UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

5SENG003C.2 Algorithms: Theory, Design and Implementation

Full Name-Sanjula Nimesh Jayasinghe

UoW ID-w2053181

IIT ID-20230812

Tutorial Group – CS 08

Table of Contents

1. My Choice of Data Structure and Algorithm

For this assignment, I went with the Ford-Fulkerson algorithm using BFS to find augmenting paths (basically the Edmonds-Karp algorithm). I chose this approach for a few reasons:

First, using BFS guarantees the algorithm will finish in polynomial time. If I had used DFS instead, it might have taken exponentially longer in some cases, which wouldn't be practical.

Second, this approach works well when the network isn't super dense with edges, which is common in many real-world problems.

The main data structures I used were:

- Adjacency lists (ArrayList<Edge>[]) to represent the graph
- Edge objects that keep track of both forward edges and their residual counterparts
- A queue for implementing BFS
- Lists to keep track of all the augmenting paths and flows

I also decided to store all the augmenting paths as the algorithm runs, which isn't strictly necessary but makes it easier to see how the solution was built step by step.

2. Running the Algorithm on the Example

I tested my implementation on the small example network with 4 vertices and 5 edges:

4

0 1 6

0 2 4

1 2 2

1 3 3

2 3 5

In this network:

- Vertex 0 is the source
- Vertex 3 is the sink
- Each line shows an edge with its capacity

When I ran my algorithm on this network, here's what happened:

The algorithm found three augmenting paths:

1. First path: 0 -> 1 -> 3 with flow 3
2. Second path: 0 -> 2 -> 3 with flow 4
3. Third path: 0 -> 1 -> 2 -> 3 with flow 1

After all these paths, the final flow values on each edge were:

- From 0 to 1: 4 units of flow
- From 0 to 2: 4 units of flow
- From 1 to 2: 1 unit of flow
- From 1 to 3: 3 units of flow
- From 2 to 3: 5 units of flow

The total maximum flow came out to 8, which matches what we'd expect for this network.

3. Performance Analysis

Looking at how my implementation performs:

The time complexity of Edmonds-Karp is $O(VE^2)$, where V is the number of vertices and E is the number of edges. This comes from:

- The fact that we can find at most O(VE) augmenting paths
- Each BFS takes O(E) time
- Updating flow along a path takes O(V) time

For space, my implementation needs O (V + E) storage for:

- The graph structure
- The BFS queue
- Storing all augmenting paths

Some key things that affect performance:

1. Using BFS instead of DFS was an important choice because it guarantees we find the shortest augmenting path each time, which is crucial for the algorithm to run efficiently.
2. I implemented the residual network using explicit residual edges that are directly linked to their forward counterparts. This makes it faster to update flows since we don't have to search for the matching edge.
3. I also made sure the BFS stops as soon as it reaches the sink instead of exploring the entire graph unnecessarily.
4. I stored all augmenting paths for analysis, which is great for understanding and debugging but does use extra memory that isn't strictly required just to compute the maximum flow.

My approach works efficiently for small and medium networks. For example, network with 4 vertices, the algorithm found the maximum flow almost instantly. As I tested with larger networks, I noticed the performance scaled reasonably well up to networks with thousands of vertices. To summarize the complexity:

4. Handling Large Networks

When I started testing my implementation on very large networks, I ran into some memory issues. For example, with networks having tens of thousands of vertices and edges (like bridge_14.txt with 32,770 vertices and 65,537 edges), I encountered an OutOfMemoryError because my original implementation was storing all augmenting paths.

To address this limitation, I enhanced my implementation to be more memory-efficient:

1. I added an optional parameter to control whether all augmenting paths should be stored. For large networks, we can disable this feature and just track the total number of paths.
2. I implemented a threshold-based approach where detailed path information is only stored for networks below a certain size (10,000 vertices in my implementation).
3. I added proper error handling to gracefully handle out-of-memory situations and skip extremely large networks that would cause problems.
4. For very large networks, I modified the output to show just the essential information (network size, maximum flow value, path count, and computation time) rather than the full details.

These changes allowed my implementation to successfully process much larger networks. For instance, I was able to compute the maximum flow for bridge_13.txt (16,386 vertices, 32,769 edges) in about 10 seconds, finding all 16,385 augmenting paths.

The performance measurements across different network sizes showed a clear pattern:

- Small networks (≤100 vertices): near-instant processing (1-5ms)
- Medium networks (100-1,000 vertices): fast processing (5-100ms)
- Large networks (1,000-10,000 vertices): reasonable processing time (0.1-2.5 seconds)
- Very large networks (>10,000 vertices): longer processing times (2.5-10+ seconds)

This scaling behavior aligns with the theoretical $O(VE^2)$ complexity of the Edmonds-Karp algorithm.