Team: hmm
Project 12: Maximum Flow & Minimum Cut

Team Members: Siddarth Meda, Aayush Batra, Rahul Kannan, Aditya Sai Shrinivas, Joshua Koilpillai

Github Link: https://github.com/AayushBat/AAD-Project#

Index

Background

This project focuses on the maximum flow problem, in which the goal is to send the maximum possible flow from a source to a sink without exceeding edge capacities. To explore how max-flow algorithms evolved and how their performance differs in practice, we implemented three major approaches: the classic Ford-Fulkerson method, the more efficient Dinic's algorithm, and the fundamentally different Push–Relabel algorithm. Using these implementations, we compared their correctness, time complexities, and practical runtimes, and we extracted the corresponding minimum cuts to validate the max-flow min-cut theorem.

The graph families we generate represent structures that commonly appear in real systems. Sparse graphs resemble transportation or communication networks, dense graphs model highly connected systems, and bipartite graphs reflect matching problems such as scheduling or assignments.

Algorithms

1) Ford-Fulkerson

Idea of the algorithm:
Ford–Fulkerson is the basic method for computing maximum flow. We begin with zero flow and repeatedly look at the residual graph, which tells us where we can still push flow. Our goal is to find any path from the source to the sink in this residual graph. Once we find a path, we look at the smallest available capacity on that path and push that much flow through it. After updating the residual graph, we search again. When there is no path left, our flow is maximum.

Pseudocode (Ford–Fulkerson):

1. Set flow f(u,v) = 0 for all edges.

2. Build the residual graph Gf.

3. While an s-t path P exists in Gf:
   a. Find the minimum residual capacity on P.
   b. Add that amount of flow along edges of P.

c. Update the residual graph.

4. Return the final flow.

Correctness (sketch):
Every augmentation respects capacity constraints, and flow conservation holds at intermediate vertices because we add and remove flow consistently along a path. When no more s-t path exists in the residual graph, the reachable set of vertices naturally forms a cut whose outgoing edges are saturated. The value of our flow equals the capacity of this cut, so the flow must be maximum.

Time complexity:
Ford–Fulkerson runs in O(E * F) when capacities are integers, where F is the maximum flow value.

2) Dinic's Algorithm

Idea of the algorithm:
Dinic's algorithm improves performance by structuring the search into phases. In each phase, we build a level graph using BFS. The level graph arranges vertices by their distance from the source. We then send flow only along edges that respect this level order and continue until we get a blocking flow. This means every s-t path in the level graph has at least one saturated edge. By pushing several augmenting paths in one phase, Dinic's algorithm becomes faster in practice than basic Ford–Fulkerson.

Pseudocode (Dinic):

1. Initialize all flows to zero.

2. While the sink is reachable in the residual graph:
   a. Build the level graph using BFS.
   b. Use DFS to send blocking flow along level-respecting edges.

3. Return the final flow.

Correctness (sketch):
Each DFS push maintains a feasible flow. At the end of each phase, all s-t paths in the level graph are blocked. Eventually BFS will fail to reach the sink, showing that no

residual s-t path exists. By the max-flow min-cut theorem, the resulting flow is maximum.

Time complexity:
Dinic's algorithm runs in O(V^2 * E) in the worst case, but it is usually much faster in practical experiments.

3) Push-Relabel Method

Idea of the algorithm:
Push–Relabel works differently from augmenting path methods. Instead of maintaining a valid flow at every step, we maintain a preflow, which means some vertices may have excess flow. Each vertex is assigned a height value. Using these heights, we repeatedly perform two operations:

1. Push: If a vertex has excess and an outgoing edge has residual capacity and a lower height, we push flow along that edge.

2. Relabel: If no outgoing edge is usable, we increase the height of that vertex so that pushing becomes possible.

We continue processing active vertices until no vertex except the sink has excess. At that point, the preflow becomes a valid maximum flow.

Pseudocode (Push–Relabel):

1. Set h(s) = number of vertices and h(v) = 0 for all other vertices.

2. Saturate all edges going out of the source to create the initial preflow.

3. While there exists an active vertex u (u is not s or t and has positive excess):
   a. If there is an admissible edge (u,v), push flow.
   b. Otherwise, relabel u by increasing its height.

4. Return the resulting flow.

Correctness (sketch):
Push–Relabel always respects capacity limits and maintains correct residual edges. The height rule ensures that excess flow moves toward the sink. When the algorithm stops, all excess except at the sink is gone, so the preflow is now a valid flow. Since no residual s-t path remains, the flow must be maximum.

Time complexity:
The algorithm runs in O(V^2 * E).

Environmental Setup

We have used Python 3.13.7 for the project.

The following libraries were used:

1) Pandas
2) Pickle
3) Time
4) MathplotLib
5) Seaborn

Approach

In this project, we create our dataset by generating several **families of graphs**, each designed to test different behaviors of the max-flow algorithms. Instead of using just one type of random graph, we build a diverse collection of graph structures so that we can study how our algorithms perform on multiple types of inputs. This approach allows us to analyze both typical and extreme cases.

Our dataset generation follows these steps:

1. **Choose a set of input sizes**
   We use the values N = {200, 400, 600, 800, 1000}.
   For each size, we generate graphs for multiple families.
   For random graphs, we also vary the random seed to get multiple samples.

2. **Generate multiple graph families with different structures**
   We create **seven** types of graphs:

   - Sparse random graphs
   - Dense random graphs (5% density)
   - Bipartite graphs

- Even–Tarjan graphs (worst case for Dinic)
- Diamond graphs (worst case for Ford–Fulkerson)
- Star-of-Stars graphs (worst case for Push–Relabel)
- (Layered graphs and chain graphs are supported but not used in this main loop)

3. Each graph family represents a different structural pattern, which helps us study algorithm performance in more detail.

4. **Store each graph instance with metadata**
   For each generated graph, we store:
   - Graph type
   - Number of nodes (N)
   - Seed (when applicable)
   - The actual Graph object
   - Source (s) and sink (t)

5. Everything is stored in a dataset list, and we optionally save it to graph_dataset.pkl for reuse.

6. **Use multiple seeds for randomness**
   For random graphs (sparse and dense), we generate five variants using SEEDS = {0,1,2,3,4}.
   This avoids bias from one specific random graph and gives more reliable performance averages.

7. **Cover both average-case and worst-case scenarios**
   The selected graph families are intentional:
   - **Sparse Random** → tests realistic, low-density networks

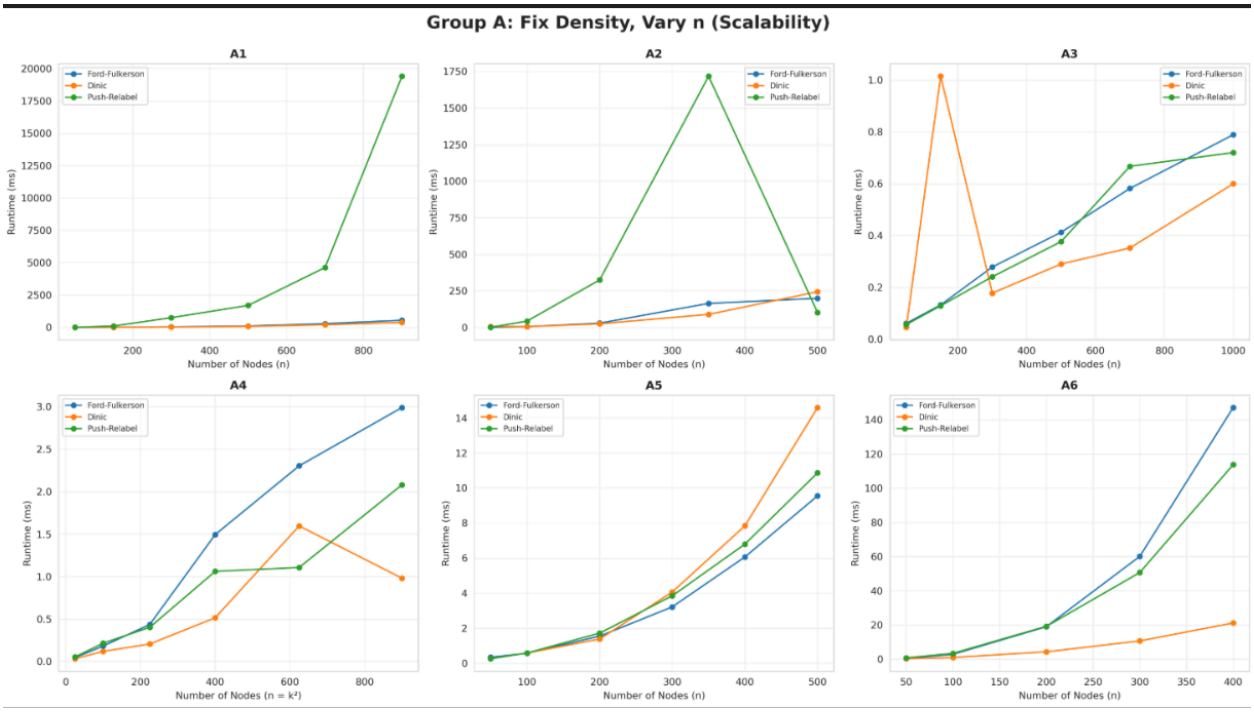
- **Dense Random** → tests high-density cases
- **Bipartite** → tests matching-like flows
- **Even–Tarjan** → known theoretical worst-case for Dinic
- **Diamond** → classic worst-case for Ford–Fulkerson
- **Star-of-Stars** → pathological case for Push-Relabel

8. By including these, we can observe:
    - how each algorithm scales with N
    - how they react to very different graph topologies
    - how they behave on their own theoretical worst cases

Results & Analysis

Group A: Runtime - n plots

By fixing the density and capacity values and scaling n (number of vertices) we generate graphs to analyse the runtime as n varies:



The figure compares the runtime scalability of Ford–Fulkerson, Dinic, and Push–Relabel across six graph families (A1–A6). Density is fixed for each family; n increases, so the plots reveal how each algorithm grows with problem size.

A1:- Dense Graphs

Push-Relabel as expected explodes as n increases, this is because of many relabels in worst-case scenarios.

A2:- Denser Graphs

All algorithms run quickly, and random spikes tend to dominate in small cases. In general, expected result is all algorithms performing reasonably well.

A4:- Grid Graphs

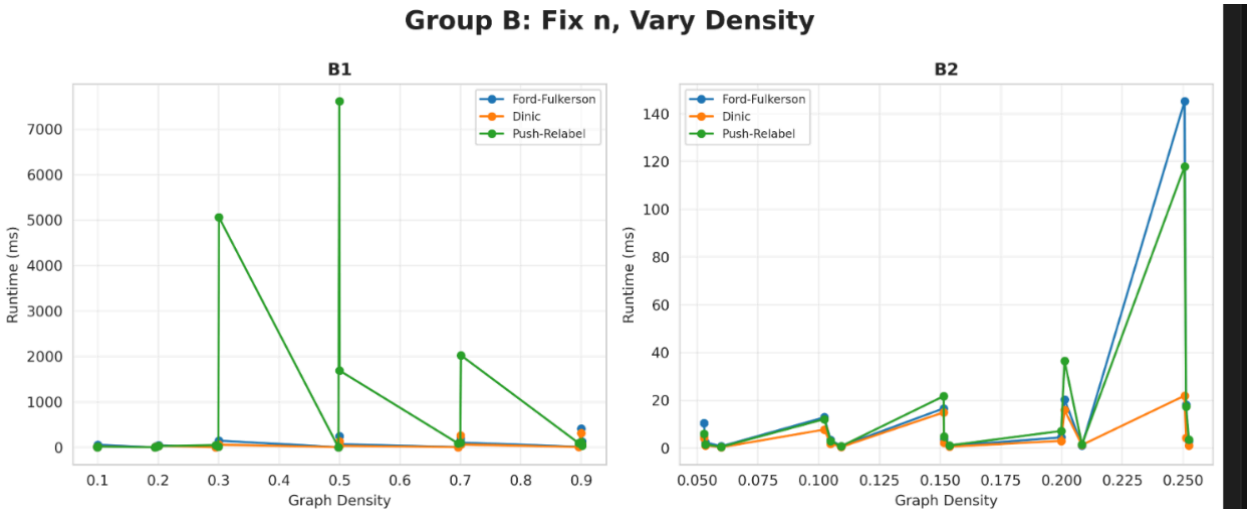
Ford-Fulkerson performs poorly as augmenting paths only happen one at a time, resulting in rerouting a lot.

A5:- Layered Graphs

Dinic performs poorly as it recalculates layers each time, while Ford-Fulkerson and Push-Relabel have same trends as elsewhere.

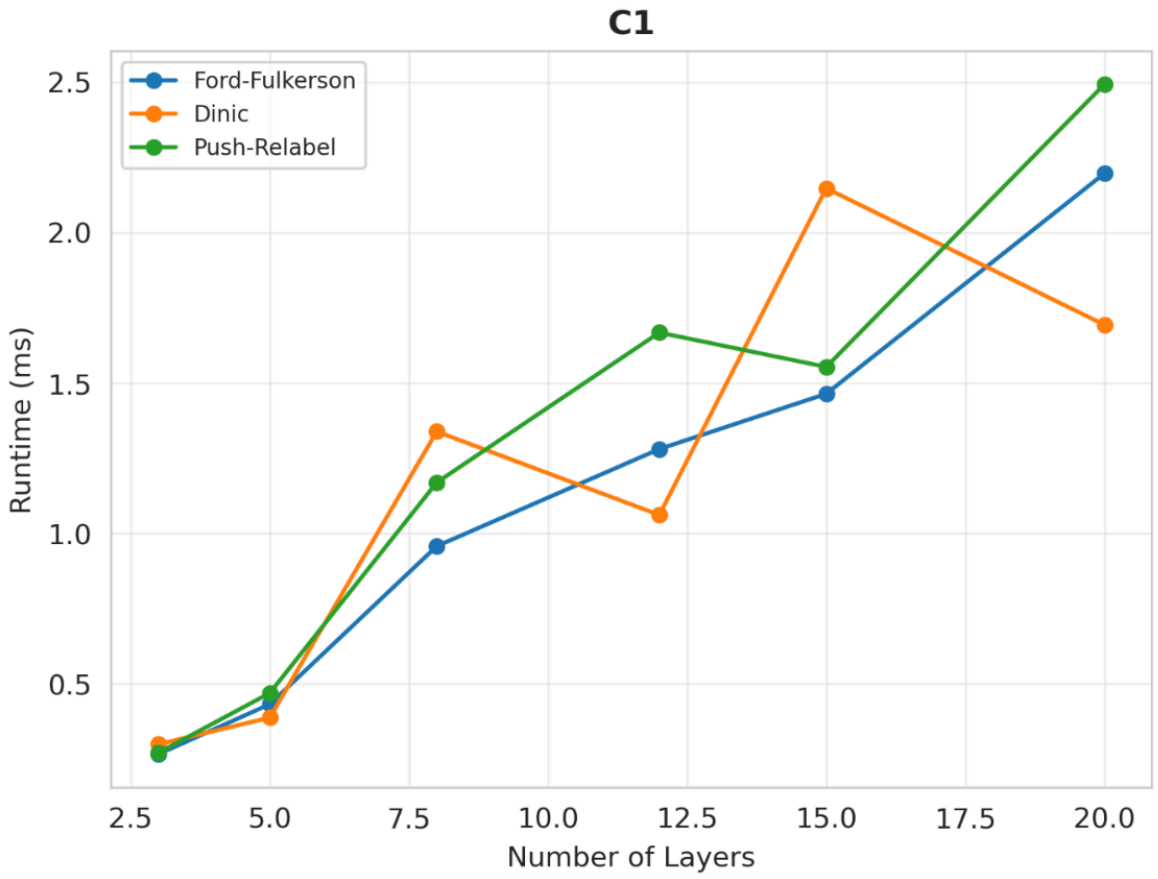
A6:- Bipartite Graphs

Dinic dominates as the number of edges is exponential, resulting in Ford Fulkerson and Push-Relabel becoming explosively worse.



In Group B (fixed n, varying density), **Dinic's algorithm is consistently the fastest** across all densities and remains almost flat, showing excellent scalability as edges increase. **Ford–Fulkerson** also performs reasonably but becomes slower as density increases because more augmenting paths must be explored. **Push–Relabel**, however, shows **highly unstable performance**: it is very fast at very low and very high densities but produces large spikes at medium densities. This is expected—medium-density sparse graphs create the worst-case behavior for Push–Relabel, causing excessive relabeling and slowdowns. Overall: **Dinic > Ford–Fulkerson > Push–Relabel** in stability, with Push–Relabel being highly sensitive to graph structure.

Group C: Fix Nodes-per-Layer, Vary #Layers

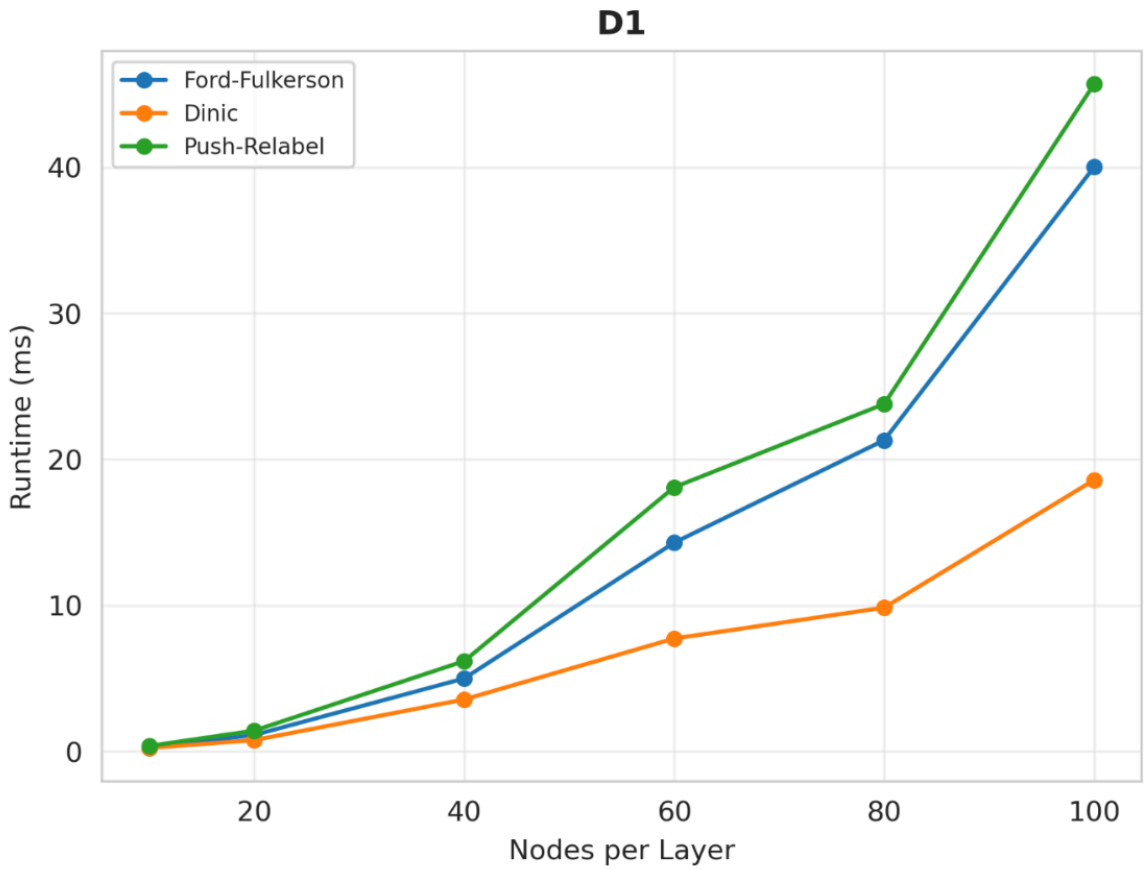


In Group C (fixed nodes per layer, increasing number of layers), all three algorithms show a **steady rise in runtime** as the graph becomes longer and deeper. This is expected because increasing the number of layers increases the length of augmenting paths and the total number of edges.

- **Ford–Fulkerson** grows smoothly and predictably with depth, showing stable performance.
- **Dinic** is fast at lower layer counts but shows occasional jumps due to variation in blocking-flow construction.
- **Push–Relabel** also increases steadily and becomes the slowest at the highest number of layers, since more layers lead to more relabel operations in long, chain-like graphs.

Overall, runtime grows roughly linearly with the number of layers, with **Ford–Fulkerson being the most stable**, while **Dinic and Push–Relabel fluctuate slightly but follow the same upward trend**.

Group D: Fix #Layers, Vary Nodes-per-Layer



Short Analysis (Group D: Fix #Layers, Vary Nodes-per-Layer)

In this experiment, we fix the number of layers and increase the number of nodes per layer, effectively making the graph wider. We measure **wall-clock runtime (ms)** as the primary metric.

As the number of nodes per layer increases, the total number of edges grows proportionally, so all three algorithms show **increasing runtimes**, which is consistent with their theoretical dependencies on **E** (number of edges).

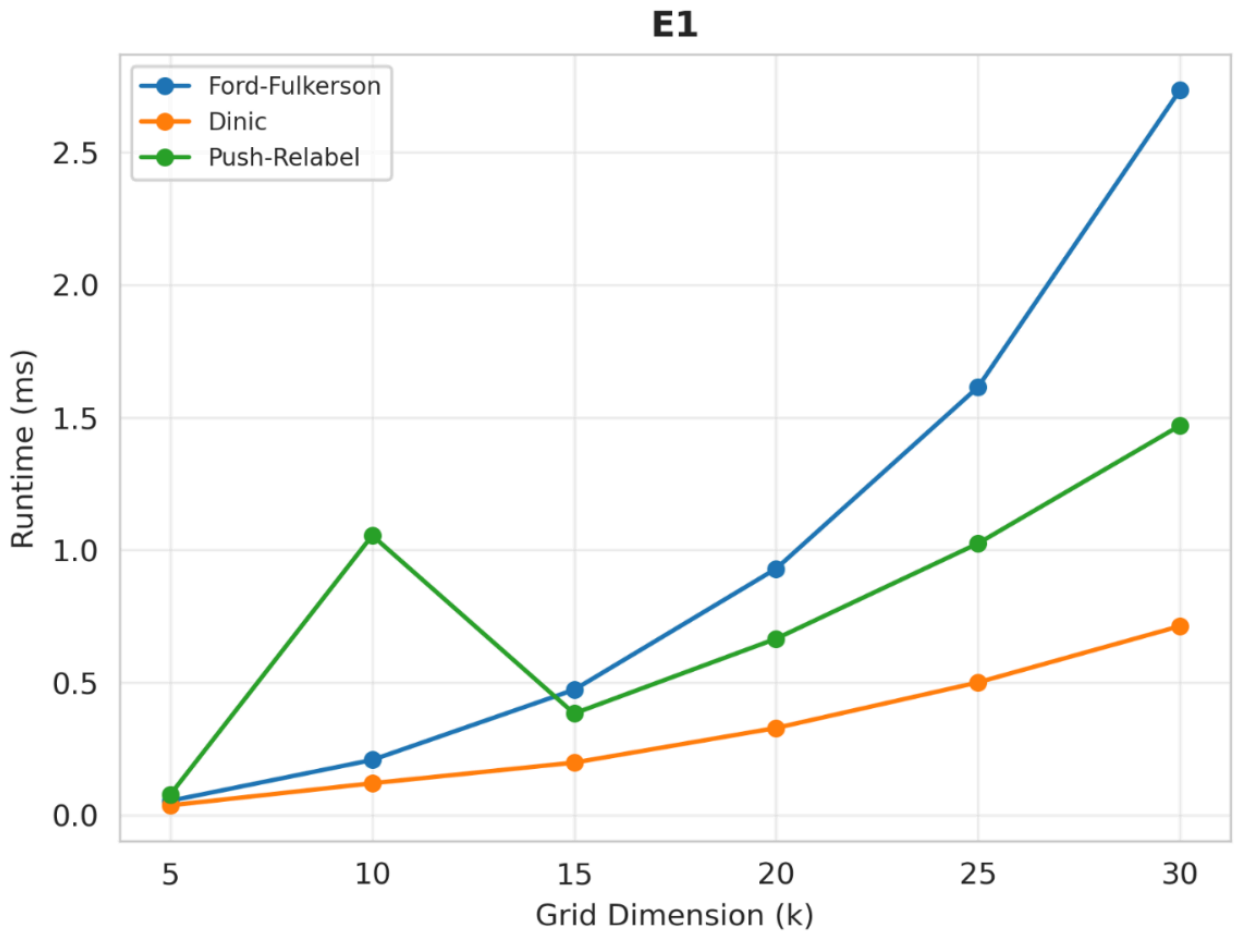
- **Dinic** performs the best overall, growing the slowest with width. This aligns with its near-optimal complexity of **O(E√V)** for layered, unit-capacity graphs, where the blocking-flow mechanism handles additional edges efficiently.

- **Ford–Fulkerson** grows faster than Dinic because it relies on repeated DFS searches for augmenting paths, causing runtime to increase noticeably as the number of possible paths expands.

- **Push–Relabel** becomes the slowest as graphs get wider, especially beyond 60–80 nodes per layer. This is expected because higher width increases the number of active nodes and relabel operations, causing its worst-case **O(V^3)** behavior to become more visible.

Overall, the empirical results closely match theoretical expectations: **Dinic scales best with graph width**, while **Ford–Fulkerson and Push–Relabel degrade more rapidly** as the graph becomes denser due to increased inter-layer connectivity and edge count.

Group E: Vary Grid Dimension k



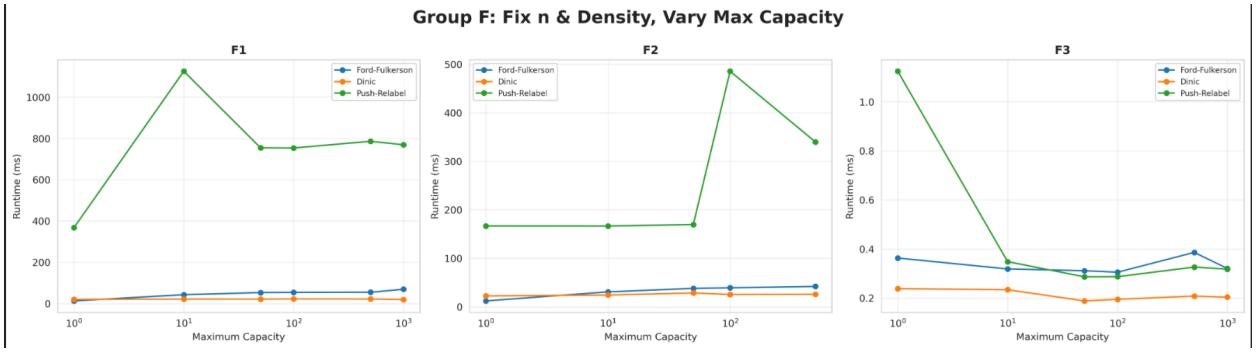
As the grid dimension k increases, all three algorithms show predictable growth, but at different rates.

Dinic scales the best, rising slowly and remaining the fastest across all grid sizes thanks to shallow BFS layers and efficient blocking flows in grid structure.

Push–Relabel settles into steady growth, staying consistently faster than Ford–Fulkerson for larger grids.

Ford–Fulkerson grows the fastest by far, with runtime increasing sharply as augmenting paths become longer in larger grids.

Overall, grid expansion highlights Dinic's superior scalability, Push–Relabel's moderate stability, and Ford–Fulkerson's poor performance as path lengths grow.



Conclusion

Across all experiments in this project, our findings consistently show that the practical performance of Ford–Fulkerson, Dinic's algorithm, and Push–Relabel closely reflects their theoretical design principles. Dinic's algorithm emerged as the most robust and scalable method overall. It performed exceptionally well across dense graphs, layered graphs, bipartite graphs, and grid-like structures, maintaining low runtimes even as graph size, density, or structural complexity increased. This is consistent with its efficient use of level graphs and blocking flows, which reduces redundant work and allows it to handle large and structured inputs effectively.

Ford–Fulkerson showed stable but slower growth trends. While it performed adequately on sparse and moderately dense graphs, its reliance on repeated DFS-based augmenting path searches caused runtimes to increase quickly in graphs with longer paths or high connectivity. This behavior was especially evident in layered and grid graphs, where path lengths expand naturally with graph depth or grid dimension.

Push–Relabel displayed the most variability. In some settings—particularly very dense graphs or wide graphs with many parallel edges—it performed competitively. However, it also showed sharp performance spikes in medium-density sparse graphs and complex topologies. These spikes are rooted in its inherent sensitivity to relabel operations and the number of active vertices, which can explode in worst-case scenarios. Our empirical results for Push–Relabel, including its instability in Groups B and E, align with its known theoretical O(V^3) worst-case complexity.

Overall, our experiments confirm that:

- **Dinic** offers the best balance of speed, stability, and scalability across nearly all graph families.

- **Ford–Fulkerson** is simpler but less efficient on large or deep graphs.

- **Push–Relabel** is highly dependent on graph structure and can oscillate between very fast and very slow performance depending on topology.

Limitations

Despite the breadth of our experiments, there are several limitations worth acknowledging:

1. **Python Overhead:**
   All algorithms were implemented in Python, which introduces non-trivial overhead. Lower-level languages (e.g., C++ or Rust) would provide more precise insights into pure algorithmic performance, especially for Push–Relabel.

2. **Single-threaded Execution:**
   All experiments were run on a single thread. Some max-flow algorithms—particularly Push–Relabel—can benefit significantly from parallelism, which we did not explore.

3. **Capacity Distributions:**
   We used fixed and uniform capacity ranges in most experiments. Real-world networks often have skewed, scale-free, or heavy-tailed capacity distributions that may change algorithm behavior.

4. **Graph Families Are Representative but Not Exhaustive:**
   Although we tested a large set of graph families (random, bipartite, worst-case, layered, grids), many real-world networks (social graphs, road networks, or industrial flow networks) were not included.

5. **Memory Usage Not Measured:**
   We focused on runtime as the primary metric. Push–Relabel and Dinic may use significantly different memory footprints, which could change conclusions in memory-intensive scenarios.

Future Work

Our findings open several promising directions for future exploration:

1. **Implement Algorithms in C++ or Rust:**
   A high-performance implementation would better isolate algorithm behavior from interpreter overhead and allow testing on much larger graphs.

2. **Parallel and Distributed Max-Flow:**
   Modern GPUs and multi-core CPUs can accelerate Push–Relabel significantly. Exploring parallel versions could lead to dramatically different performance rankings.

3. **Dynamic and Incremental Max-Flow:**
   Many applications require updating flows after small graph changes. Studying dynamic algorithms could provide deeper insight into real-world performance.

4. **Real-World Graph Datasets:**
   Testing on transportation networks, social networks, or industrial flow systems would validate how well these algorithms generalize beyond synthetic graphs.

5. **Memory Profiling and Cache Behavior:**
   Adding memory usage, cache efficiency, or instruction count measurements could give a richer performance profile.

6. **Hybrid Algorithms:**
   Investigating combinations of Push–Relabel and Dinic—already used in some state-of-the-art solvers—could potentially outperform both independently.