# Choice of Data Structure and Algorithm

I implemented an adjacency list to represent the flow network effectively. Each node contains a list of Edge objects, which capture essential details such as the destination node, the edge's capacity, and the current flow. This structure facilitates efficient flow management and analysis.

This structure was chosen because:

* It offers efficient traversal of outgoing edges ( needed for BFS in Edmonds-Karp).
* It saves memory for sparse graphs compared to an adjacency matrix.
* It’s easy to support residual edges, which are key in flow algorithms.

To support residual capacity, I included reverse edges in the graph with a capacity of 0 and a reference to the original edge, allowing quick updates during flow augmentation.

Algorithm:

I implemented the Edmonds-Karp algorithm, a specific implementation of the Ford-Fulkerson method that uses Breadth-First Search (BFS) to find the shortest augmenting path regarding the number of edges.

The Edmonds-Karp algorithm was chosen because:

* It guarantees polynomial time complexity of O(VE²).
* It is more predictable and stable than the generic DFS-based Ford-Fulkerson.
* It is simple to implement and works well with integer capacities.

# A Run of the Algorithm on the Smallest Benchmark Example

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

# Performance Analysis and Big-O Classification

To check out how well the algorithm performs, I mixed some hands-on testing with a bit of theory. I ran the algorithm with different benchmark files, testing it on graphs that got bigger and bigger, all the way up to millions of nodes and edges. I kept an eye on the actual run times and how it scaled by using the doubling hypothesis - basically, I doubled the size of the input and watched how that affected the execution time.

I put together some benchmark files (ladder\_1 to ladder\_17, bridge\_1 to bridge\_20) that get bigger and more complicated as you go along. For each file, I looked at: - How many iterations (or augmenting paths) it took - How long it took to calculate the max flow - How much memory was used. The findings show that small graphs wrap up in just milliseconds, while larger ones, like bridge\_13, can need over 14,000 iterations and might even throw an OutOfMemoryError because of the heavy path tracking.

The Edmonds-Karp algorithm works in O(VE²) time, which is a bit of a mouthful, but here’s the deal: each BFS takes O(E) time, and each edge can get fully used O(V) times. So if you look at the worst-case scenario, you'd have around O(VE) iterations overall, leading to that O(VE²) total complexity. This means it's a solid choice for dealing with moderately sized graphs that have integer capacities.