# A COMPARATIVE STUDY ON MAXIMUM FLOW ALGORITHMS

Maaz Syed
Indiana University
Bloomington, Indiana – 47408
maazsyed@indiana.edu

Vishwas Vijaya Kumar
Indiana University
Bloomington, Indiana – 47408
visvijay@indiana.edu

Venkat Sambandhan
Indiana University
Bloomington, Indiana - 47408
vensamba@indiana.edu

## ABSTRACT

**Network flow involves finding the most beneficial flow through a single source. It is a problem that often arises in Operations research, Computer Science and many real life situations. In this project we compare the linear programming paradigm followed by Ford-Fulkerson's algorithm. And, compare that study with more recent Maximum flow algorithms like the closely related Dinic's block flow method followed by a Push-Relabel method.**

**We gauge these algorithms on a number of graphs, having varying multinomial time complexities. We analyze the algorithms theoretically and practically and compare these values with their practical implementations.**

## Keywords

Graph Algorithms, Maximum-flow, Augmentation Paths, Ford–Fulkerson, Dinic's and Push-Relabel

## 1. INTRODUCTION

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

The maximum flow problem is a special case of more complex network flow problems. This problem involves a directed graph with edges carrying flow. The only relevant parameter is the upper bound on edge flow, called flow capacity. The problem is to find the maximum flow that can be sent through the graph edges source node to a second sink node.

Applications of this problem include Airline flight crews scheduling, the maximum flow of water through a storm sewer system, and the maximum flow of product through a product distribution system, among others. A specific instance of the problem with source node s =A and sink node t = F is shown in Fig. 1 with the solution. In particular, the solution is the assignment of flows to edges. For feasibility, conservation of flow is required at each node except the source and sink, and each edge flow must be less than or equal to its capacity. The solution indicates that the maximum flow from A to F is 15. A cut is a set of edges whose removal will interrupt all paths from the source to the sink. The capacity of a cut is the sum of the edge capacities of the set.

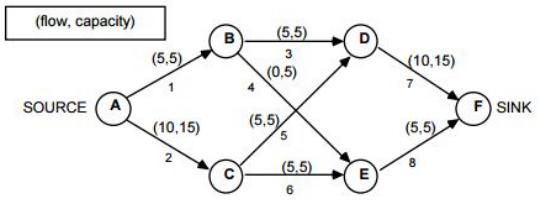



Fig 1: Example Max flow problem

## 2. PROBLEM

If we are given a network such as a directed graph, in which the capacity associated with every edge is c, a starting vertex (source), and an ending vertex (sink). We are asked to associate another value f (flow) for every edge, such that for every vertex (other than the source and the sink), the sum of the values associated to the edges that enter it must equal the sum of the values associated to the edges that leave it.

Consider a few real world max flow problems. Given a list of pipes which are interconnected and having different flow-capacities, we need to calculate the maximum amount of water that we can route between any two ends. Similarly consider a factory located in a city where products are manufactured and needs to be transported to another city where the distribution center is located. Given the one-way road map that connects the two cities and the maximum number of trucks that can drive along each road. We need to calculate the maximum number of trucks that the company can send to the distribution center.

## 3. DEFINITIONS

### 3.1. Residual Graph

Consider two vertices V1 and V2. Let the flow from V1 to V2 to be F12 along the edge E12 and flow from V2 to V1 to be F21 along the edge E21. If no edge exists from one vertex to another, then set the flow value to be 0. Also let C12 and C21 represent the maximum capacity of edges, E12 and E21.

The Residual graph contains all the vertices present in the original graph. For each pair of vertices V1' and V2' in the residual graph, define the following:

$$F'12 = (C12 - F12) + C21$$

$$F'21 = (C21 - F21) + C12$$

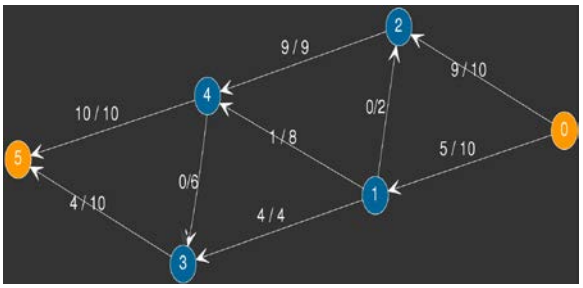
The residual graph in essence, builds a graph representing the maximum possible change in flow from V1 to V2 or, V2 to V1 constrained by the capacities. This graph is used as a tool to increase or decrease _ow values along edges, in order to increase the overall source-sink flow. One can discard all edges with zero flow edges in the residual graph.

## 3.2. Level Graph

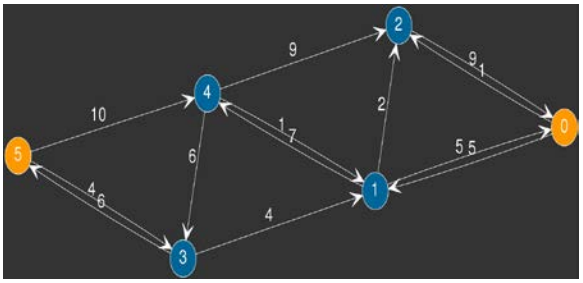
The Level graph is a subgraph of the residual graph. We first partition the vertices of the residual graph into various levels. A vertex is said to belong to level 'I' if the shortest distance from the source vertex is 'I'. After dividing the vertices into levels, retain only those edges, which go from a level 'I' to level I + 1.

## 3.3. Augmentation Path

Given a flow network and its corresponding maximum capacities, an augmenting path is a source-sink path, in which all the edges have flow strictly less than the maximum capacity of the edge. Augmenting paths can be used to increase the source-sink flow. If we find an augmenting path, then we can increase the flow along that path by Min (Cuv - Fuv) for all edges Euv such that Euv belongs to the Augmenting Path. Note Fuv and Cuv denote the flow and capacity values. Figure 2 shows an example graph and its corresponding residual network




(a) Flow Network




(b) Residual graph for flow network

Figure 2: Example of Residual graph

## 3.4. Blocking Flow

Blocking flow is a flow network where all possible source-sink paths have at least one saturated edge i.e. Cuv = Fuv for at least one edge Euv, along every source-sink path. Alternatively, the graph has no augmenting path. A blocking flow need not be a max flow, but the converse is true. Figure 3 shows an example to illustrate this.

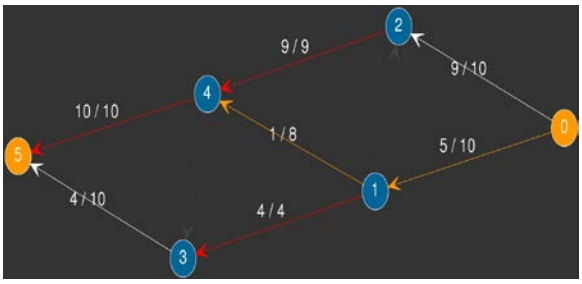



Figure 3: Blocking Flow with Red edge is saturated

# 4 SOLUTIONS

The code for the following study can be found at https://github.com/VishwasKumar/ComparisionOfMaxFlowAlgorithms

## 4.1 Ford-Fulkerson's Algorithm

### 4.1.1 Description

The Ford–Fulkerson algorithm (FFA) is used to calculate the maximum flow in a flow network. It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. It is also used to develop the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

### 4.1.2 Pseudocode

**Algorithm** Ford–Fulkerson

**Inputs** Given a Network $G = (V, E)$ with flow capacity $c$, a source node $s$, and a sink node $t$

**Output** Compute a flow $f$ from $s$ to $t$ of maximum value

1. $f(u, v) \leftarrow 0$ for all edges $(u, v)$
2. While there is a path $p$ from $s$ to $t$ in $G_f$, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
    1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
    2. For each edge $(u, v) \in p$
        1. $f(u, v) \leftarrow f(u, v) + c_f(p)$ (*Send flow along the path*)
        2. $f(v, u) \leftarrow f(v, u) - c_f(p)$ (*The flow might be "returned" later*)

### 4.1.3 Complexity

Maximum flow can be achieved such that no more flow augmenting paths can be found in the graph by adding augmenting paths to the already existing flow in the graph. But, there is no guarantee that this situation will ever be reached, so if the algorithm terminates, one can guarantee the solution is correct. However if the algorithm runs forever, the flow might not even converge towards the maximum flow.

This situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford–Fulkerson is bounded by O(E.f) (big O notation), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O(E) time and increases the flow by an integer amount of at least 1.

### 4.1.4 Drawbacks of Ford-Fulkerson

1. The major drawback is that the running time is not polynomial. So the complexity is exponential (i.e. it is pseudo-polynomial) since it depends on U.

2. Here any augmenting path can be selected. Consider the figure below, the algorithm could take 4000 iterations for a problem with maximum capacity of 2000. This is indicates that the augmenting path chosen is not the ideal choice. An ideal choice of the augmenting path leads to polynomial time algorithms.

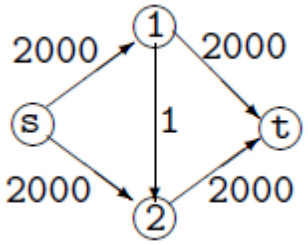



Figure 4: Sample Network

3. Also for irrational capacities, this algorithm may converge to the wrong value.

### 4.1.5 Enhancing the performance of Ford-Fulkerson algorithm

1. Augmenting along maximum capacity augmenting paths.

2. Using the concept of capacity scaling to cope with the non-polynomial running time.

3. Augmenting along the shortest (in number of edges) path. This way each alternative leads to different types of max-flow algorithms.

## 4.2. Dinic's Algorithm

### 4.2.1. Pseudo Code

**Algorithm 1** Dinic Algorithm
**Data:** Graph capacities
**Result:** Maxlflow network and maxflow value
**Initialization :** Set the flow network values to be zero for all edges
**for** *(i=1 ; i<=V ; i++)* **do**
  G' = Level (Residual(G))
  G' = BlockingFlow (G')
  Increment edges in G with values from G'
**end**
**return** G

The runtime of the algorithm is $O(V^2E)$. Finding the level graph or residual graph takes O(E) since, it suffices to iterate over all edges in the graph. A blocking flow can be found in O(V E). Combining these two results, we get the required complexity of the algorithm.

The blocking flow subroutine calls modified DFS 'E' times.

T(BlockingFlow) = E * T(DFSRoutine)

Each DFS routine takes O(V) time to find a source-sink path in each iteration. Additionally, over all the E iterations in the Blocking Flow routine, the DFS routine can delete at most of E edges.
Note that :
T(DFS path finding) = O(V )
T(DFS Deletion over E function calls) = O(E)
T(BlockingFlow) = E * T(DFS path finding) + T(DFS Deletion)
= T(Blocking Flow) = O(V E)

**Algorithm 2** Blocking Flow
**Data:** Graph capacities
**Result:** Blocking Flow of the Graph
**Init** : Set the flow network values to be zero for all edges and Path_Exists = Yes
**while** *Path_Exists=Yes* **do**
  Path = Modified DFS(G)
  /* Find Augmenting Path
  if *No Path found* **then**
    Path_Exists = No
  **else**
    Increase flow along Augmenting Path in G
  **end**
**end**
**return** G

**Algorithm 3** Modified DFS
**Data:** Flow Network
**Result:** Augmenting Path
Do DFS from source
Stop DFS if sink is reached
Delete Vertex is no source-sink path possible from Vertex
**if** *Sink encountered* **then**
  Augmenting_Path = Path found from DFS
**else**
  Augmenting_path = None
  /* If Sink is never encountered then no Source-Sink path exists
**end**
**return** Augmenting_path

### 4.2.2. Analysis

Firstly, we shall prove that the algorithm always terminates. Each iteration in the Dinic function always, increases the source-sink distance in the level Graph. If this is true, since, the source-sink distance is less than V, termination is ensured. Using proof by contradiction, assume that the source-sink distance does not change, after updating the graph with a blocking flow. Path present in the previous iteration cannot exist in current iteration since,
∋ Euv, such that Fuv = Cuv,
Hence this edge will be removed in the Residual graph and hence the path cannot represent in the next iteration. Hence new edge must be present in the path. Any new edge must be from Level 'i' to Level 'j' where 'i' >= 'j'. Note that the edges are with respect to the previous iteration. No edges can go to higher levels, by virtue of construction of the level graph. Hence since edges, are constrained by being from Level 'i' to Level 'j' where 'i' >= 'j', there can exist no source-sink path that is of same length as previous iteration. Hence we see that the algorithm terminates.
**Correct Result**

We shall first try to prove the following result. Let F be a flow-network. Also let G' = Level(Residual(G)). Then the following three statements are equivalent:
1. F is a max flow
2. There is no augmenting path in Graph G'
3. $\exists X, X' \subset V$ where $X \cup X' = V, X \cup X' = \phi$; and $|\text{flow\_val}(X, X')| = \text{capacity}(X, X')$

Proving 1 => 2 is trivial, since if there existed an augmenting path, then it is possible to increase the flow along that path. Hence F will not be a max flow.
Proving 3 => 1 is also an easy task. We see that the |flow_val(X,X')| = Capacity(X,X'). Also |flow_val(X,X')| is also equal to the source-sink flow. Hence, when both the quantities are equal, The source-sink flow must be a maximum and hence it is a max flow.

To Prove 2 => 3, we know that there exists no Augmenting path in G'. Hence there exists no source-sink path in the Graph G'. Let X be set of all vertices that can be reached from the source and let the rest of the vertices belong to X'. Since X and X' have no edges between them in G', all edges in G from X to X' must be saturated. Hence |flow_val(X,X')|= capacity(X,X') for the selected X and X' proving that 2 =>3.

Hence all the above statements are proved. We already know that the algorithm terminates. When it terminates, the resultant graph has no source-sink path since, the Dinic routine loops V times and each iteration increases the Source-Sink distance by 1(Source-Sink distance $\leq$ V-1). Hence there exists no augmenting path on termination. Hence from the earlier stated theorem, the flow is a max flow.

## 4.3 Push-relabel max flow algorithm with FIFO vertex selection rule

### 4.3.1 Pseudocode

1. Send as much flow from *s* as possible.
2. Build a list of all vertices except *s* and *t*.
3. As long as we have not traversed the entire list:
    1. *Discharge* the current vertex.
    2. If the height of the current vertex changed:
        1. Move the current vertex to the front of the list
        2. Restart the traversal from the front of the list.

The Push–relabel algorithm (also known as the Preflow–push algorithm) got its name from the two operations used in the algorithm. The algorithm maintains a "preflow" and converts it into a maximum flow by moving the flow between adjacent vertices using push operations guided by a network which is in turn maintained by relabel operations.

The Push–relabel algorithm is one of the most efficient maximum flow algorithms. The algorithm has a running time of $O(V^2E)$. Efficient augmenting path algorithm have been developed from the concept of distance labels, which in turn can be mixed and matched with the Push–relabel algorithm to create a specialized variant.

### 4.3.2 Procedures - Push, Relabel, and Select

In Push-relabel we search for vertices where the excess e(v) $\geq \Delta$. This way, we make big pushes during the execution of the algorithm relative to other vertices' excesses. This improves the running time of the procedure.

```
PUSH(u, v)
    // e(u) > 0 and (u, v) ∈ A_f and d(u) = d(v) + 1
1   δ(u, v) = min{e(u), c_f(u, v), Δ − e(v)}
    // If a = (u, v) is a forward arc
2   if (u, v) ∈ A
3       f(u, v) = f(u, v) + δ(u, v)
4   else
5       f(v, u) = f(v, u) − δ(u, v)
6   update e(u) and e(v)
7   if e(u) ≤ Δ/2
8       delete u from LIST[d(u)]
9   elseif u ∈ {s, t} and e(v) > Δ/2
10      add u to LIST[d(u)]
```

```
RELABEL(u)
    // e(u) > 0
1   delete u from LIST[d(u)]
2   d(u) = 1 + min{d(v)|(u, v) ∈ A_f}
3   add u to LIST[d(u)]
```

```
SELECT(u)
1   let (u, v) be the current arc of u, and found = FALSE
2   while found ≠ TRUE and (u, v) ≠ NIL
3       if d(u) = d(v) + 1 then set found to TRUE
4   else replace (u, v) with the next arc adjacent to vertex u
5   if found = TRUE then PUSH(u, v)
6   else RELABEL(u)
```

### 4.3.3 First-in First-out Algorithm

We will use a first-in, first-out algorithm for selecting vertices to perform push and relabel operations by creating a queue of vertices.

```
public int maxFlow(int s, int t) {
        int n = cap.length;
        int[] h = new int[n];
        h[s] = n - 1;

        int[] maxh = new int[n];
        int[][] f = new int[n][n];
        int[] e = new int[n];

        for (int i = 0; i < n; ++i) {
            f[s][i] = cap[s][i];
            f[i][s] = -f[s][i];
            e[i] = cap[s][i];
        }

        for (int sz = 0;;) {
            if (sz == 0) {
                for (int i = 0; i < n; ++i)
                    if (i != s && i != t && e[i] > 0) {
                        if (sz != 0 && h[i] > h[maxh[0]])
                            sz = 0;
                        maxh[sz++] = i;
                    }
            }

            if (sz == 0) break;
            while (sz != 0) {
                int i = maxh[sz - 1];
                boolean pushed = false;
                for (int j = 0; j < n && e[i] != 0; ++j) {
                    if (h[i] == h[j] + 1 && cap[i][j] - f[i][j] > 0) {
                        int df = Math.min(cap[i][j] - f[i][j], e[i]);
                        f[i][j] += df;
                        f[j][i] -= df;
                        e[i] -= df;
                        e[j] += df;

                        if (e[i] == 0) --sz;
                        pushed = true;
                    }
                }

                if (!pushed) {
                    h[i] = Integer.MAX_VALUE;
                    for (int j = 0; j < n; ++j) {
                        if (h[i] > h[j] + 1 && cap[i][j] - f[i][j] > 0)
                            h[i] = h[j] + 1;
                    }

                    if (h[i] > h[maxh[0]]) {
                        sz = 0;
                        break;
                    }
                }
            }
        }

        int flow = 0;
        for (int i = 0; i < n; i++) flow += f[s][i];

        return flow;
    }
```

## 4.3.4 Analysis

To analyze the algorithm, we need to familiarize ourselves with what a pass over a queue means in the current context. A pass involves discharging the vertices added to the queue during the previous pass. And, the first pass would involve discharging the vertices added to the queue during initialization.

The number of passes over the queue is at most $4V^2$. Let $\varphi = \max \{h(u) \mid u \text{ is active}\}$:

- If the heights do not change during a given pass, each vertex has its excess transferred to the vertices present at a lower height. Therefore $\varphi$ decreases during the pass. If $\varphi$ does not change, it means that there is at least one vertex whose height increases by at least 1. If $\varphi$ increases during a pass, then some vertex's height increases by $\varphi$.
- The maximum number of passes for which $\varphi$ increases or remains same is $2V^2$. [maximum height of the vertex]
- The total number of passes for which $\varphi$ decreases is $2V^2$. Therefore the total number of passes is $4V^2$.

The number of non-saturating pushes is at most of $4V^3$:

- There can be a non-saturating push per vertex in a single pass. Therefore, the total number of non-saturating pushes is at the most $4V^3$.

On combining our analysis, the total running time for the FIFO algorithm is

⇨ $O(V^3)$

## 4.3.5 Related Problems

In problems where the time limits are strict, push-relabel is more likely to flourish as compared to the augmenting path approach.

Scenarios where only the maximum flow value is needed to be known, the algorithm can be tweaked to work faster and increase the efficiency.

## 4.3.6 Practical implementations

The Push–relabel algorithm has a running time of $O(V^2E)$ and the Push-relabel algorithm with FIFO vertex selection rule has a running time of $O(V^3)$ by choosing appropriate push and relabel operations.

# 5. EXPERIMENTAL SETUP

We now describe our experimental setup in detail.

**Synthetic**. We constructed synthetic segmentation graphs, where we started with a basic grid structure and randomly added edges depending on the vertices.

**Implementations:** We compare the following:

1. Ford Fulkerson algorithm:
2. Dinic's algorithm (DF):
3. Push-Relabel (PRL).

**Common setup** All implementations were coded in JAVA and we wrote a common interface. For all experiments, we generated the max-flow problems, which was then fed as input to all methods. In each experiment, we measured the time taken to compute the max-flow (maxflow-time). Different methods use vastly different internal representations and the init-time varied between families of algorithms. All experiments were performed on a 64-bit 4-core 1.6 GHz Intel(R) Core i7 GNU/Windows machine with 16GB RAM. All times were measured via the JAVA System.nanoTime() function measures process time with a typical resolution of 1 nanosecond.

# 6. RESULTS

The Figure shows the result of our comparison. It compares run-time of the graphs. For synthetic graphs, we can directly control the number of vertices present.
We also noticed that the best performance was by Dinic's algorithm which was consistent. Ford-Fulkerson's algorithm was slow compared to the other two algorithms. Push relabel was observed to be little inconsistent, showing that it was the

fastest among the three compared algorithms for some cases, but overall it was pretty fast.

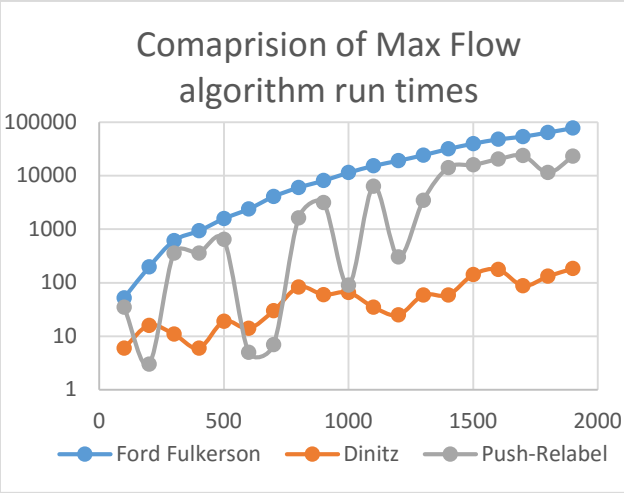



Figure 4: Results of comparison

## 7. CONCLUSION

In summary, we compared a number of max-flow algorithms. Among other things, we found that the most popularly used implementation of Push-Relabel is not the fastest algorithm available, which was surprising to the authors. We hope our findings will be useful. There are a number of ways in which this comparison could be extended, most notably to include multi-core and distributed implementations.

## 8. REFERENCES


• Yefim Dinitz (2006). "Dinitz' Algorithm: The Original Version and Even's Version" (http://www.cs.bgu.ac.il/~dinitz/Papers/Dinitz_alg.pdf). In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. Theoretical.

• Computer Science: Essays in Memory of Shimon Even. Springer. pp. 218–240. ISBN 978-3-540-32880-3.

• Tarjan, R. E. (1983). Data structures and network algorithms.

• B. H. Korte, Jens Vygen (2008). "8.4 Blocking Flows and Fujishige's Algorithm". Combinatorial Optimization:Theory and Algorithms (Algorithms and Combinatorics, 21). Springer Berlin Heidelberg. pp. 174–176. ISBN 978-3-540-71844-4.

• Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. European Journal of Operational Research, 97(3): 509 – 542, 1997.

• Andreas Kaufmann, Bbi5291, Evergrey, Gawi, Giftlite, Headbomb, Magioladitis, Michael Hardy, Milicevic01, NuclearWarfare, Octahedron80, Omnipaedista, R'n'B, Rjwilmsi, Sun Creator, Tcshasaposse, Urod, X7q, 17

• Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 26.2: The Ford–Fulkerson method". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 651–664. ISBN 0-262-03293-7.

• Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc. ISBN 0-13-617549-X.

• Schrijver, A. (2002). "On the history of the transportation and maximum flow problems". Mathematical Programming 91 (3): 437–445. doi: 10.1007/s101070100259

• D.S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. Operations Research, 56(4):992–1009, 2008.

• Andrew V. Goldberg and Robert E. Tarjan, A new approach to the maximum-flow problem.

• Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms.

• Gladius. Algorithm Tutorial: Introduction to graphs and their data structures.

• NilayVaish. Push-Relabel Approach to the Maximum Flow Problem

• Rahul Mehta. Improvements on the Push-Relabel Method: Excess Scaling