Jožef Stefan International Postgraduate School

Maximum Flow in Graphs

Seminar assignment

Course: Programming, Data Structures and Algorithms (ICT2)

Supervisor: prof. dr. Anton Biasizzo

Student: Matic Kladnik

Study Year: 2021/2022

Table of Contents

1 Introduction

In this seminar we will focus on finding the maximum flow in a graph or network. This is an optimization problem that involves finding a feasible flow through a flow network to obtain the maximum possible flow rate. The problem is similar to the minimum capacity problem in graphs. A flow network is a directed graph where each edge has a capacity and receives a flow as weighted values.

Solutions to the maximum flow issue have many applications in various systems, such as water delivery through a pipe system [2], sewage capacity, logistics, traffic in computer networks, etc. One of these is also finding the maximum flows of currents in an electrical circuit. The main idea is to find the maximum quantity of some commodity, such as liquid or bits of data, that can be transferred on network between a source node or vertex and a sink node or vertex.

An interesting application of Ford-Fulkerson is a flow optimization application to allow sources in a computer network to adapt their bandwidth usage to the network, increase network utilization and mitigate congestion problems [1].

We will continue with a more detailed description of the maximum flow problem with some examples. We will look at the methodologies that we focused on and used in our implementation. These are the Ford-Fulkerson algorithm and Edmonds-Karp algorithm. After that we will continue with an overview of our implementation, after which we will reach the conclusion.

2 Description

Maximum flow problem is a type of a network optimization problems. It is also known as blocking flow. Let us say we are given a directed graph $G = (V, E)$, where each edge e has a capacity $c(e) > 0$, and where there are two special nodes, s (source) and t (sink). Our goal is to maximize the amount of flow from s to t, where flow on any edge cannot exceed its capacity.

Another way to look at the problem is to see it as a minimum cut problem. In this case we want to remove edges from the graph to remove any possible paths from the source node to the sink node. To get the minimum cut, we remove those edges that have a minimum combined cost, while still removing all paths between the source and the sink nodes.

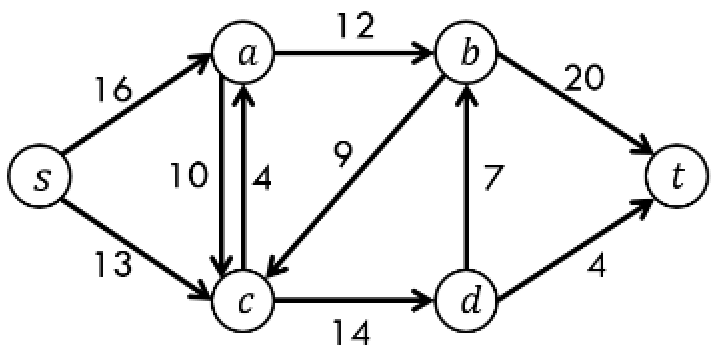

Figure 1 - Sample directed graph

If we look at Figure 1 above, we can see the source node is positioned on the left of the graph and the sink node on the right of the graph. We can see that there are two edges connected towards the sink node, *b-t* and *d-t*, with summed capacity of 24. However, node b can only be fed up to 19 units, meaning we cannot use the full capacity of edge b-t, which is 20. This gives the maximum flow of this graph of 23.

In Figure 2 below we can see how the capacities of each edge are used to get to the maximum flow of 23.

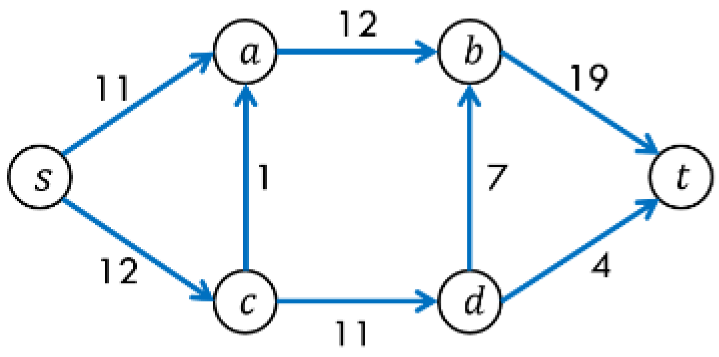

Figure 2 - Maximum flow in the sample graph

3 Methodologies and Implementation

The main algorithm that we will focus on in this seminar is the Ford-Fulkerson algorithm. It is also known as the Ford-Fulkerson method, as some parts of the protocol are not specified in detail and there are various ways to implement those parts. We will also look at the Edmonds-Karp algorithm, which is an improved implementation of Ford-Fulkerson. After which we present an overview of our implementation of the algorithm which is based on the Edmonds-Karp algorithm.

3.1 Ford-Fulkerson Algorithm

We continue by looking at the pseudocode of the Ford-Fulkerson algorithm below.

```
initialize flow to 0
path = findAugmentingPath(G, s, t)
while path exists:
    augment flow along path
    G_f = createResidualGraph()
    path = findAugmentingPath(G_f, s, t)
return flow
```

Figure 3 - Loosely defined Ford-Fulkerson pseudocode

As we can see, some of the lines are not defined in detail, such as the “augment flow along the path” line. To give a bit more detail about it, we can look at a more well-defined pseudocode below.

Augmenting flow along the path is done by increasing the flow of each edge on found path with the residual capacity. Forward edges are those that exist in the original graph G, while backwards edges are created during the process by reversing the original edge.

Residual graphs are used when calculating the maximum flow. They are used to find augmenting paths between the source and sink nodes.

```
flow = 0
for each edge (u, v) in G:
    flow(u, v) = 0
while there is a path, p, from s -> t in residual network G_f:
    residual_capacity(p) = min(residual_capacity(u, v) : for (u, v) in p)
    flow = flow + residual_capacity(p)
    for each edge (u, v) in p:
        if (u, v) is a forward edge:
            flow(u, v) = flow(u, v) + residual_capacity(p)
        else:
            flow(u, v) = flow(u, v) - residual_capacity(p)
return flow
```

Figure 4 - Detailed Ford-Fulkerson pseudocode

Residual capacity is a new capacity of an edge after we subtract the quantity of the flow. Each time we find a path from node s to node t, we have to find the minimum capacity of all edges on that path as that is the maximum increase of the flow.

After we have exhausted all paths between the source and the sink nodes, we get the value of the maximum flow for a specific graph.

For a better understanding of the algorithm, we will look simulate one loop in a sample graph.

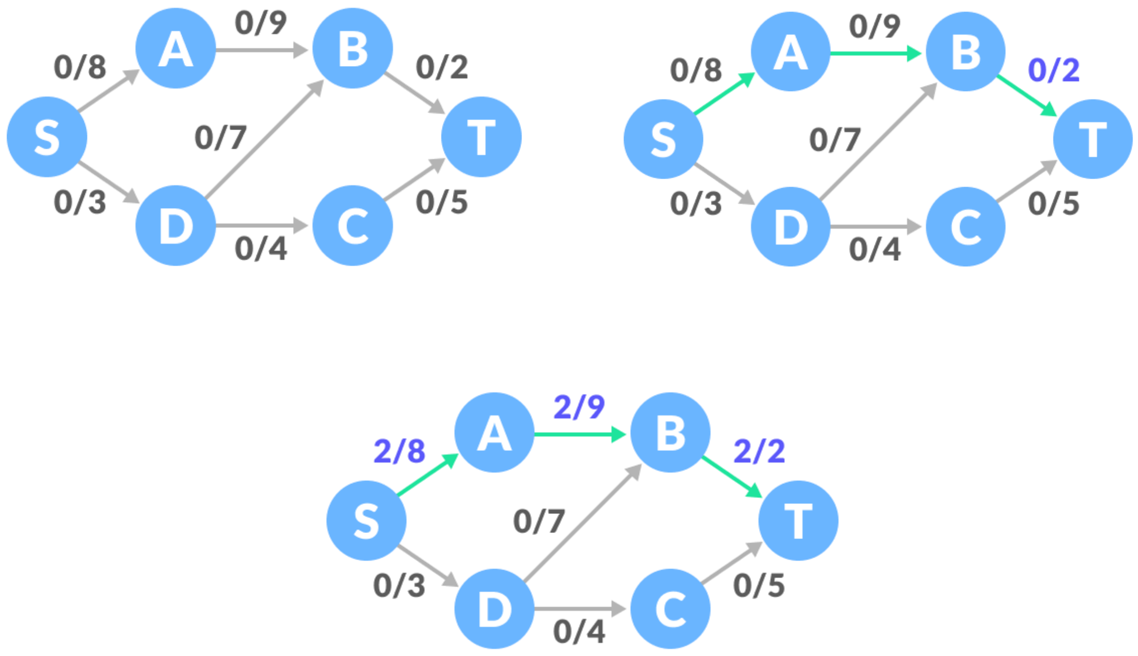

Figure 5 - Simulated loop of Ford-Fulkerson algorithm

As we can see in Figure 5 above, we start (top-left image) by setting the flow of each edge in the sample graph to 0. We continue (top-right image) by finding a path between node S (source) and node T (sink) and finding the minimum capacity on the path. In this example, the minimum capacity is present on the edge B-T and is equal to 2. In the next step in first iteration of the loop, we add the minimum capacity to the flow of the edges on the path (bottom image). After this we subtract those values from the reversed edges, e.g., reversed edge B-A would have a flow of -2.

In terms of complexity, the Ford-Fulkerson algorithm is bound by **O(E f_max)**, where E is the number of edges in the graph and f_max is the maximum flow in the graph. Each path between nodes S and T can be found in O(E), and each time we increase the flow by an integer of at least 1, which gives the upper bound f_max. Incrementing flows in graph by 1 would be the least efficient implementation of flow incrementation (line 4 in Figure 3).

3.2 Edmonds-Karp Algorithm

We continue by looking at Edmonds-Karp algorithm. While Ford-Fulkerson algorithm or method is not fully specified, the Edmonds-Karp algorithm is. Edmonds-Karp algorithm is a specific implementation of Ford-Fulkerson algorithm with certain improvements. One of the big differences is that Edmonds-Karp specifies Breadth-First Search (BFS) to be used as an algorithm for finding augmenting paths between the source and the sink nodes.

```
inputs
    C[n x n] : Capacity Matrix
    E[n x n] : Adjacency Matrix
    s : source
    t : sink
output
    f : maximum flow
Edmonds-Karp:
    f = 0                 // Flow is initially 0
    F = [n x n]        // residual capacity array
    while true:
        m, P = Breadth-First-Search(C, E, s, t, F)
        if m = 0:
            break
        f = f + m
        v = t
        while v != s:
            u = P[v]
            F[u, v] = F[u, v] - m       // reducing the residual capacity
of the augmenting path
            F[v, u] = F[v, u] + m        // increasing the residual
capacity of the reverse edges
            v = u
    return f
```

Figure 6 - Edmonds-Karp algorithm pseudocode

We continue by looking at the pseudocode of Edmonds-Karp Algorithm above. As we can see on the Figure 6 above, Edmonds-Karp defines to use matrices to represent edges between nodes with an adjacency matrix and capacities of those edges with a capacity matrix. We also need to provide indexes of the source and sink nodes.

One of the improvements that can be made during the implementation of the algorithm, is to simply focus on using the adjacency matrix and incorporate capacities into it. Meaning, instead of having 0 and 1 or false and true values in the adjacency matrix, we can insert capacities of edges for each cell in the matrix that represents an existing edge between two adjacent nodes.

In terms of complexity, Edmonds-Karp algorithm is bound by **$O(|V|\ |E|^2)$** since each iteration in Edmonds-Karp is bound by $O(|E|)$ and there are at most $|V|\ |E|$ iterations. This is an improvement as it makes the runtime of Edmonds-Karp independent of the maximum flow of the network (f_max). It is especially important in graphs with large flows.

3.3 Overview of the Edmonds-Karp Implementation

Our implementation of Edmonds-Karp algorithm is done in Python. We use type hints which are available since Python 3.5. They help improve code readability and ease of maintenance.

In our implementation of the Edmonds-Karp algorithm, we use an [n x n] adjacency matrix to represent the graph, where n is the number of nodes. We could also use a representation with node and edge objects with pointers. In case the graph we are processing is large, we can use the sparse-matrix as the adjacency matrix. This would require some rework in the implementation but would greatly reduce the memory consumption during the runtime of our implementation.

We use index values to define source and sink nodes. Instead of initializing and defining the path list each time we run the BFS path-finding function, we can simply initialize and define it once and then modify it during the runtime as the previous path can be ignored when searching for a new path.

As an optimization to the pseudocode of Edmonds-Karp algorithm, we use just one adjacency matrix to store information about edges between nodes as well as capacities of those edges. To save on memory, we use the provided adjacency matrix as a residual matrix by adding reverse edges and modifying it during runtime.

3.4 Dinic’s Algorithm

Let us continue by looking at the Dinic’s algorithm for finding maximum flow in graphs. Dinic’s algorithm uses various concepts to support the implementation. One of which is the **residual flow graph**, which is a directed graph with edges that have a specific maximum capacity. Apart from the capacity each edge has a flow value associated with it which holds the current flow that is passing through the edge. The flow of a specific edge cannot be higher than the edge’s maximum capacity. This flow graph is residual because it allows us to undo flow by going backwards between the edges (due to the defined directions of edges). Flow graph has a source node (s) and a sink node (t). When initialized, the flow through each edge is set to 0.

The second concept is the **augmenting path**. This is a path of edges from source (s) to sink (t) node with an unused capacity (also known as residual capacity). Each augmenting path has a bottleneck value, which is equal to the minimum unused (or residual) capacity of edges on the augmenting path.

The next concept is the **level graph** L (also known as the layered graph L). This graph assigns each node the distance from the source node (s). Where distance is the number of edges between a specific node and the source node. This graph is obtained via a Breadth-first search (BFS) method from the source to all other nodes. An edge is included in the level graph if it makes progress towards the sink (t) node. The key is that edges go from a specific node at level i to another node at level i + 1. With this approach we reduce the backwards edges.

Another concept is **augmenting the flow**. This means to update the flow values of edges along the augmenting path. For forward edges, we increase the flow by the bottleneck value on the augmenting path. At the same time, we need to decrease the flow along each residual edge (also known as backwards edge) by the same bottleneck value. Residual edges are used when we want to undo or revert bad augmenting paths which do not lead us to a maximum flow.

Before continuing to the pseudocode and a more detailed description, let us look at the general algorithm procedure. First, we create the level graph with the BFS approach on the current flow graph. If the sink is never reached while building the level graph, we stop and return the maximum flow value. Then we perform multiple DFS (Depth-first search) methods using only valid edges in the created level graph in order to find augmenting paths – this is sometimes referred to as sending multiple flows. Here is one of the main optimizations compared to Edmonds-Karp where we only send one flow at a time. We do this until we reach a blocking flow and finish the process by summing up the bottleneck values of all augmenting paths that were found to calculate the max flow. A blocking flow exists when no more flow can be sent through an edge on it using the level graph.

We continue by looking at the pseudocode below.

```
G = ((V, E), c, f, s ,t
c(u,v) – capacity, f(u,v) - flow
G_f – residual graph
G_L – level graph
1) Set f(e) = 0 for each e in E
2) Construct G_L from G_f of G.
     If dist(t) = Inf -> stop and output f.
3) Find a blocking flow f' in G_L
4) Increase augment flow f by f' and return to  step 2)
```

Figure 7: Dinic's algorithm pseudocode

In terms of complexity, the time complexity of Dinic's algorithm is bound by **O(V² E)**. Since the number of layers in each blocking flow increases by at least 1 each time, there are at most |V| - 1 blocking flows in the algorithm. And for each of them, the level graph G_L can be constructed by BFS in time bound by O(E) and a blocking flow in the level graph G_L can be found in time bound by O(V E). This means the total running time is bound by O(E + V E) = O(V E) for each layer. Consequently, the running time of Dinic's algorithm is bound by O(V² E). By using dynamic trees, the running time of finding a blocking flow in each phase can be reduced to O(E log V). In bipartite graphs, where the problem maps to a bipartite matching problem, the number of phases is bounded by O(sqrt(V)), therefore the algorithm is bounded by O(sqrt(V) E) in such case.

We continue by looking at an example or simulation of the Dinic's algorithm. In each step of the simulation, the network graph (G) appears on the left side, the residual graph (G_f) on the right side and the level graph (G_L) on the right side. In the level graph the red values in vertices represent distance to sink node (or level of the node) and paths in blue form a blocking flow.

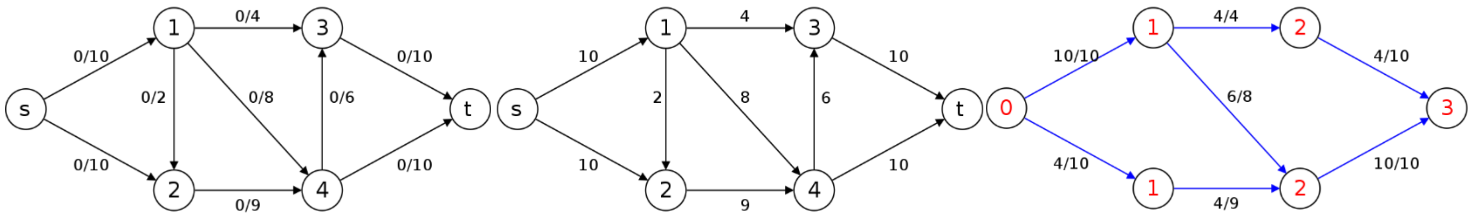

Figure 8: Dinic's algorithm simulation - phase 1

At the first phase of the simulation (Figure 8), the blocking flow consists of the following augmenting paths with 3 edges each: {s, 1, 3, t} with 4 units of flow, {s, 1, 4, t} with 6 units of flow, and {s, 2, 4, t} with 4 units of flow. The sum of these flows is 14, which represents the blocking flow, meaning the value of flow |f| is equal to 14.

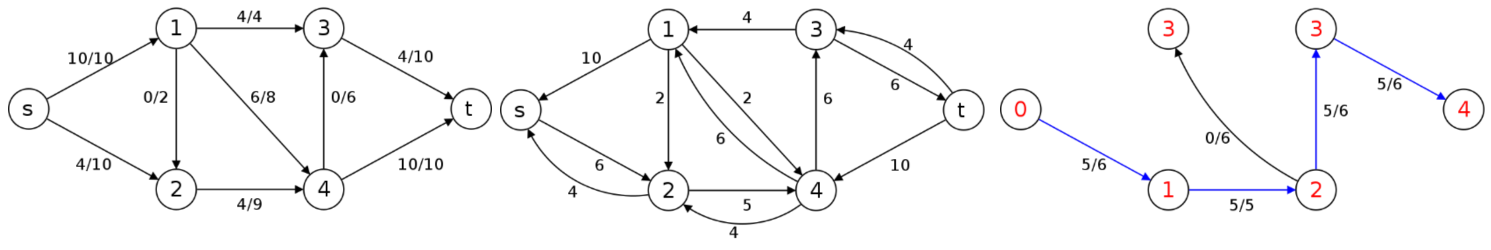

Figure 9: Dinic's algorithm simulation - phase 2

In phase 2 of the simulation (Figure 9), we find a blocking flow that consists of a single augmenting path with 4 edges: {s, 2, 4, 3, t} with 5 units of flow. The blocking flow has value of 5 and the value of flow |f| is now 19 (since we add 5 units to the previous 14).

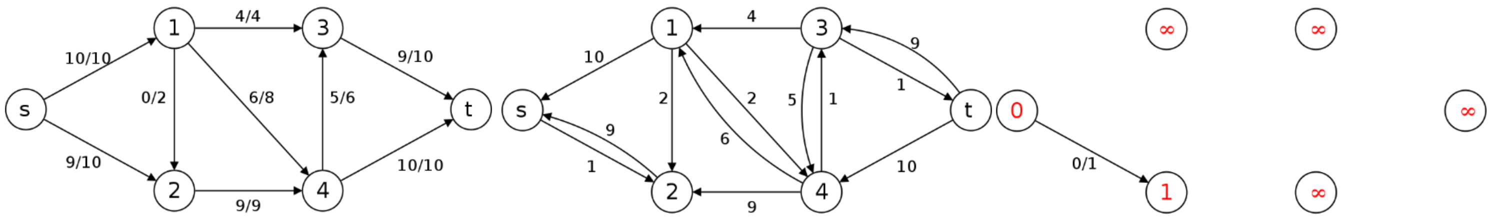

Figure 10: Dinic's algorithm simulation - phase 3

In phase 3 of the simulation (Figure 10), we cannot reach the sink node (t) in the residual graph G_f anymore. The algorithm terminates and returns the value of the maximum flow of 19 units. An interesting note is that in each step of the Dinic's algorithm, where we send multiple flows, the number of edges in the augmenting path increases by at least 1.

3.5 Overview of the Implementation of Dinic's Algorithm

We continue by looking at some details of our implementation of Dinic's algorithm. Here we also sue type hinting, so using Python of version at least 3.5 is required. The whole algorithm is implemented within a class so it can be simply used on a project simply by importing the class from the Python module (script).

We implemented construction of level graph with a level list where a value on index i denotes the distance of node i from source node. We denote the residual matrix as the flow matrix and the graph as the capacity matrix. Both matrices are implemented with an [n x n] matrix. In case of very large graphs, this could be further improved by using sparse matrices or linked lists to reduce memory demand.

As defined in the algorithm, we use a DFS method to find augmenting paths and a BFS method to construct a level graph (previously mentioned level list).

4 Conclusion

We started the seminar by looking at the problem of maximum flow in graphs in general. There are many applications for maximum flow in various complex systems. Throughout the years this network optimization topic has been studied and improved upon with various algorithms.

We have looked at one of the most important algorithms in finding the maximum flow in a directed graph, namely the Ford-Fulkerson algorithm and its improved implementation – Edmonds-Karp. Since the Edmonds-Karp algorithm is more efficient in terms of time complexity, as well as defined in more detail, we have used it as the basis of our implementation of finding the maximum flow in a directed graph.

Another implementation of finding the max flow in graphs was by implementing Dinic's algorithm. This algorithm is especially efficient on bipartite graphs, but is efficient and often used even on other directed graphs. It tries to find multiple flows in the same step of the algorithm and uses a level graph to determine distances of node to the starting node.

Currently our implementations work on graphs where flows have integer values. With some modification we could support graphs with flows of floating-point values. To support very large graphs with many nodes and edges, we could modify our implementation by using sparse matrices instead of [n x n] matrices to improve memory consumption. Linked lists would also be an option of a more efficient implementation.

To further explore the optimization problem of maximum flows in graphs, we could look at some additional algorithms, such as the Dinic's algorithm, general push-relabel algorithm, and push-relabel algorithm with dynamic trees.

5 Literature

[1] Gustavo Rau de Almeida Callou, “An Approach Based on Ford-Fulkerson Algorithm to Optimize Network Bandwidth Usage,” Conference, Brazilian Symposium on Computing Systems Engineering (SBESC), 2015

[2] Myint Than Kyi, Lin Lin Naing, “Application of Ford-Fulkerson Algorithm to Maximum Flow in Water Distribution Pipeline Network,” International Journal of Scientific and Research Publications (IJSRP), 2018, 8(12)

[3] https://en.wikipedia.org/wiki/Maximum_flow_problem

[4] https://brilliant.org/wiki/ford-fulkerson-algorithm/

[5] https://brilliant.org/wiki/edmonds-karp-algorithm/

[6] Jaehyun Park, “Network Flow Problems”, CS 97SI, Stanford University, 2015, https://web.stanford.edu/class/cs97si/08-network-flow-problems.pdf

[7] https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm

[8] https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm

[9] https://en.wikipedia.org/wiki/Dinic%27s_algorithm

[10] https://medium.com/smucs/understanding-dinics-algorithm-ebf892e66227