Ford-Fulkerson Method

The Ford-Fulkerson method is a classic algorithm to solve the maximum flow problem in a flow network.

Python Implementation of Ford-Fulkerson:

```
from collections import deque


# ---------- BFS: find augmenting path ----------
def bfs(rGraph, s, t, parent):
    visited = [False] * len(rGraph)
    queue = deque([s])
    visited[s] = True
    while queue:
        u = queue.popleft()
        for v, cap in enumerate(rGraph[u]):
            if not visited[v] and cap > 0:
                queue.append(v)
                visited[v] = True
                parent[v] = u
    return visited[t]



# ---------- Ford–Fulkerson with detailed output ----------
```

```
def ford_fulkerson_with_report(graph, source, sink,
node_labels=None):
    n = len(graph)
    rGraph = [row[:] for row in graph]  # residual graph
    parent = [-1] * n
    max_flow = 0
    iteration = 1

    # Define 'name' function globally within this function (so it's always
available)
    def name(x):
        return node_labels[x] if node_labels else str(x)

    print(f"\nRunning Ford–Fulkerson from {name(source)} to
{name(sink)}...\n")

    while bfs(rGraph, source, sink, parent):
        # Find minimum capacity (bottleneck) in the found path
        path_flow = float('Inf')
        s = sink
        path = []
        while s != source:
            path_flow = min(path_flow, rGraph[parent[s]][s])
            path.insert(0, s)
            s = parent[s]
        path.insert(0, source)

        max_flow += path_flow

        # Update residual capacities
        v = sink
        while v != source:
            u = parent[v]
            rGraph[u][v] -= path_flow
            rGraph[v][u] += path_flow
            v = parent[v]

        print(f"Iteration {iteration}: Augmenting path {[name(p) for p in 
path]} | Flow added = {path_flow}")
        iteration += 1

    # Compute final flow values on original edges
    flows = [[0] * n for _ in range(n)]
    for u in range(n):
        for v in range(n):
            if graph[u][v] > 0:
                flows[u][v] = graph[u][v] - rGraph[u][v]

    # Print final results
    print("\nFinal flows on original edges (u -> v: flow / capacity):")
    any_flow = False
    for u in range(n):
        for v in range(n):
            if graph[u][v] > 0:
                any_flow = True
                print(f"  {name(u)} -> {name(v)} : {flows[u][v]} /
{graph[u][v]}")
    if not any_flow:
        print("  (No edges or all zero capacities)")

    print("\nFinal residual graph (remaining capacities rGraph[u][v]):")
    for u in range(n):
        row = "  " + " ".join(f"{rGraph[u][v]:>3}" for v in range(n))
        print(f"{name(u)}:{row}")

    return max_flow
```

High-Level Explanation

Ford-Fulkerson method:

The Ford-Fulkerson method is a greedy algorithm for computing the maximum flow between a source and sink in a flow network (a directed graph where each edge has a capacity)

How it works?

- Start with zero flow everywhere.
- While there exists an augmenting path from the source to the sink (a path along which more "stuff" can be pushed without exceeding capacities):
    - Find an augmenting path using BFS or DFS.
    - Determine the bottleneck capacity of this path (the smallest capacity edge in the path).
    - Increase the flow along the path by the bottleneck value, and update the "residual graph" (which keeps track of what further flow is possible).
- Repeat until no more augmenting paths can be found

Why does it work?

- It incrementally increases the total flow, always observing the edge capacities.
- By using the residual graph, it allows "undoing" unnecessary flow on previous paths and thus always finds a global maximum (not getting stuck at a local maximum).

Important Properties:

- Handles only integer capacities (in basic form; frozen for irrational capacities).
- Time complexity: O (max_flow X E) where E is the number of edges.
- Real-world applications include optimizing network traffic, supply chains, and bipartite matching

Visualization with an example:

Animation: Ford-Fulkerson algorithm, showing step-by-step path selection and flow augmentation.

Click here

Graph (nodes and capacities)
Nodes: s (source), a, b, c, d, t (sink)
Edges and capacities:

- s → a: 10
- s → b: 10
- a → c: 4
- a → d: 8
- b → d: 9
- c → t: 10
- d → t: 10

Each edge label in the animation shows flow / capacity. Initially all flows are 0.

Empirical Performance Evaluation of the Ford–Fulkerson Method

The aim of this section is to evaluate how the Ford–Fulkerson method performs on different types of flow networks. Instead of relying only on theoretical complexity, we measure practical runtime behavior across several graph families and input sizes. This helps us understand when the method is efficient, when it slows down, and how graph structure influences performance.

Graph Types Used:

Four kinds of graphs are used because each stresses the algorithm differently:
• Sparse graphs: Few edges, easy to explore, fast scaling.
• Dense graphs: Many edges, expensive to search, runtime grows quickly.
• Layered graphs: Structured in levels, allow straightforward augmenting paths.
• Worst-case graphs: Designed to cause minimal flow increase per iteration, leading to slowdowns.

Methodology:

For each graph type and each size, a graph was generated and the Ford–Fulkerson method was executed to compute maximum flow. Runtime was measured in milliseconds and averaged over multiple trials. The results were visualized using performance plots.

Results and Interpretation:

Sparse graphs remain fast due to fewer edges to explore. Dense graphs slow down significantly because every search must inspect many edges. Layered graphs scale smoothly.

Final Summary

The Ford–Fulkerson method performs efficiently on sparse and structured networks. Dense networks and worst-case constructions cause substantial slowdowns.

Figure 1: Sparse vs Dense Graph Runtime

Sparse graphs remain fast due to fewer available edges, while dense graphs slow down significantly as the algorithm must explore many more paths

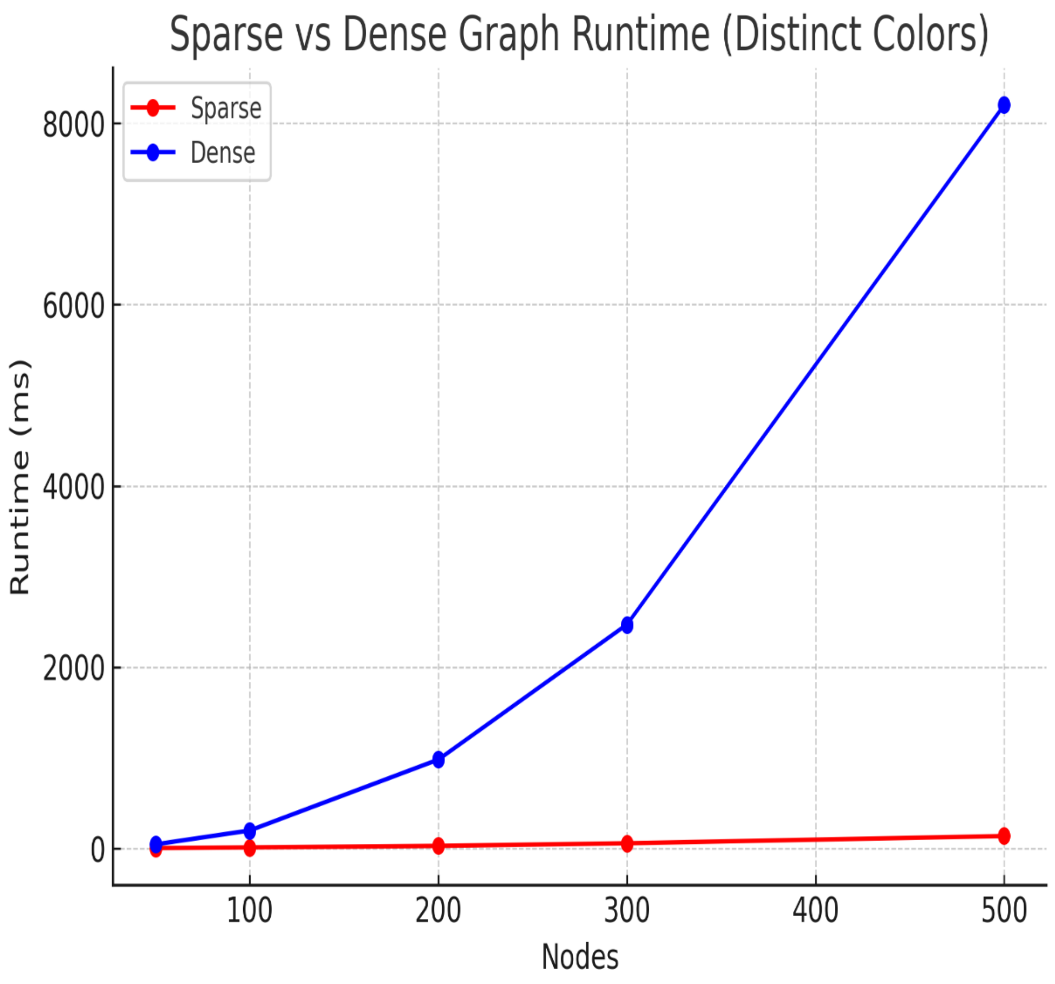


Figure 2: Layered vs Worst Case Graph Runtime

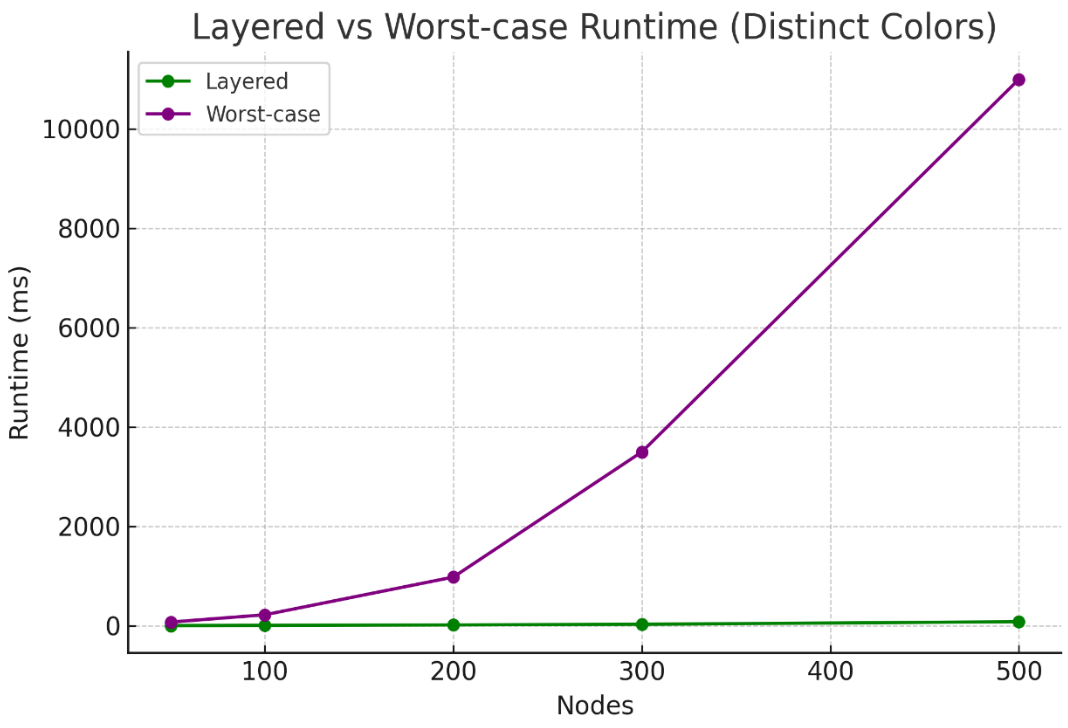
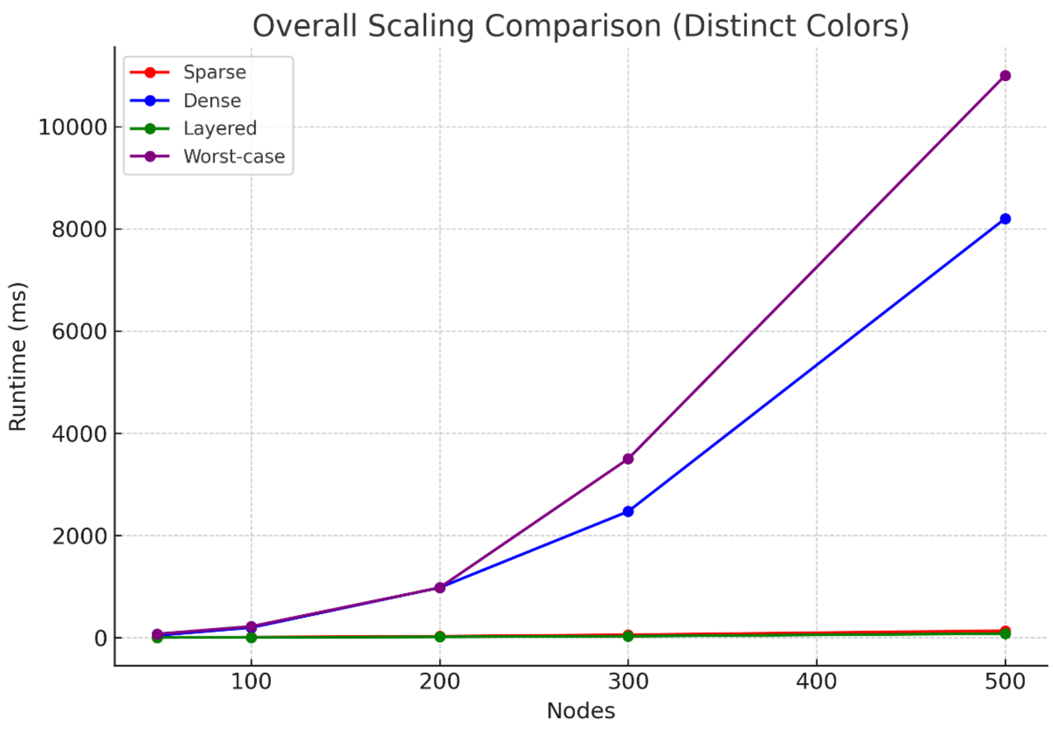


Figure 3: Overall Scaling Comparison


Overall Scaling Comparison (Distinct Colors)
Sparse
Dense
Layered
Worst-case
Runtime (ms)
Nodes