[[1]](#footnote-1)

Ford Fulkerson Max Flow Algorithm

*Student Name: Nachiket Bhagwat – nabh1518*

*Abstract*— In this paper, we will discuss what network flow algorithms are, what is max flow problem. We will find out techniques to solve max flow problem and test the techniques vs each other to find out advantages and limitations of them.

# INTRODUCTION

Network flow problems are very common in computer science and all branches of engineering. Any network flow problem is optimization problem of some feature of the flow. If we are talking about no. of hops or cost of transferring data from source to destination, in the network, the problem becomes shortest path problem. If we are talking about maximum throughput of the system, the problem becomes max flow problem.

Any network flow problem has following features.

* Network (directed graph, digraph): m nodes connected by n directed arcs
* Arcs are ordered pairs (i, j) of nodes
* We assume there is at most one arc from node i to node j
* There are no loops (arcs (i, i))

In optimization theory, maximum flow problems involve finding a feasible flow through a single-source, single-sink flow network that is maximum.

The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem. The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the network, as stated in the max-flow min-cut theorem. We will prove this later.

Given a network (G = (V, E), s, t, c), the problem of finding the maximum flow in the network can be formulated as a linear program by simply writing down the definition of feasible flow.

We have one variable f (u, v) for every edge (u, v) ∈ E of the network.

The problem is as follow:

**Maximize**

**Subject To:**

**=**

**f(u, v) ≤c(u,v) for all (u,v)**

**f(u,v) ≥ 0 for all (u,v)**

Mathematically, it means, maximize flow from a source to a destination, constrained to following.

No node in the network accumulates flow at any time, hence flow in the node = flow out of the node.

There is non negative flow through any arc limited by positive capacity of the arc.

The problem has historical significance. The maximum flow problem was first formulated in 1954 by T. E. Harris and F. S. Ross as a simplified model of Soviet railway traffic flow. During cold war, US army general thought about a problem of blocking good transport in Russia. To do this effectively, they needed to know what is maximum capacity to carry the train network and which tracks are most critical for it. In this way, the problem is to find out minimum cut in an undirected graph with paths from source node and destination node.

There are multiple solutions to this problem. While this can be set up as a linear programming problem with as many equations as there are cities in the network, and hence can be solved by the simplex method, it turns out that in the cases of most practical interest, where the network is planar in a certain restricted sense, a much simpler and more efficient hand computing procedure can be described.

Ford and Fulkerson came with very simple solution in 1956. We will look at the solution now.

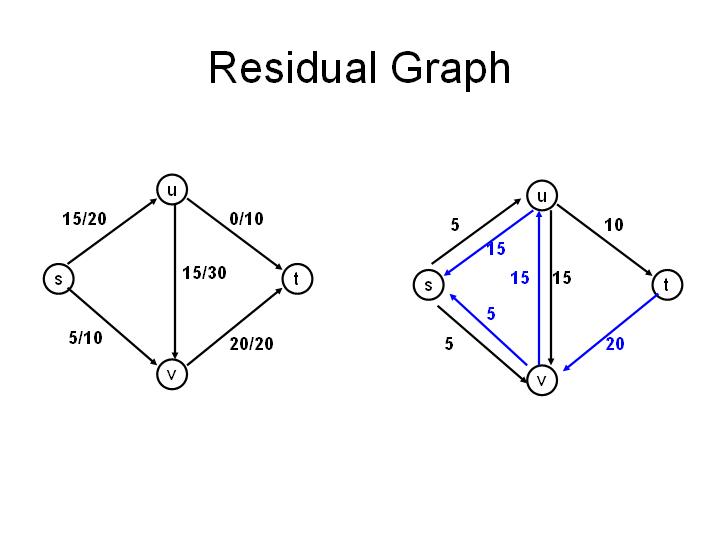
# Ford Fulkerson Max Flow Algorithm

In this algorithm, we use concept of residual graph.

Given a flow network G, and a flow f on G, we define the residual graph Gf of G with respect to f as follows.

* The node set of G\_{f} is the same as that of G.
* Each edge e = (u, v) of Gf is with a capacity of ce - f(e).
* Each edge e' = (v, u) of Gf is with a capacity of f(e).

Intuitively, we can explain this as followed.



In this picture, we ae passing data from s to t. We can have multiple paths from s-t eg, sut or svt or suvt.

Suppose we are passing a 5-unit flow from s to v. Then, we have new capacity for edge s-v as 10-5 = 5. This means, we can pass 5 more units forward from s-v.

At the same time, we are passing 5-unit flow from s to v. This means, we are passing -5-unit flow from v to s. Hence, we can pass 5 unit flow more from v to s. So, we add a back edge with capacity 5 and flow 0.  
In this figure, we are just showing ce-f(e) for both backward and forward edges.

# Algorithm

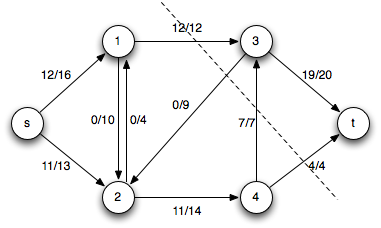
**Solution:**

* *Start with f(e) = 0 for all edge e ∈ E*
* *Find an s-t path P where each edge has f (e) < c(e).*
* *Find out bottleneck around that path and pass on as much flow as bottleneck.*
* *Augment flow along path P as shown in figure above.*
* *Repeat until you get stuck.*

**Proof of Correctness:**

The algorithm finds out every path from s to v. In each step, if there exists a path from s to v, we increase flow by adding more flow along that path. We create a residual graph from it. Hence, after some iterations, there would be no path from s to v.

This problem is same as finding out min cut in a weighted directed graph where we want to find minimum cut such that we don’t have any s-t path left. When we find out bottleneck around a path, the edge, that is creating bottleneck, is critical edge. This is because at equilibrium, along every path, there should exist at least one edge that has flow as much as capacity. Otherwise, we can pass more flow along the edge and hence, equilibrium is not yet reached. But as we know equilibrium is already reached, this implies, for every path, there exists at least one edge that is carrying flow as much as capacity.



In above example, path 1-3, 3,2 and 4,3, 4-t are all bottlenecks, or critical edges. Hence, they create a min-cut for s-t tree.  
Note that, we can not find any path from s-t if we remove these edges.

This is useful in real life situation of Soviet railway traffic flow. If you can break the 4 edges in this setting, you can not pass anything from s-t. This could be used by military to create maximum damage on transportation system of the enemy.

**Running Time Analysis:**

Let’s assume that all the capacities are integral values **.**

In this case, assume Fmax is the maximum flow possible in the system.

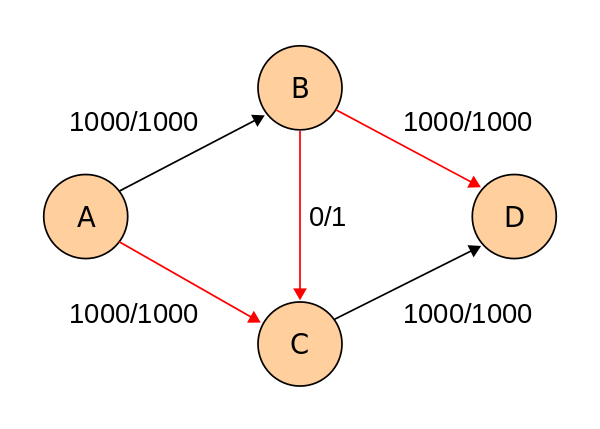
**Running time = Time Taken for 1 iteration \* #Iterations.**

During each iteration, we traverse the whole graph, using backtracking algorithm like DFS. If there are |E| edges, DFS takes time O(|E|). Hence, for 1 iteration, we need time O(|E|).  
  
Also, during each iteration, we increase the flow by at least 1 as our capacities are integral. Hence, we would need maximum of Fmax iterations to reach equilibrium.

Hence, total running time of the algorithm =  
O(Fmax\*|E|).

**Limitations:**In the algorithm, we are not deciding which branch should be used first to find out path. Hence, we are not utilizing our system properly.

Consider following example.

  
In this example, we are trying to find max flow along A-D. Path ABCD is a valid path. Hence, we can send a flow of unit 1 along that path. After this, we have to add a backedge from C to B of size 1 and capacities of AC and BD are reduced to 999.

Now, ACBD is valid path because of the added backedge. Hence, we can pass a flow of unit 1 along the path.

The process can continue 1000 times after which our system will reach equilibrium.

Even worse example would be if the capacity Ce(B,C) was irrational number. Suppose Ce(B,C) = r which is irrational.

Hence, after multiple iterations, the system may come to a state where flow along AC and BD = (1000-nr) and flow along BC = r and r > (1000-nr).

In this case, our system will never converge as r is not multiple of (1000-nr) as r is irrational.

In real life setting, although we can not have irrational numbers as capacities, but our capacities can be rational float value. In that case, our algorithm will take long time to converge.  
  
 The limitation stems from the fact that we are stuck in a loop because we are not working efficiently. To counter this problem, we will learn about another algorithm.

# Edmonds Karp Algorithm

In this scenario, we are using modified version of Ford Fulkerson algorithm. We will try to find out shortest path from s-t instead of choosing an arbitrary path.

**Solution:**

* *Start with f(e) = 0 for all edge e ∈ E*
* *Find the shortest s-t path P where each edge has f (e) < c(e) using breadth first search.*
* *Find out bottleneck around that path and pass on as much flow as bottleneck.*
* *Augment flow along path P as shown in figure above.*
* *Repeat until you get stuck.*

As this is modified version of Ford Fulkerson, this algorithm will also converge.

**Running time analysis:**

Running time = Time Taken for 1 iteration \* #Iterations.

During each iteration, we traverse the whole graph, using backtracking algorithm like DFS. If there are |E| edges, BFS takes time O(|E|). Hence, for 1 iteration, we need time O(|E|).

Now, an edge is critical if it is carrying maximum flow possible. Every edge can be critical in the graph. Hence, no. of maximum critical edges = |E|.

But each edge can become critical multiple times. In fact, every edge can become critical exactly |V|/2 times. Let’s see how.

Suppose an edge E1(u,v) became critical in a path of length from s-u is n. This is the current shortest path in the graph from S-T. As we have already visited all adjacent vertices of u during this iteration, we can not reach from s to u in exact n+1 times. We need at least n+2 times to reach a node u where (u,v) is again a critical edge. As there are |V| vertices, this situation can come at most |V|/2 times. Hence, every edge can become critical at most |V|/2 times.

#Iterations = max times an edge becomes critical \* no of critical edges.

Hence, #Iterations = |V|\*|E|.

Hence, running time = O(|E|) \* O(|V|\*|E|) = O(|VE2|).

Notice, our running time is completely independent of flow values. Also, every edge can become critical finite amount of times hence our algorithm works even for irrational flow values. This is huge improvement over Ford Fulkerson.

# Code Snippets

****

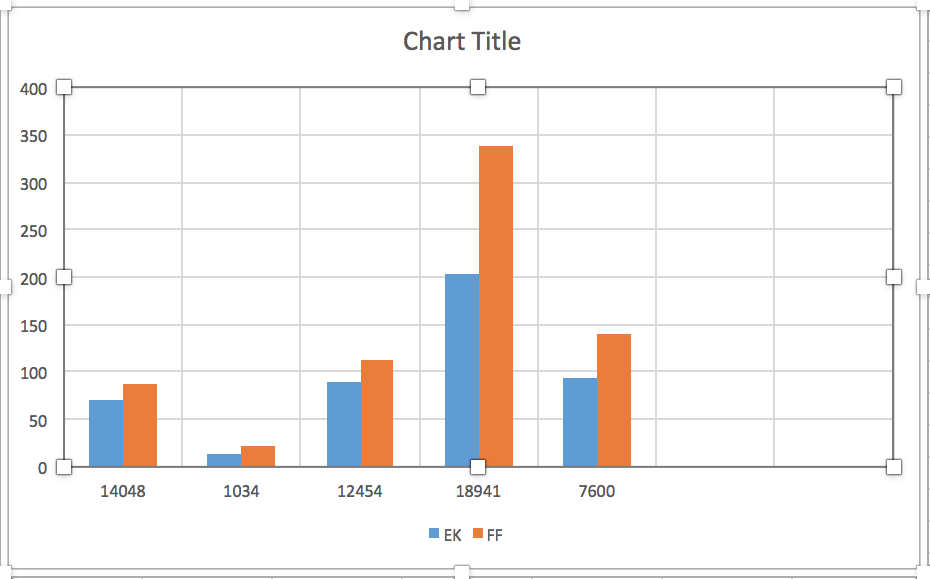
This is a code snippet of Ford Fulkerson algorithm. We are using stacks (Python implementation by queues) to implement DFS. We can use same code to implement Edmonds Karp algorithm if we use BFS with Queues.

# Empirical Analysis

I ran FF and EK on some random use cases by changing the no of nodes and vertices and passing the same input to both.

The output looks a bit like this. Edges vs No of Iterations for algorithm to converge.

|  |  |  |
| --- | --- | --- |
| **Edges** | **EK** | **FF** |
| 14048 | 70 | 88 |
| 1034 | 13 | 22 |
| 12454 | 90 | 113 |
| 18941 | 204 | 339 |
| 7600 | 94 | 140 |

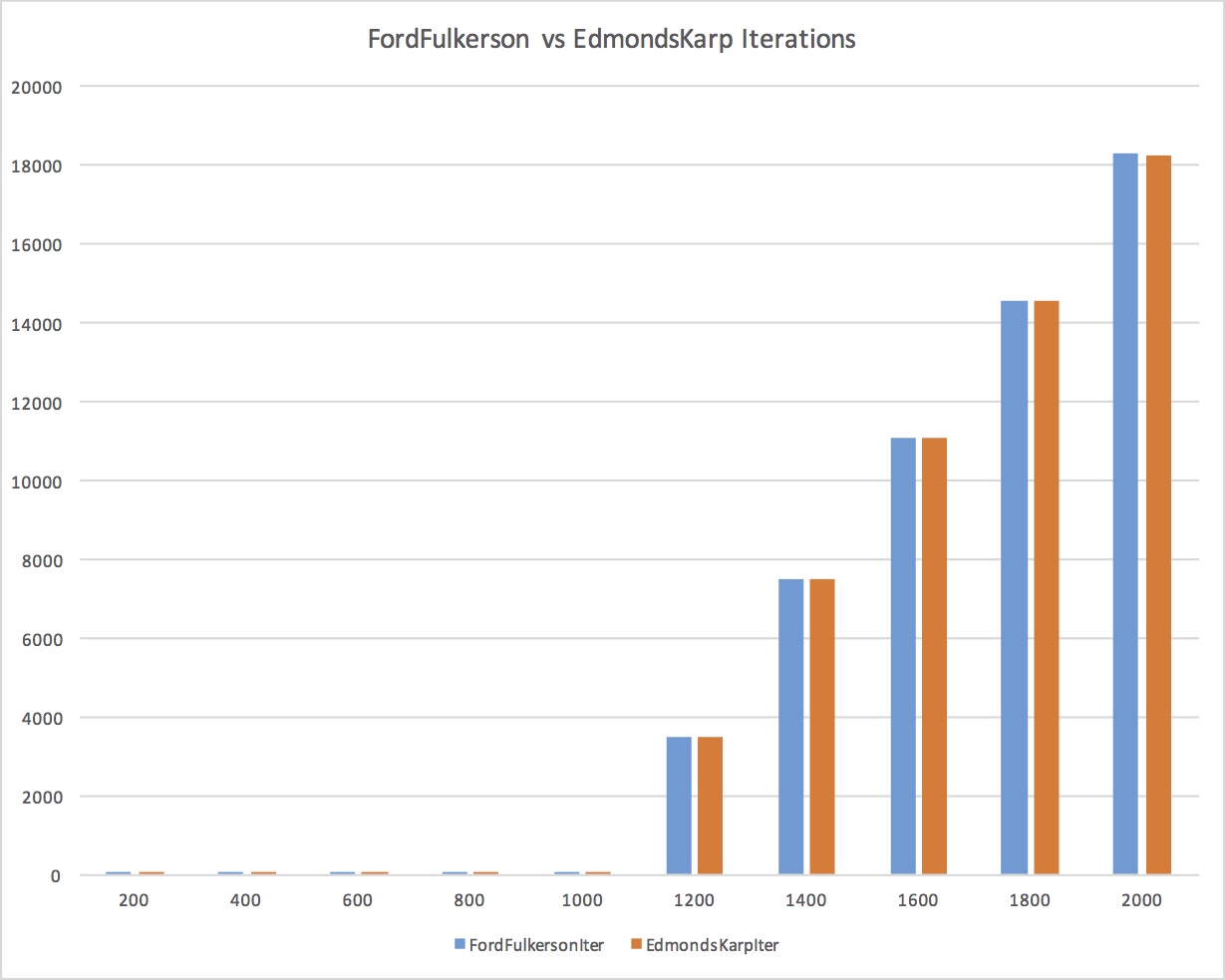


I also ran Ford Fulkerson and Edmonds Karp algorithms on dataset from University of Chicago Vision Problems datasets. I found significant differences while running both the algorithms.

The graph contains 2002 nodes and 47872 edges. As the algorithm takes a lot of time to converge, I divided the data in small chunks and linearly added number of nodes. After running both Ford Fulkerson and Edmonds Karp, I got following results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nodes** | **Edges** | **FFIter** | **EKIter** | **FFTime** | **EKTime** |
| 200 | 198 | 1 | 1 | 0.14 | 0.15 |
| 400 | 398 | 1 | 1 | 0.14 | 0.14 |
| 600 | 598 | 1 | 1 | 0.13 | 0.15 |
| 800 | 798 | 1 | 1 | 0.13 | 0.15 |
| 1000 | 998 | 1 | 1 | 0.14 | 0.15 |
| 1200 | 5383 | 3511 | 3493 | 1.8 | 19 |
| 1400 | 10083 | 7517 | 7499 | 12.88 | 150 |
| 1600 | 14783 | 11072 | 11054 | 30.5 | 493 |
| 1800 | 19483 | 14575 | 14557 | 53 | 914 |
| 2000 | 23915 | 18269 | 18251 | 80.2 | 1733 |

In above table, FFIter is number of augmented paths found by Ford Fulkerson algorithm. EKIter is number of augmented paths found by Edmonds Karp algorithm. As we can see, Edmonds Karp find less augmented paths because we always choose the shortest path. While no of augmented path were less in Edmonds Karp, the time taken to complete Ford Fulkerson vs Time taken to complete Edmonds Karp was opposite. Edmonds Karp was taking much larger time to find equilibrium. This maybe because in our setting, there maxflow is not very big. If we have less number of edges and more flow through each edge, we should see substantial differences in running times of the two algorithms.



# Uses of max flow algorithms

* **Disjoint paths and network connectivity –**

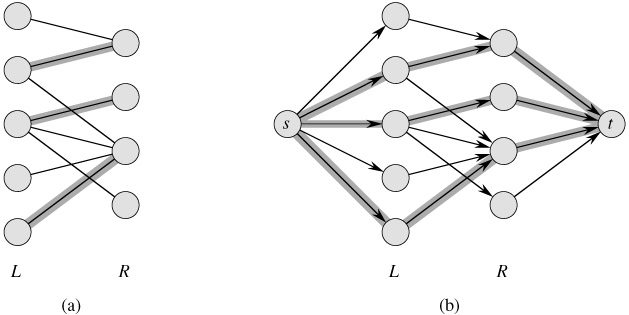
Disjoint path network: G = (V, E, s, t)

* + Directed graph (V, E), source s, sink t
  + Two paths are edge-disjoint if they have no arc in common.

We can assign unit capacity to all edges on paths from s-t. Run Max Flow Algorithm. As every edge has unit capacity, only paths which do not share any edges can contain flow value.

* **Bipartite matchings.**
  + Input: undirected, bipartite graph G = (L ∪ R, E)
  + M ⊆ E is a matching if each node appears in at most edge in M
  + Max matching: find a max cardinality matching.

We can use max flow algorithm to find out bipartite matching.



Consider above example of bipartite graph. We can add imaginary nodes s and t to the individual sets as shown in the figure and make the graph directed. Add unit capacity arcs from source to all nodes in L and from all nodes in R to t and infinite capacity arcs between L and R.

We can see that matching in (a) of Cardinality k introduces a flow k in graph (b).

By integrality theorem, there exists {0, 1}-valued flow f of value k.

* + Consider M = set of edges from L to R with f(e) = 1. each node in L and R participates in at most one edge in M
  + M| = k: consider cut (L ∪ s, R ∪ t)

Perfect bipartite matching.

* + Input: undirected, bipartite graph G = (L ∪ R, E), |L| = |R| = n.
  + Can determine if bipartite graph has perfect matching by running matching algorithm.

Bipartite matching is an important problem as it has applications in computer vision techniques.

* **Airline scheduling – Toy Example**

This is complex problem faced at airports while scheduling flights. It produces schedules for thousands of routes each day that are efficient in terms of. It has one of largest consumers of high-powered algorithmic techniques.

* + Manage flight crews by reusing them over multiple flights. Input: list of cities V
  + Travel time t(v, w) from city v to w
  + Flight i: (oi, di, ti) consists of origin and destination cities, and departure time.

# Future Work

I would like to know more max flow algorithms like Dinic’s algorithm and James B Orlin's + KRT algorithm which are working much faster than Edmonds Karp.

Citations:  
<https://en.wikipedia.org/wiki/Maximum_flow_problem>  
<http://link.springer.com/article/10.1007%2Fs101070100259>

<http://cms.math.ca/openaccess/cjm/v8/cjm1956v08.0399-0404.pdf>

<https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm>

<https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm>

<https://en.wikipedia.org/wiki/Dinic%27s_algorithm>

<http://ttic.uchicago.edu/~dbatra/research/mfcomp/>

<http://www.cs.princeton.edu/~wayne/cs423/lectures/max-flow-applications>

<http://www.cs.toronto.edu/~sven/Papers/bipartite.pdf>

<https://www.youtube.com/watch?v=XIfaR8U3c60>

1. [↑](#footnote-ref-1)