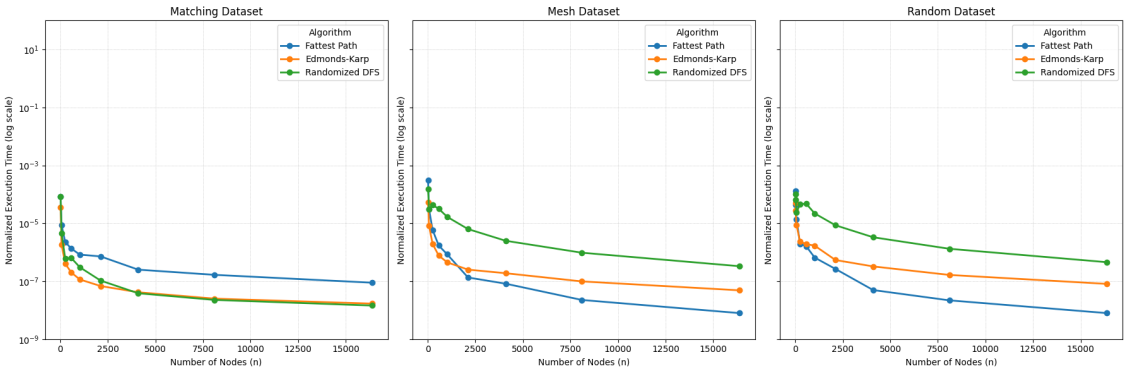

Figure 1.6 – Normalized exec. time $\frac{T}{nm(n+m)}$ for incremental graphs on each dataset.

Figure 1.6 enables comparison of the computational complexity of each variant across differently structured graphs, using a normalized metric that accounts for input size. However, normalized time may obscure practical performance differences. To address this, Table 1.5 reports the average execution time in seconds over the three largest graphs of each dataset. This highlights the real-world cost of executing each algorithm on large inputs.

Table 1.5 – Execution time (s) for largest 3 graphs of each dataset.

Algorithm	matching	mesh	random
Randomized DFS	18.3358	60.7708	83.5501
Edmonds-Karp	9.8362	8.9239	14.8891
Fattest Path	37.1445	1.4574	1.4570

1.5 Conclusion

Analyzing the results, it is possible to formulate some hypotheses and conjectures:

- **Hypothesis 1:** As observed in Table 1.4, the Randomized DFS algorithm explores a larger portion of the residual graph per iteration compared to Edmonds-Karp and Fattest Path. This is especially noticeable in the mesh and random-mesh datasets, where $\bar{r}$ exceeds 15%. In contrast, Edmonds-Karp and Fattest Path maintain consistently low $\bar{r}$ values (below 0.5%), indicating more targeted and efficient path discovery. This difference is likely due to the randomized DFS strategy, which introduces variability and may lead to broader graph traversal before finding augmenting paths, particularly in denser or irregularly structured graphs. Nevertheless, all algorithms remain well below their worst-case iteration counts.

- **Hypothesis 2:** As observed in Figures 1.1 and 1.2, the behavior of critical arcs reveals a dependence on graph structure. For Edmonds-Karp, the fraction of critical arcs stabilizes at approximately 23% for mesh graphs, 20% for random mesh graphs, and 10% for matching graphs. The lower saturation in matching graphs is likely due to their bipartite nature and shorter augmenting paths. Furthermore, the average criticality per arc decreases logarithmically with graph size, reaching values as low as 0.0000001%, indicating that in large graphs, the impact of any single critical arc is minimal due to flow being distributed over a greater number of paths.

- **Hypothesis 3:** As observed in Figures 1.3 and 1.4, the number of visited arcs and vertices increases logarithmically with graph size, but varies depending on algorithm and graph type. Matching graphs consistently result in fewer visited elements due to their bipartite nature and shorter source-to-sink paths. In contrast, BFS—especially in mesh topologies—tends to visit a significantly larger fraction of the graph (up to 90%) due to its layer-by-layer exploration, making it more exhaustive than Randomized DFS or Fattest Path.

- **Hypothesis 4:** As observed in Figure 1.5, the Fattest Path strategy maintains a relatively stable ratio of inserts, deletemaxes, and updates compared to their respective worst-case bounds. Notably, the number of deletions is lower than the number of insertions, which can be attributed to the algorithm terminating early upon reaching the sink. On average, the ratio of insert operations reaches approximately 75% of the worst-case, while deletions stabilize around 50%. Additionally, the ratio of updates remains significantly below its theoretical upper bound—typically under 0.1%—indicating that in practice, update operations occur far less frequently than expected.

- **Hypothesis 5:** As observed in Figure 1.6, the normalized execution time of all algorithms across the different datasets follows a logarithmic trend as the graph size increases. This suggests that, in practice, the algorithms perform significantly better than their theoretical worst-case time complexities. For mesh topologies, the Fattest Path strategy demonstrates the best scalability, maintaining lower execution times as the number of vertices grows. In contrast, for matching graphs, Randomized DFS and Edmonds-Karp perform similarly well, with comparable execution times. This indicates that the structure of the graph influences the relative efficiency of each approach and that, overall, empirical execution times remain far below worst-case bounds.

- **Hypothesis 6:** As observed in Table 1.5, the practical execution time on the largest graphs reveals notable performance differences not captured by normalized metrics. The Fattest Path strategy is the fastest on mesh and random datasets, completing execution in under 1.5 seconds on average. This aligns with its favorable scaling properties observed in Figure 1.6. However, on matching graphs, Edmonds-Karp outperforms the others, with a significantly lower average time of 9.8 seconds compared to 18.3 for Randomized DFS and 37.1 for Fattest Path. This can be explained by the nature of the matching topology: Edmonds-Karp efficiently explores short-hop augmenting paths typical of bipartite graphs, whereas the Fattest Path algorithm, due to its greedy strategy, may follow high-capacity arcs that lead to zig-zag paths, delaying its progress toward the sink.

The experimental results indicate that all evaluated algorithms perform substantially better in practice than their theoretical worst-case time complexities suggest. Across various graph families, the observed execution times and internal operations exhibit logarithmic growth trends, demonstrating efficient behavior even on large inputs.

Among the approaches, the Fattest Path strategy consistently achieves the best overall performance on mesh and random graph topologies, benefiting from its capacity-based prioritization mechanism that reduces the number of iterations. In contrast, the Edmonds-Karp algorithm performs more effectively on bipartite graphs, such as those in the matching dataset, by exploiting the short and direct nature of augmenting paths inherent to the structure.

The Randomized DFS variant generally underperforms in comparison. Its lack of deterministic guidance leads to broader and less efficient exploration of the residual graph, resulting in longer execution times and higher operational costs. While it remains functionally correct, this strategy is not recommended for practical use in scenarios where efficiency is critical.

On a final note, the general recommendation is to adopt the Fattest Path strategy as the default approach due to its overall strong empirical performance. However, in cases where the underlying topology enables efficient breadth-first traversal—such as in bipartite graphs—the Edmonds-Karp algorithm may be preferable.

REFERÊNCIAS

Robert Sedgewick, *Algorithms for the Masses*, ANALCO'11, San Francisco, January 2011. Note: Modi Memorial Lecture, Drexel University, April 2012.

David S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, Proceedings of the 5th and 6th DIMACS Implementation Challenges, 2002.

Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th edition, Addison-Wesley, 2011.

W3Schools, *Ford-Fulkerson Algorithm - Maximum Flow*. Available at: W3Schools Ford-Fulkerson Algorithm. Accessed April 2025.

Moliya, *Ford-Fulkerson Algorithm*. Available at: https://www.slideshare.net/MrMoliya/ford-fulkerson-algorithm-251984779. Accessed April 2025.

Trevisan, L., *CS 261 Lecture 10: The Fattest Path*. Available at: https://lucatrevisan.wordpress.com/2011/02/04/cs-261-lecture-10-the-fattest-path/. Accessed April 2025.

Diego Hommer, *Dijkstra k-ary Heap*. GitHub repository. Available at: https://github.com/diegohommer/dijkstra-k-ary-heap. Accessed April 2025.

Marcus Ritt, *Fluxo Máximo*. GitHub repository. Available at: https://github.com/mrpritt/Fluxo_Maximo. Accessed April 2025.