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

GABRIEL KENJI YATSUDA IKUTA

Assignment 2 - Max Flow

Porto Alegre
2025

1.1 Task

Implement the Ford-Fulkerson family of algorithms for computing the maximum s-t flow in a given capacitated directed graph. The implementations of all the algorithms and the graphs used in this study can be found on my GitHub repository: <https://github.com/Kenjikuta-br/max_flow>.

- The Ford-Fulkerson method using random depth-first search (DFS) for path selection
- The Edmonds-Karp variant using breadth-first search (BFS) for shortest augmenting paths
- A "fattest path"variant that selects augmenting paths with maximum bottleneck capacity
- The capacity scaling variant that processes edges in decreasing order of residual capacity using delta

The analysis will focus on measuring:

- The number of iterations compared to theoretical upper bounds
- The actual number of vertices and arcs processed during path searches
- The empirical running time relative to the predicted complexity

1.2 Solution

1.2.1 Implementation Overview

The Ford-Fulkerson family algorithms were implemented in `C++` with four distinct path-selection strategies. The residual graph is represented using adjacency lists. Key algorithmic variants include:

- **Edmonds-Karp (BFS)**: Shortest augmenting paths via breadth-first search
- **Randomized DFS**: Depth-first search with neighbor shuffling for path diversity
- **Fattest Path**: Maximum bottleneck selection using priority queues
- **Capacity Scaling**: Δ-scaling phases with thresholded DFS

1.2.2 Algorithm-Specific Details

For **Edmonds-Karp**, the BFS implementation uses a queue with early termination when reaching the sink. Each iteration tracks visited nodes/arcs through a token-based system, avoiding array reinitialization. The adjacency list structure enables O(1) residual capacity checks through direct edge references.

The **randomized DFS** variant employs a Mersenne Twister RNG engine to shuffle neighbor exploration order. A stack-based traversal with pre-allocated permutation vectors reduces memory overhead. Path reconstruction occurs immediately upon sink discovery, minimizing redundant operations.

In the **fattest path** implementation, the algorithm employs `std::priority_queue` from C++'s Standard Library, which implements a max-heap using a binary heap structure. Nodes are prioritized by their accumulated bottleneck capacity using a `State` struct containing `(cap, node)` pairs, with the comparison operator configured for descending capacity order.

The **capacity scaling** variant maintains its Δ threshold through the `FFStats` structure, persisting across iterations:

- Initial `max_cap` is computed once during first path search.
- Bitwise operation `1 << (31 - __builtin_clz(max_cap))` determines starting Δ.
- The Δ threshold decreases only when no augmenting paths exist for the current Δ value.

This persistent state management enables the scaling algorithm to maintain context between successive scaling phases while avoiding complete recomputations. The `visitedToken` system - a global array paired with an incrementing counter - prevents array reinitialization between searches through versioned marking, enabling O(1) state checks. By tracking visited nodes, residual arcs, and forward arcs during each Δ-phase, the implementation provides detailed statistics while maintaining the efficiency gains of the scaling approach.

1.2.2.1 Heap Operation Accounting

The fattest path implementation rigorously tracks priority queue behavior through four metrics in the `FFStats` structure:

- `heap_real_inserts`: Counts unique node insertions (1 per node discovery), incremented when a node enters the heap for the first time in the current iteration
- `heap_total_inserts`: Tracks all `push()` operations, including duplicate entries created by capacity updates
- `heap_implicit_updates`: Calculated as `total_inserts−real_inserts`, representing capacity upgrades where nodes are reinserted with better bottleneck values before being processed
- `heap_deleteMins`: Counts all `pop()` operations from the heap

1.2.3 Testing Infrastructure

A custom benchmarking framework was developed using:

- `Makefile` configurations for algorithm variants
- Bash scripts automating test case generation and execution
- Synthetic graphs with controlled vertex/edge counts and capacity distributions

1.2.4 Statistics collection

It occurs through a unified `FFStats` structure, capturing:

- Visited nodes/arcs per iteration
- Heap operations (inserts/updates/deletions)
- Scaling phase effectiveness
- Path reconstruction efficiency

1.2.5 Optimization Strategies

Several algorithm optimizations were implemented:

- Token-based visited marking (avoiding array resets)
- Pre-allocated neighbor order buffers for randomized DFS
- Bitwise delta computation in capacity scaling

- Early termination checks during path searches

1.2.6 Graph Datasets Generation

The experimental datasets were systematically generated using the `new washington.c` generator provided by Professor Ritt (<https://github.com/mrpritt/Fluxo_Maximo>), implementing specialized network structures, which produce four distinct graph families with controlled topological properties.

For each graph type, we generated 30 distinct instances, creating four comprehensive datasets that will be referenced throughout our analysis as: *Square Mesh Dataset*, *Matching Dataset*, *Random 2-Level Dataset*, and *Basic Line Dataset*. Each algorithm variant (BFS (Edmonds-Karp), randomized DFS, fattest path, and capacity scaling) was executed 5 times on every graph instance within these datasets, with performance metrics averaged across these runs to ensure statistical reliability.

- **Square Mesh Graphs** (Type 5): Simulating 2D grid networks where each node connects to its neighbors. Generated with:
  - Side length $s \in [50, 110]$ in steps of 2 (yielding s^2 vertices)
  - Fixed maximum degree $d = 25$
  - Uniform edge capacities $C = 1000$
- **Matching Graphs** (Type 4): Bipartite graphs with perfect matching characteristics, constructed with:
  - Partition sizes $n \in [2500, 7465]$ with $\Delta n = 165$
  - Fixed degree $d = 25$
  - Capacity bound $C = 1000$
- **Random 2-Level Graphs** (Type 3): Layered networks with random inter-level connections featuring:
  - Rows r and columns c both ranging $[50, 110]$ (step 2)
  - Each vertex connects to 3 random vertices in the next level
  - Capacities uniformly distributed up to $C = 1000$
- **Basic Line Graphs** (Type 6): Hybrid grid-random structures with:
  - Lines n and segments m in $[50, 110]$ (step 2)
  - Fixed inter-line degree $d = 25$
  - Maximum capacity $C = 1000$

1.3 Test Environment

1.3.1 Hardware Specifications

The results were obtained on a system with the following hardware configuration:

- **CPU:** AMD Ryzen 5 5600X
  - Base frequency: 3.7GHz
  - Max boost frequency: 4.6GHz
  - Total cache: 35MB
  - Cores: 6
  - Threads: 12

    - Platform: AM4
- **Motherboard:** Asus TUF Gaming X570-Plus
- **RAM:** 32GB DDR4
    - Modules: XPG Spectrix D50 RGB
    - Frequencies: 3000MHz and 3200MHz
- **Storage:** WD Black SN750 500GB NVMe SSD
    - Read speed: 3430MB/s
    - Write speed: 2600MB/s
- **GPU:** Zotac Gaming NVIDIA GeForce RTX 2060
    - Memory: 6GB GDDR6

1.3.2 Software Environment

The experiments were conducted on an **Ubuntu 24.04.2 LTS** operating system (codename *noble*), with the following software components:

- **Python 3.12.3** — using virtual environment (`venv`) for project dependency isolation. Main libraries included:
    - `pandas`
    - `numpy`
    - `statsmodels`
    - `matplotlib`
    - `seaborn`
    - `scikit-learn` (`sklearn.preprocessing.StandardScaler`)
    - `os`, `io.StringIO`
- **GCC 13.3.0** — compiler used for C/C++ codes, installed via Ubuntu 24.04 official packages.

The `venv` virtual environment enabled isolated installation and management of required libraries, ensuring reproducibility and control over the execution environment.

1.4 Results

1.4.1 Experimental Protocol

I conducted comprehensive testing across all four datasets (Square Mesh, Matching, Random 2-Level, and Basic Line) using the following maximum flow algorithms:

- Edmonds-Karp (BFS-based)
- Randomized DFS
- Fattest Path
- Capacity Scaling

For each graph instance in every dataset, we executed all four algorithms five times and averaged the results to minimize noise and ensure measurement stability. The analysis focused on multiple performance metrics:

- **Iteration Counts**:
  - Actual iterations (I) vs theoretical bound ($bound$)
  - Ratio $r = I/bound$ measuring empirical efficiency

  The theoretical bounds (used to calculate r) were computed algorithm-specifically:

$$bound = \begin{cases} C & \text{(DFS)} \\ nm/2 & \text{(Edmonds-Karp)} \\ m \log C & \text{(Fattest Path/Capacity Scaling)} \end{cases}$$

- **Search Behavior**:
  - Visited nodes per iteration ($\bar{s}$)
  - Inspected arcs per iteration ($\bar{t}$)
- **Heap Operations** (Fattest Path only):
  - Real inserts and implicit updates
  - DeleteMin operations
  - Normalized counts against graph size

  I tracked operation counts against expected values:

$$\text{Updates}_{norm} = \frac{\text{Actual Updates}}{(\alpha - 1)n \ln n}, \quad \alpha = \log_n m$$

I evaluated four normalized time metrics to analyze the empirical complexity (not all charts will be included, $\frac{T}{nI}$ and $\frac{T}{mI}$ showed real similarity in the resulting graphs).:

- **Time per arc per iteration**:

$$\frac{T}{mI} = \frac{\text{Total time}}{\text{Residual arcs} \times \text{Iterations}}$$

- **Time per vertex per iteration**:

$$\frac{T}{nI} = \frac{\text{Total time}}{\text{Vertices} \times \text{Iterations}}$$

- **Time normalized by graph size**:

$$\frac{T}{nm} = \frac{\text{Total time}}{\text{Vertices} \times \text{Residual arcs}}$$

- **Operation-normalized time**:

$$\frac{T}{I(\bar{s}n + \bar{t}m)} = \frac{\text{Total time}}{\text{Iterations} \times (\text{Avg. node visits} + \text{Avg. arc visits})}$$

  Where:

- T = Total execution time (seconds)
- n = Number of vertices

- m = Number of residual arcs
- I = Total iterations
- $\bar{s}$ = Average fraction of visited nodes per iteration
- $\bar{t}$ = Average fraction of visited residual arcs per iteration

1.4.2 Algorithm-Level Complexity

The boxplot shown in Figura 1.1 provides a comparison of the iteration counts for different algorithms across all graph types. The parameter r, which is the ratio of the iteration count I to the theoretical bound, is used to evaluate the efficiency of each algorithm. As illustrated in the figure, the BFS algorithm typically demonstrates the lowest values of r, which is consistent with the theoretical analysis of its complexity. The bound for BFS includes both the number of vertices and edges (nm), making it naturally more dependent on the graph size compared to other algorithms. This behavior is expected, given that BFS often requires more iterations to explore the graph comprehensively, especially when the number of edges is large.

The figure highlights the relative performance of the algorithms in terms of their iteration counts, with each box representing the distribution of r values for an aggregation of all graph types.

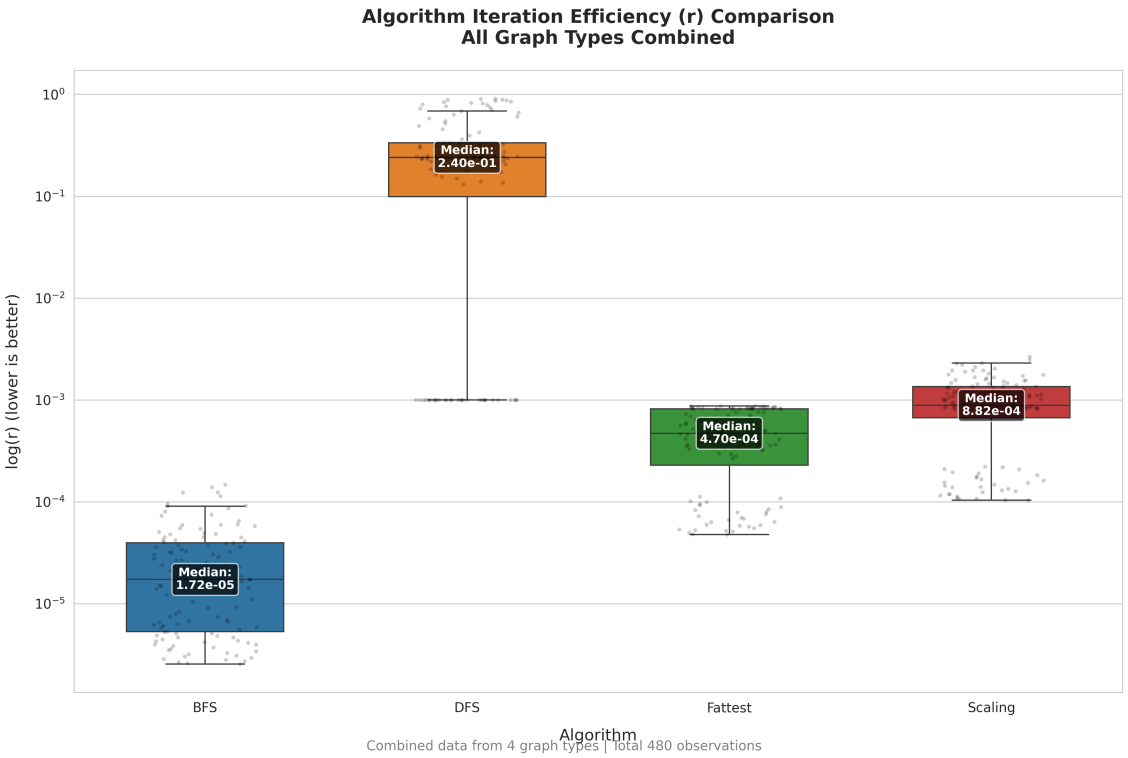


Figura 1.1 – BoxPlot of r for all graphs

For the Basic Line, Random 2-Level, and Square Mesh graph types, the behavior is quite similar. In all cases, BFS exhibits the best performance in terms of the lowest values of r, followed by Fattest Path, Scaling, and DFS. This indicates that these algorithms require fewer iterations, as r is calculated as the ratio of iterations to the upper bound of each operation. Since these three graph types show similar trends, we do not include the corresponding plots for Random 2-Level and Square Mesh, as they closely resemble the plot for the Basic Line graph type.

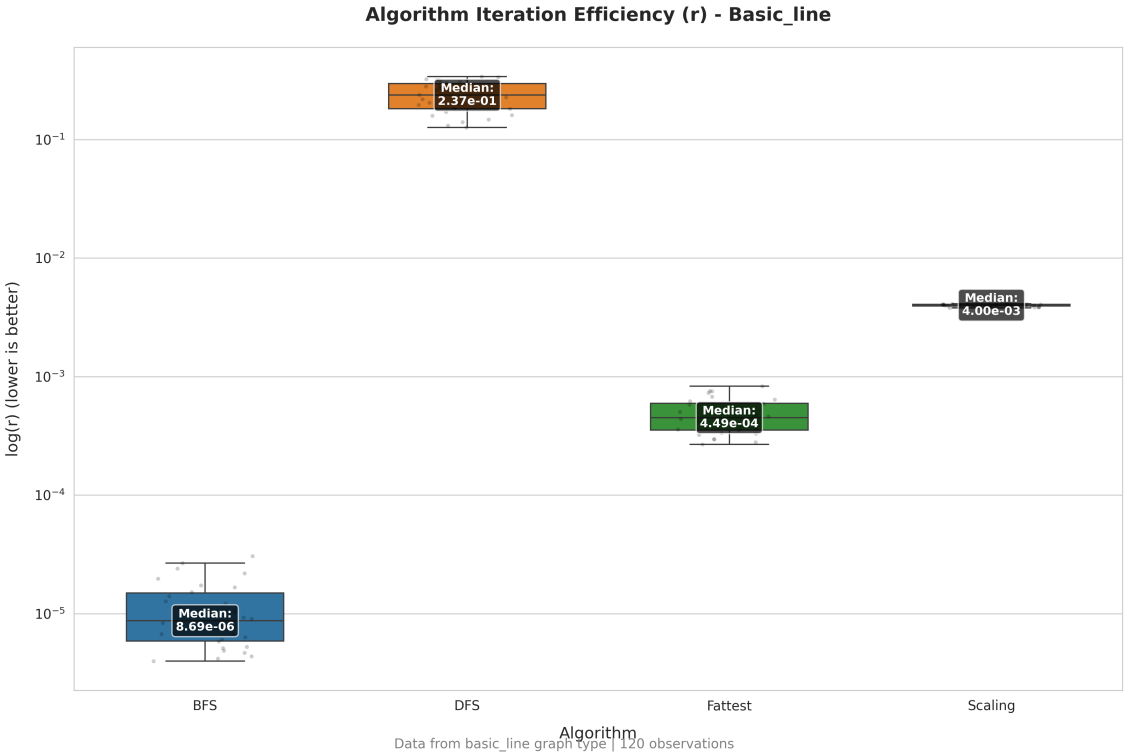


Figura 1.2 – BoxPlot of r for Basic Line Graph

For the Matching graph, an interesting behavior is observed. While BFS remains the most efficient algorithm in terms of r, the Fattest Path, Scaling, and DFS algorithms exhibit a much closer performance, with the order of their r-values remaining the same. However, these three algorithms converge more closely in terms of r for the Matching graph type. This occurs because the Matching graph type has a specific structure that influences the performance of these algorithms. In this case, the sparsity and regularity of the graph may result in a smaller gap in the number of iterations required by Fattest Path, Scaling, and DFS, making their performance more similar.

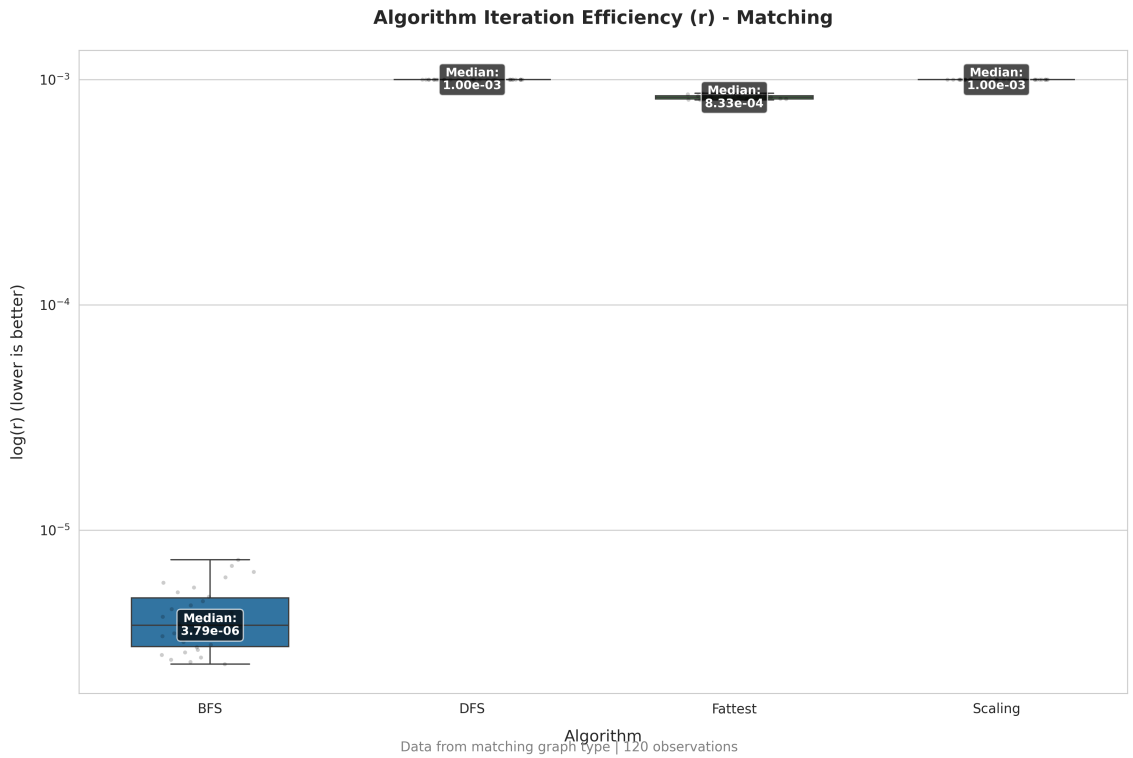


Figura 1.3 – BoxPlot of r for Matching Graph

1.4.3 Critical Edge Analysis (Edmonds-Karp)

In Figura 1.4a(a), we observe that the fraction of critical edges C remains almost constant as the number of vertices increases for most graph types.

Observation 1: For Basic Line, Random 2-Level, and Matching graphs, C shows negligible variation with graph size. *Hypothesis:* These graph families grow in a relatively uniform manner—either in a simple path structure or with regular bipartite matchings—so the proportion of edges that ever become critical stabilizes quickly and does not depend strongly on scale.

Observation 2: In the Square Mesh graph, C increases slightly with the number of vertices. *Hypothesis:* As the grid becomes larger, more alternate routes exist, causing additional edges to become critical in some iterations. This additional connectivity can lead to a gradual rise in the fraction of edges that are ever on a bottleneck path.

Observation 3: The Random 2-Level graph exhibits the highest and most irregular C values—approximately double those of the other types. *Hypothesis:* Random two-level constructions produce highly uneven degree distributions and unpredictable bottleneck locations. This irregularity forces a larger subset of edges to become critical at least once.

Overall, the ranking of C across graph types (from highest to lowest) is: Random 2-Level, Square Mesh, Matching, and Basic Line. Each hypothesis above offers a structural explanation for these trends.

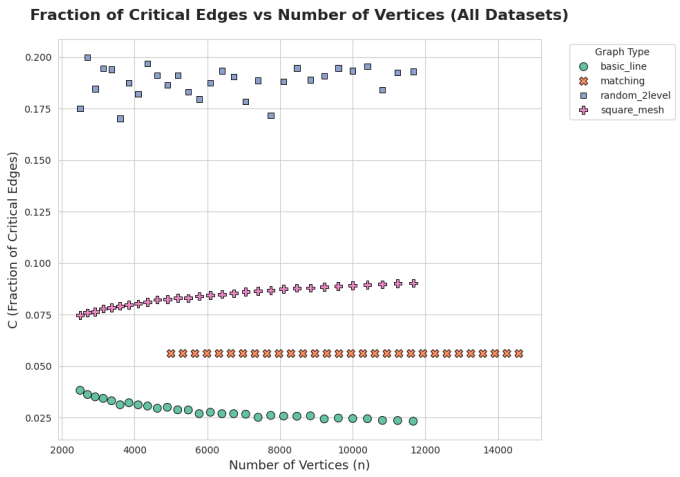
In Figura 1.4b(b), the average criticality ratio $\bar{r}$ exhibits a clear downward trend as the number of vertices increases, across all graph types.

Observation: Across all graph types—Basic Line, Matching, Random 2-Level, and Square Mesh—the average criticality ratio $\bar{r}$ is highest for the smallest graphs and declines steadily as the number of vertices increases.

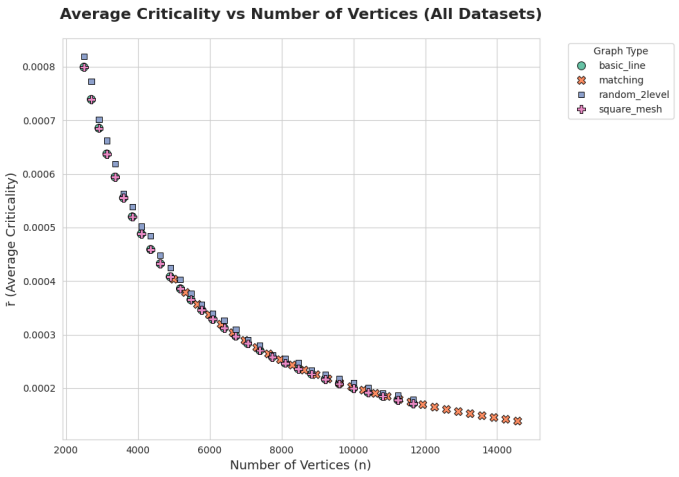
Hypothesis 1: As the graph grows, the number of edges that ever become critical spreads over a larger set of arcs. Each individual edge is less likely to remain on a

bottleneck path repeatedly, driving down the per-edge average.

Hypothesis 2: Larger graphs typically admit more alternate augmenting paths. With more choices available, the algorithm redistributes flow adjustments across different edges, reducing the average frequency with which any single edge is saturated.



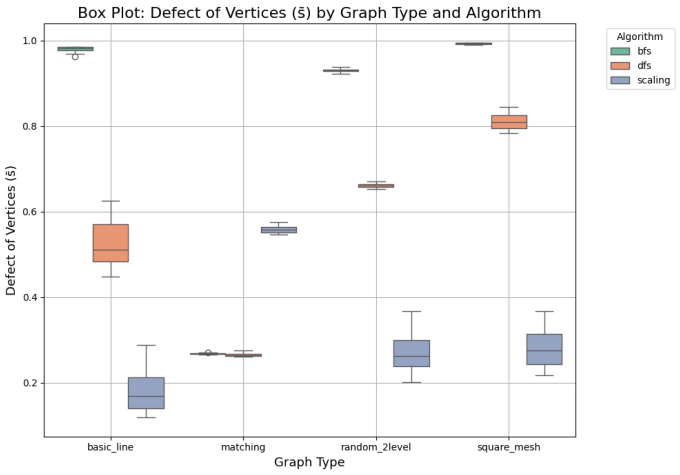
(a) Fraction of Critical Edges vs. Number of Vertices



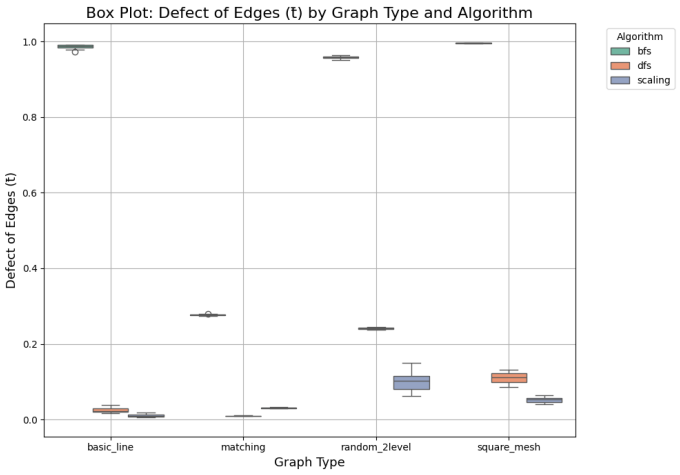
(b) Average Criticality vs. Number of Vertices

Figura 1.4 – Analysis of edge criticality across datasets showing (a) fraction of edges that became critical at least once (C) and (b) average criticality ratio ($\bar{r}$)

1.4.4 Path Search Efficiency



(a) Boxplot $\bar{s}$ by graph and algorithm (All Datasets)



(b) Boxplot $\bar{t}$ by graph and algorithm (All Datasets)

Figura 1.5 – Comparison of average fractions of visited nodes ($\bar{s}$) and edges ($\bar{t}$) across all datasets.

Inspection of Figura 1.5 reveals two distinct patterns:

1. **Basic Line, Random 2-Level, and Square Mesh.** In these three families the ordering of algorithms is consistent for both $\bar{s}$ and $\bar{t}$:

$$\text{BFS} > \text{DFS} > \text{Scaling},$$

where "$>$" denotes a larger average fraction. BFS visits the greatest proportion of nodes and edges per search, DFS is intermediate, and Scaling the smallest.

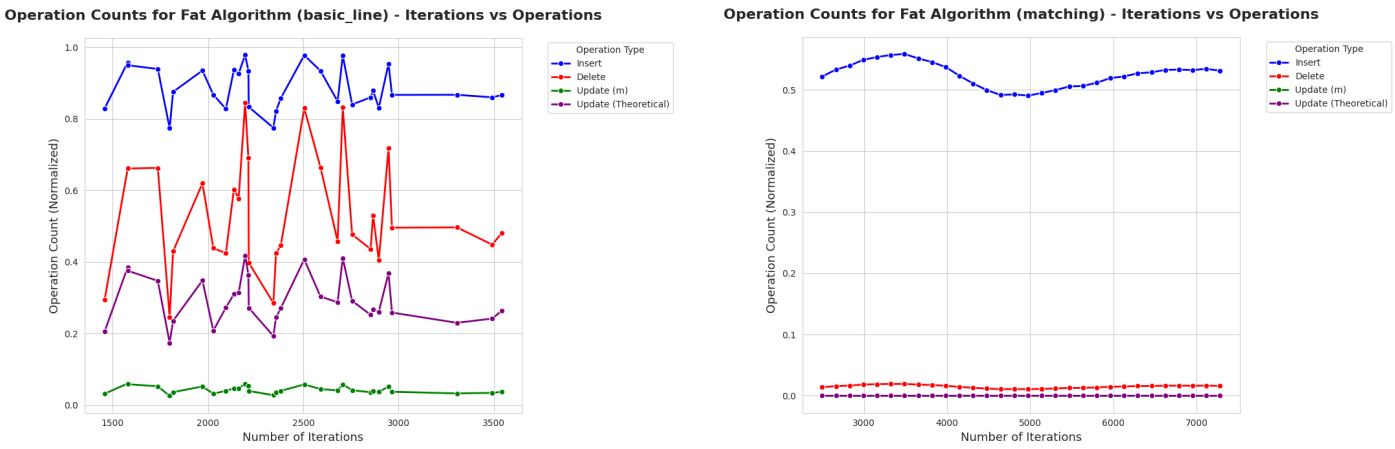
2. **Matching.** Here the behavior diverges:

- For $\bar{s}$: BFS and DFS show similar values, while Scaling actually becomes the largest.
- For $\bar{t}$: BFS remains the highest, Scaling is lower, and DFS is the lowest.

Hypotheses:

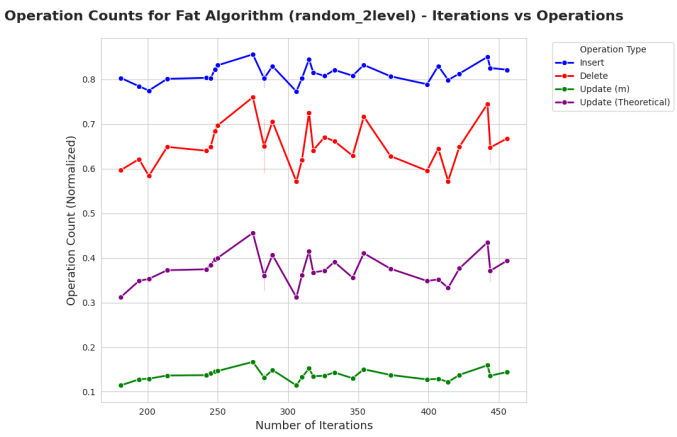
- *Uniform structures (Basic Line, Random 2-Level, Square Mesh)* yield predictable search costs: BFS's breadth exploration touches most vertices and edges, DFS's depth-first visits fewer, and Scaling—by limiting augmenting paths via capacity thresholds—visits the fewest.
- *Matching graphs* consist of sparse, regular bipartite connections. In this context:
  - Both BFS and DFS can quickly reach the opposite partition with similar traversal footprint, explaining their convergence in $\bar{s}$.
  - Scaling's priority on high-capacity paths may force it to traverse more edges to find augmenting paths, increasing its $\bar{s}$.
  - BFS's exhaustive layer-by-layer search still visits many edges ($\bar{t}$ highest), whereas DFS's narrower path exploration visits fewer, placing DFS lowest.

1.4.5 Heap Operations (Fattest Path)

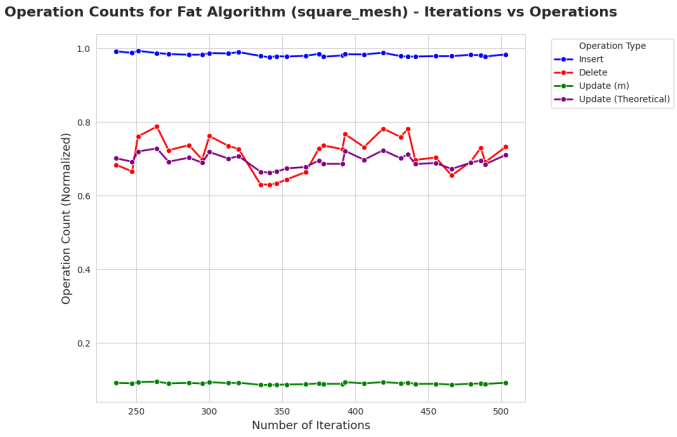


(a) Operation Counts vs Iterations (Basic Line Dataset) (b) Operation Counts vs Iterations (Matching Dataset)

Figura 1.6 – Comparison of heap operation counts (inserts, updates, deletions) across iterations for Fattest Path algorithm on (a) Basic Line and (b) Matching graph datasets



(a) Operation Counts vs Iterations (Random 2-Level Dataset)



(b) Operation Counts vs Iterations (Square Mesh Dataset)

Figura 1.7 – Comparison of heap operation counts across iterations for Fattest Path algorithm on (a) Random 2-Level and (b) Square Mesh graph datasets. Note: Figures are intentionally wider to show detail.

Figures Figura 1.6a, Figura 1.7a, and Figura 1.7b reveal a consistent pattern for the Basic Line, Random 2-Level, and Square Mesh datasets:

- **Order of operations:** Inserts have the highest frequency, followed by deletions, then update_m, and finally update_theoretical.
- **Square Mesh detail:** Deletion and theoretical updates are nearly identical, reflecting the mesh's regular connectivity.
- **Basic Line and Random 2-Level:** Deletions exceed theoretical updates, suggesting more frequent heap removals.

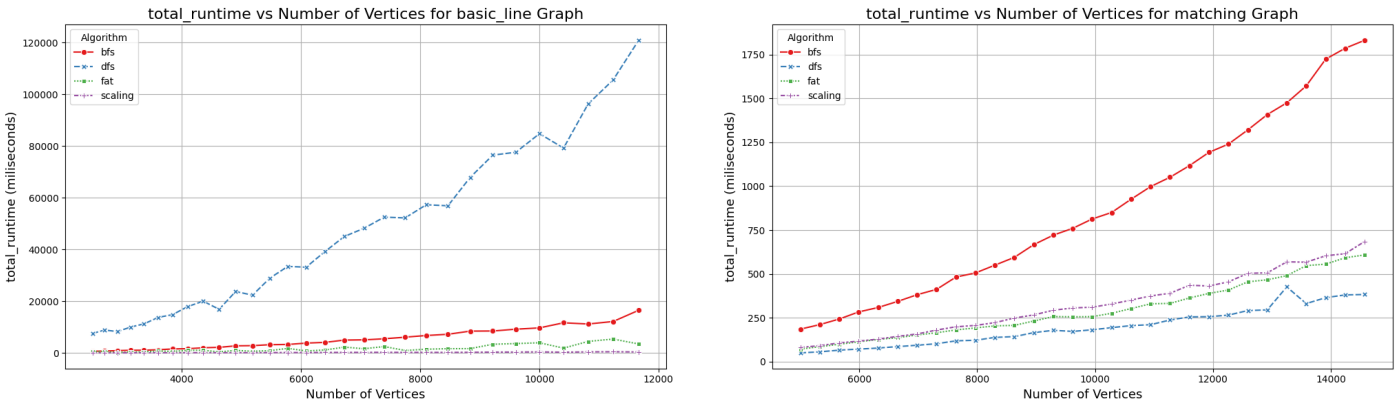
In contrast, Figure Figura 1.6b (Matching dataset) exhibits a markedly different profile:

- **Inserts dominate:** The insertion count is overwhelmingly larger than all other operations.
- **Few deletions:** Deletion operations are relatively infrequent.
- **Virtually no updates:** Both update_m and update_theoretical appear near zero, indicating very few decrease-key operations.

Hypotheses for observed behavior:

- *Uniform vs. irregular topology:* In uniform structures (Square Mesh), heap operations follow predictable trends. The mesh's regularity equalizes deletions and theoretical updates, while in sparser/path graphs (Basic Line, Random 2-Level) the deletes are more frequent then the updates because once a node reaches the heap with a "good enough"value, it can often be finalized (deleted) without needing further improvement (update).
- *Matching graph sparsity:* The bipartite Matching graph ($d = 25$, $n \in [2500, 7465]$) is extremely sparse (edge density $\frac{25}{n}\% \approx 0.0033\%$ to 0.01%) and consequently has very few alternative augmenting paths. Fattest Path therefore inserts many candidate edges but performs almost no deletions or updates, since few edges ever become suboptimal.

1.4.6 Time Complexity



(a) Total Runtime vs. Vertices (Basic Line) (b) Total Runtime vs. Vertices (Matching)

Figura 1.8 – Comparison of total runtime scaling with graph size for (a) Basic Line and (b) Matching datasets. Both axes use logarithmic scaling to show polynomial relationships.

1.4.6.1 Total Runtime Comparison Across Datasets

Based on the runtime plots, we can observe distinct behaviors between the datasets:

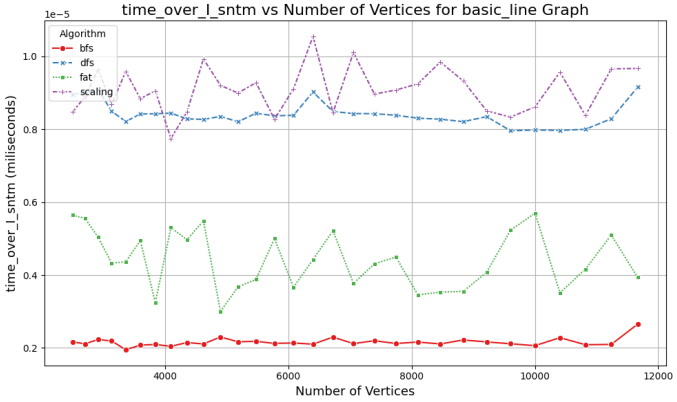
- **Basic Line, Random 2-Level, and Square Mesh:** In these datasets, the worst performing strategy in terms of total runtime is *DFS*, by a significant margin. Following that comes *BFS*. The two best-performing algorithms are *Fattest Path* and *Scaling*, with *Scaling* achieving the best runtime overall among them.
- **Matching Dataset:** The behavior here is notably different. In the Matching graphs, the worst total runtime is observed for *BFS*. Then, *Scaling* and *Fattest Path* perform similarly, achieving better runtimes than BFS. Surprisingly, *DFS* achieves the best performance among all strategies in this dataset.

Hypotheses:

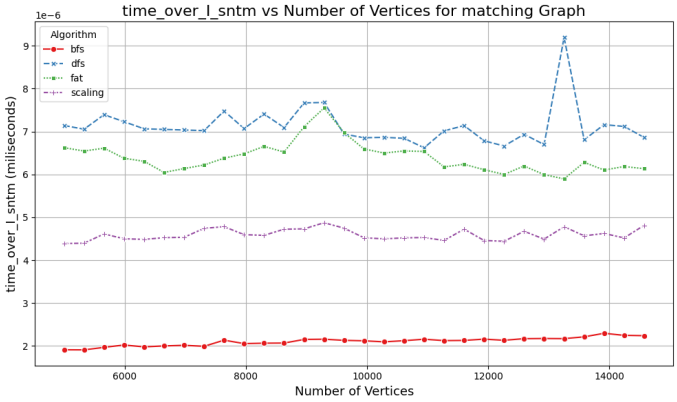
- In Basic Line, Random 2-Level, and Square Mesh graphs, the structured connectivity may favor augmenting path searches that quickly find good paths without excessive exploration, which benefits the *Scaling* and *Fattest Path* methods. *DFS* tends to get trapped exploring deep but suboptimal paths, leading to inefficient runtimes.
- In the Matching graphs, the structure is highly sparse and specialized (perfect matching-like patterns), where deep paths leading directly to augmenting paths are more effective. This naturally favors *DFS*, which aggressively explores paths to their end before backtracking. Conversely, *BFS* suffers because level-by-level exploration becomes inefficient when augmenting paths are long and sparse.

1.4.6.2 Normalized Runtime Comparison Across Graphs

The following scatter plots present the normalized runtime, expressed as $\frac{T}{I(\bar{s}n+\bar{t}m)}$, across different graph structures. While there are some similarities, each graph displays distinct behavior patterns in terms of runtime scaling.

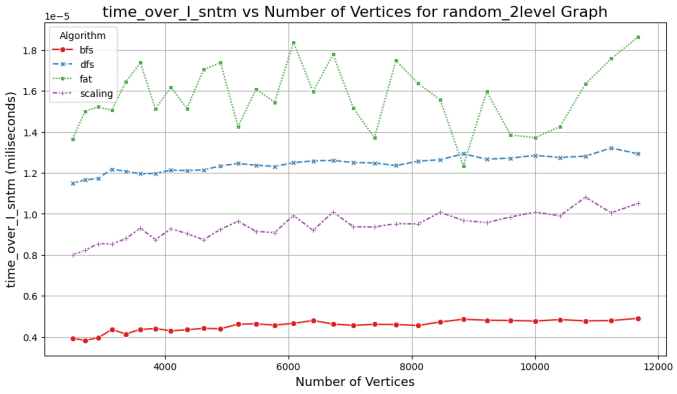


(a) Scatter $\frac{T}{I(\bar{s}n+\bar{t}m)}$ vs. Vertices (Basic Line)

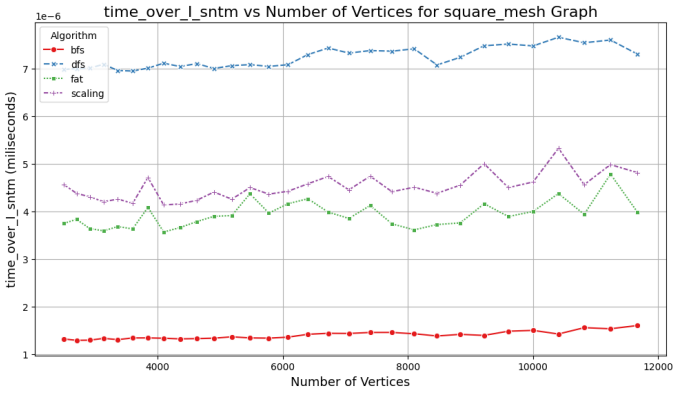


(b) Scatter $\frac{T}{I(\bar{s}n+\bar{t}m)}$ vs. Vertices (Matching)

Figura 1.9 – Comparison of normalized runtime $\frac{T}{I(\bar{s}n+\bar{t}m)}$ scaling with graph size for (a) Basic Line and (b) Matching datasets.



(a) Scatter $\frac{T}{I(\bar{s}n+\bar{t}m)}$ vs. Vertices (Random 2-Level)



(b) Scatter $\frac{T}{I(\bar{s}n+\bar{t}m)}$ vs. Vertices (Square Mesh)

Figura 1.10 – Comparison of normalized runtime $\frac{T}{I(\bar{s}n+\bar{t}m)}$ scaling with graph size for (a) Random 2-Level and (b) Square Mesh datasets.

- **Basic Line Graph, Matching Graph, Random 2-Level Graph, and Square Mesh Graph:** In all datasets, *BFS* shows the lowest normalized runtime, meaning it visits more nodes and edges compared to the other algorithms. However, this does not imply that *BFS* is the fastest or most efficient in raw time. The lower normalized time for *BFS* results from its exploration of a greater portion of the graph's structure, whereas the other algorithms tend to focus on more specific exploration paths. The normalized time reflects the algorithm's ability to cover more of the graph's nodes and edges, rather than indicating it is the fastest in terms of raw execution time.

Note on Time Normalization: All time measurements have been normalized by iteration complexity, specifically the formula:

$$T/[I \cdot (\bar{s} \cdot n + \bar{t} \cdot m)]$$

where T is the total time, I is the number of iterations, $\bar{s}$ is the average number of nodes visited per iteration, n is the total number of nodes, $\bar{t}$ is the average number of arcs visited per iteration, and m is the total number of arcs. This normalization ensures that the observed performance is adjusted based on the node and arc exploration (operation-based complexity), and the lowest normalized time indicates the best performance in terms of

efficiency in exploring the graph's structure, not necessarily the raw time taken by the algorithm.

1.4.7 Conclusion

Key Findings:

- *BFS* consistently demonstrates the best performance in terms of iteration counts, particularly for basic graph types such as Basic Line, Random 2-Level, and Square Mesh.
- The *Scaling* and *Fattest Path* algorithms generally exhibit better runtime efficiency, especially in structured graphs like Basic Line and Square Mesh, where their targeted approach in finding augmenting paths proves effective.
- For the Matching graph type, *DFS* outperforms other algorithms in total runtime, highlighting the advantages of its depth-first exploration in sparse, highly regular graphs.
- The analysis of critical edge behavior reveals that graph structure plays a significant role in determining the proportion of edges that become critical, with Random 2-Level graphs exhibiting the highest criticality due to their irregular degree distributions.
- Path search efficiency results suggest that BFS tends to explore more nodes and edges compared to other algorithms, particularly in uniform graph structures, while *Scaling* visits the least, consistent with its more selective path exploration.
- Heap operation analysis for the *Fattest Path* algorithm reveals a clear pattern in heap activity, with insert operations being the most frequent, particularly in sparse graphs like Matching, where few edges require updates or deletions.

  Implications: The findings suggest that there is no one-size-fits-all solution for maximum flow algorithms. Instead, the choice of algorithm should be tailored to the specific characteristics of the graph in question. For graphs with regular structures and well-connected paths, *Scaling* and *Fattest Path* are optimal due to their efficiency in finding augmenting paths. In contrast, sparse graphs like the Matching graph benefit from depth-first search (DFS), which excels in scenarios with fewer but deeper augmenting paths.

  Future Work: Future studies could explore the scalability of these algorithms in more complex graph types, including those with dynamic edge weights, or investigate the application of parallel or distributed versions of these algorithms to further reduce runtime for large-scale networks. Additionally, a deeper analysis of memory usage and other practical performance metrics would provide a more holistic view of algorithm efficiency.