Max Flow Algorithms

FORD-FULKERSON, EDMOND-KARP, GOLDBERG-TARJAN

MARKUS JOSEFSSON AND MARTIN MÜTZELL
Max Flow Algorithms

Ford-Fulkerson, Edmond-Karp, Goldberg-Tarjan

Comparison in regards to practical running time on different types of randomized flow networks.

Markus Josefsson

Martin Mützell

Degree Project in Computer Science, DD143X

Supervisor: Per Austrin

Examinor: Örjan Ekeberg

Sweden, Stockholm 2015
Abstract

Max Flow Algorithms
Ford-Fulkerson, Edmond-Karp, Goldberg-Tarjan

The original algorithm proposed by Ford and Fulkerson to solve the maximum flow problem is still in use but is far from the only alternative. This paper introduces that algorithm as well as the similar Edmond-Karp and the more modern Goldberg-Tarjan.

Ford-Fulkerson uses depth-first-searches to find augmenting paths through a residual graph. Edmond-Karp instead uses breath-first-searches to achieve a polynomial time complexity. Goldberg-Tarjan solves the problem by gradually pushing flow through the residual graph.

A comparison on randomized graphs where the size, graph density and the maximum edge capacity was varied one at a time was performed. The results show that the Goldberg-Tarjan algorithm has higher performance than the others, but only if it uses heuristics.
Max Flow Algorithms

Markus Josefsson/Martin Mützell

marjos@kth.se

mmutzell@kth.se

Contents

1. Introduction 5
   1.1 About Algorithms 5
   1.2 The Maximum Flow Problem 6
   1.3 Problem Statement 6

2. Background 7
   2.1 About Graphs 7
      2.1.1 Directed Graphs 7
   2.2 Flow Network 8
   2.3 Flow-Capacity-Directed Graph 12

3. Method 13
   3.1 Ford-Fulkerson Max Flow Algorithm 13
      3.1.1 Residual Graf 13
      3.1.2 Augmenting Path 15
      3.1.3 The Algorithm 16
      3.1.4 Example 17
      3.1.5 F-F Time-Complexity Analysis 17
1. Introduction

It is explained in this section why algorithms that can solve the maximum flow problem are important and interesting. It includes an overview of algorithms in general, a description of the maximum flow problem without mathematical definitions and some examples of its applications.

1.1 About Algorithms

The first step toward an understanding of why the study and knowledge of algorithms is important is to define exactly what is meant by an algorithm. According to the textbook *Introduction to Algorithms*, “an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output,” (Cormen, Leiserson, Rivest and Stein, 2001). In other words, algorithms are like road maps for accomplishing a given, well-defined task. Consequently this means that a chunk of code that calculates the terms of the Fibonacci sequence is an implementation of a particular algorithm. Even a simple function for adding two numbers is an algorithm. To summarize “An algorithm is a sequence of step designed to solve a problem”, (Evans and Minieka, 1992, p. 33).

Some algorithms, like those that compute the Fibonacci sequences, are intuitive and may be innately embedded into our logical thinking and problem solving skills. However, for most people, complex algorithms are best studied so they can be used as building blocks to more efficiently solve a logical problem in the future. In fact one may be surprised to learn just how many complex algorithms people use every day when they check their e-mail, listen to music on their computers as well as other repeated tasks.

One of the most important aspects of an algorithm is how fast it is. It is often easy to come up with an algorithm to solve a problem, but if the algorithm is too slow, it would be unable to solve large instances of the problem within any relevant timeframe. The purpose of the algorithm is most often to be used over and over again; either in different contexts or to solve a larger problem. In both scenarios, speed is highly desirable. Although worst-case time complexities can be determined (and are determined for each algorithm in section 3), the practical running time is in most cases wildly different from those. This is the reason why benchmarking tests, such as this one, are important.

This paper studies and compares the three Max Flow algorithms mentioned below. They were chosen so that different time periods could be represented.

Ford-Fulkerson (1950-1960)
Edmond-Karp (1970)
Goldberg-Tarjan (1980-1990)
1.2 The Maximum Flow Problem

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. The general problem can be described as: given a network, as well as a source (start) and a sink (destination) in that network, how does one route as much flow as possible from the source to the sink?

General characteristics of the networks where the problem is applicable are:

- Source: “materials” are produced at a steady rate
- Sink: “materials” are consumed at the same rate
- Flows through conduits are constrained to max values

Examples of applications for the problem can be:

- Liquid flow through pipes
- Current flow through an electrical circuit
- Information in a communications network
- Production factory with various tools
- Controlling network/Internet traffic

“We are daily and directly convinced of the importance of traffic networks, and more particularly of road networks, and of the desirability that the factors that ensure their smooth running should be well understood. There is a vast literature on the subject, and a substantial body of theory, which must indeed take account of new factors” (Whittle, 2007, p. 85).

In 1955, Lester R. Ford, Jr and Delbert R. Fulkerson created the first known algorithm, the Ford-Fulkerson algorithm. Over the years, various improved solutions to the maximum flow problem were discovered, notably the shortest augmenting path algorithm of Edmonds and Karp and independently Dinitz; the blocking flow algorithm of Dinitz; the push-relabel algorithm of Goldberg and Tarjan.

1.3 Problem Statement

How does the original algorithm proposed by Ford and Fulkerson hold up in comparison with the very similar Edmond-Karp and the modern Goldberg-Tarjan in regards to practical running time on different types of randomized flow networks?
2. **Background**

This section is dedicated to the definitions and terminology that will be used in this paper.

2.1 **About Graphs**

This paper assumes that the reader has basic knowledge of graph theory. However, a refresher on the subject might be in order. Section 2.1 is dedicated to this.

A graph is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$ and a relation that associates with each edge two vertices (not necessarily distinct) called its end points.

In other words, a graph is another term for a network, i.e. a set of objects (called vertices or nodes) that are connected together. The connections between the vertices are called edges or links.

![Undirected Graph](http://upload.wikimedia.org/wikipedia/commons/9/96/Goldner-Harary_graph.svg)

*Figure 2.1 Undirected Graph*
2.1.1 Directed Graphs

“A directed network or directed linear graph \( G = [N; A] \) consists of a collection \( N \) of elements \( x, y, \ldots \), together with a subset \( A \) of the ordered pairs \( (x, y) \) of elements taken from \( N \). It is assumed throughout that \( N \) is a finite set, since our interest lies mainly in the construction of computational procedures. The elements of \( N \) are variously called nodes, vertices, junction points, or points; member of \( A \) are referred to as arcs, links, branches, or edges.” (Ford and Fulkerson, 1962, p. 2).

In formal terms, a directed graph is a pair \( G=(V, A) \) or sometimes \( G=(V,E) \). In this paper \( G=(V, E) \) is going to be used. When drawing a directed graph, the edges are typically depicted as arrows indicating the direction, such as in figure 2.1.1. If all edges are bidirectional, or undirected, the graph is an undirected graph, as illustrated in figure 2.1 above.

![Directed graph diagram](http://mathinsight.org/directed_graph_definition)
2.2 Flow Network

The logic of a flow network and some terminology to know:

A flow network is a directed graph $G=(V,E)$ where each edge $(u,v) \in E$ has a non-negative capacity $c(u,v)$.

1. **NON-NEGATIVE CAPACITY**

   $C_e =$ Capacity of edge $e$ (non-negative)
   
   $0 \leq C_e$

2. **CAPACITY CONSTRAINT**

   $f(e) =$ Flow on edge $e$ (it is non-negative and less than or equal to the capacity of the edge)
   
   $0 \leq f(e) \leq C_e$ is a function that assigns a number $f(e)$ to each $e$.

3. **CONSERVATION CONSTRAINT** (Nothing is lost)

   “The flows in a network are assumed to satisfy the continuity condition that the flows do not stagnate, i.e., the sum of the entering (or outgoing) flows at each node is equal to zero”
   
   (Masao, 1969, p. 76)

   $\forall v \neq s, t$ For every vertex $v$ other than $s$ and $t$.

   $\Sigma f(e \text{ into } v) = \Sigma f(e \text{ out of } v)$ Flow into vertex is equal to flow out of vertex.
4. **VALUE OF FLOW**

\[ V(f) = \Sigma f(e \text{ out of } s) \]

\( V(f) = \) Value of the flow, \( \Sigma f(e) = \) sum of the value \( e \) out of \( s \).

Now the problem is to find an assignment flow \( f \) to maximize \( V(f) \).

**Def:**

\[ f(\text{ out of } v) = \Sigma f(e \text{ out of } v) \]

**Def:**

\[ f(\text{ into } v) = \Sigma f(e \text{ into } v) \]

\( S \subseteq V \)

\( S = \) some subset of vertices

\( V = \) The set of all the vertices

\( V \neq v, S \neq s \)

\[ f(\text{ out of } S) = \Sigma f(\text{ out of } v, \text{ where } v \in S) \]

What is out of a set of nodes. little \( v \) is contained in \( S \).

\[ f(\text{ into } S) = \Sigma f(\text{ into } v, \text{ where } v \in S) \]

What is into a set of nodes.

---

**FLOW NETWORK**

![Flow Network Diagram](https://www8.cs.umu.se/~jopsi/dinf504/chap14.shtml)

Figure 2.2.1  Flow Network
The Maximum Flow Problem Definition

A directed graph G, a start node s, and a sink node t, are given. Each edge e in G has an associated non-negative capacity c(e), where for all non-edges it is implicitly assumed that the capacity is 0.

Figur 2.2.2 Flow/Capacity directed graph G

http://en.wikipedia.org/wiki/Maximum_flow_problem

The goal is to push as much flow as possible from s to t in the graph (Figure 2.2.2). The rules are that no edge can have flow exceeding its capacity, and for any vertex except for s and t, the flow in to the vertex must equal the flow out from the vertex. That is,

**Capacity constraint:** On any edge e we have \(0 \leq f(e) \leq c(e)\).

**Flow conservation:** For any vertex v, flow in equals flow out.

Subject to these constraints, one wants to maximize the total flow into t. For instance, imagine somebody wants to route message traffic from the source to the sink, and the capacities tell us how much bandwidth we’re allowed on each edge.

E.g., in the above graph, what is the maximum flow from s to t?

How can one see that the above flow is a maximum?

Is the maximum s-t flow \(\leq\) the capacity of the minimum s-t cut?

An important property of flows, that will not be proved in this paper is that the maximum s-t flow is in fact equal to the capacity of the minimum s-t cut. This is called the Maxflow-Mincut Theorem. In fact, the algorithm will find a flow of some value k and a cut of capacity k, which will be proofs that both are optimal!

This paper does not include proofs for most of the theorems presented. However, the proofs can be found via the references given.
2.3 Flow-Capacity-Directed Graph

A directed graph $G$, a start node $s$, and a sink node $t$. Each edge $e$ in $G$ has an associated non-negative capacity $c(e)$, where for all non-edges it is implicitly assumed that the capacity is 0. No edge can have flow exceeding its capacity.

Figure 2.3 Flow/Capacity directed graph.

http://en.wikipedia.org/wiki/Maximum_flow_problem
3. Method

A performance comparison was made using network graphs generated from *A Graph Generation Package* (Johnsonbaugh and Kalin, 1991). For each set of parameters, ten graphs were generated and the average running time for each algorithm was logged. The running time was measured using the `getRealTime` function (Nadeau, 2012).

Sections 3.1 to 3.3 presents the three algorithms and describes how they operate. Section 3.3 also presents the global relabeling heuristic. Goldberg-Tarjan with this heuristic is the fourth algorithm that was tested.

3.1 Ford-Fulkerson Max Flow Algorithm

The maximum flow problem concerns itself with determining how to move as much data or material as possible from one point in a network to another. This has obvious relevance today to the Internet, where getting as much data as possible from one point to another is important. It also comes up in many business settings, and is an important part of operations research.

The first efficient algorithm for finding the maximum flow was conceived by two Computer Scientists, named Ford and Fulkerson. The algorithm was subsequently named the Ford-Fulkerson algorithm, and it is one of the more famous algorithms in computer science. This paper is going to study this algorithm to a lower degree, for more details see *Flows in Networks* (Ford and Fulkerson, 1962). In the last 50-60 years, a number of improvements have been made to the Ford-Fulkerson algorithm to make it faster, some of which are very complex.

3.1.1 Residual Graph

We can define the Residual Graph, which provides a systematic way to search for forward-backward operations in order to find the maximum flow. The residual capacity of an edge is $c_f(u, v) = c(u, v) - f(u, v)$. This defines a residual network denoted $G_f(V, E_f)$, giving the amount of available capacity. See that there can be a path from $u$ to $v$ in the residual network, even though there is no path from $u$ to $v$ in the original network. Since flows in opposite directions cancel out, decreasing the flow from $v$ to $u$ is the same as increasing the flow from $u$ to $v$.

Some refreshment steps:

1. Find an s-t directed path in graph $G$ below.
   
   Ex. s-u-v-t

2. Find the minimum Capacity of the edge on that path s-u-v-t.
   
   Let “b” be the minimum residual capacity on path $P$. 
\[ b = \text{Min}(20, 30, 20) = 20. \]

3. Build a residual Graph \( G_f \), given \( f \). For any edge \( e = (u, v) \) with \( f(e) \geq 0 \) then direct an edge from \( v \) to \( u \) (backward edge) with residual capacity \( \text{residual capacity} = f(e) = c'(e) \) where \( c'(e) \) = backward residual capacity.

4. Find an \( s \)-\( t \) directed path in the residual graph \( G_f \), (if there is one, else stop) and let “b” be the minimum residual capacity on path \( P \).
Ex. s-v-u-t

\(b=\text{Min}(10, 20, 10)=10\) is the minimum residual capacity.

5. Now augment the flow \(f\) using the path \(P\) and the minimum capacity \(b\).
6. Then go to back and iterate this process as long as you are able to find an \(s-t\) flow in the residual graph.

\[\text{Figure 3.1.1}.4\quad \text{Residual Graph } G_f \text{ with no } s-t \text{ path.}\]

Now there is no longer a path in the residual graph \(s\) to \(t\), so the algorithm says stop. A network is at maximum flow if and only if there is no augmenting path in the residual network \(G_f\).

### 3.1.2 Augmenting Path

Is a path \((u_1, u_2, u_3, \ldots, u_k)\) in the residual network, where \(u_1=s\) (source) and \(u_k=t\) (terminal/sink) and residual capacity \(c_f(u_i, u_{i+1})>0\). A network is at maximum flow if and only if there is no augmenting path in the residual network \(G_f\). See figures in section 3.1.1.

An augmenting path is a simple path- a path that does not contain cycles- through the graph using only edges with positive capacity from the source to the sink. So augmenting path is a path constructed by repeatedly finding a path of positive capacity from a source to a sink and then adding it to the flow.

Augmenting means increase-make larger. As mentioned above, in a given flow network \(G=(V,E)\) and a flow \(f\), an augmenting path \(p\) is a simple path from source \(s\) to sink \(t\) in the residual network \(G_f\). By the definition of residual network, we may increase the flow on an edge \((u,v)\) of an augmenting path by up to a capacity \(c_f(u,v)\) without violating constraint, on whichever of \((u,v)\) and \((v,u)\) is in the original flow network \(G\). Also the maximum amount by which we can increase the flow on each edge in an augmented path \(p\) is called the residual capacity of \(p\).
3.1.3 The Algorithm

Part1: setup
Start with null flow:
\[ f(u,v) = 0 \quad \forall (u,v) \in E; \]
Initialize residual network:
\[ N_f = N; \]

Part2: Loop
Repeat
Search for directed path \( p \) in \( N_f \) from \( s \) to \( t \)
If (path \( p \) found)
\[ D_f = \min \{ c(u,v), (u,v) \in p \}; \]
For (each \( (u,v) \in p \) do
If (forward \( (u,v) \))
\[ f(u,v) = f(u,v) + D_f; \]
if (backward \( (u,v) \))
\[ f(u,v) = f(u,v) - D_f \]
update \( N_r; \)
until (no augmenting path)
3.1.4 Example

![Figure 3.1.4.1](image1)
![Figure 3.1.4.2](image2)

![Figure 3.1.4.3](image3)
![Figure 3.1.4.4](image4)

3.1.5 F-F Time-Complexity Analysis

As mentioned above, one of the most important aspects of an algorithm is how fast it is. The exact speed of an algorithm depends on where the algorithm is run, as well as the exact details of its implementation, computer scientists typically talk about the runtime relative to the size of the input.

We may say the following about **Ford-Fulkerson Runtime Analysis**:

. Depends on the order of choosing the augmenting paths P

. Each iteration takes m time, where m is the number of edges in the network algorithm

. The flow network increases at least one unit each iteration

. For the max Flow f*, the maximum value |f*| denotes a bound in the number of iteration

. Total running time is O(|f*|m)
3.2 Edmonds-Karp Algorithm

The Edmonds-Karp algorithm is one of the first and simplest max flow algorithms. It was published in 1972 by Jack Edmonds and Richard Karp. It is a small variation on the Ford-Fulkerson algorithm from 1956, that limits the number of augmenting paths to \( O(nm) \), where \( n = |V| \), and brings the worst case running time from \( O(|f^*|m) \) to \( O(nm^2) \).

3.2.1 The Algorithm

The algorithm works by repeatedly finding the shortest augmenting path using a breadth first search from \( s \) to \( t \). When such a path \( P=(v_1, v_2, \ldots, v_k) \) where \( k \geq 2, v_1=s, v_k=t \) is found, it calculates the bounding minimum capacity on that path, and sends that much flow over the same path. It keeps doing this in the residual network until no more augmenting paths exists.

Correctness follows from the fact that the algorithm terminates when no more augmenting paths from \( s \) to \( t \) are found in the residual network, and the fact that the algorithm always keeps a valid flow. The algorithm never violates any capacity constraints, because when it sends flow, it sends flow according to the minimum residual capacity on the path. It also never produces any excess in nodes other than \( s \) and \( t \), because all flow is pushed along paths from \( s \) to \( t \).

3.2.2 Example

![Figure 3.2.2.1](image1.png)

![Figure 3.2.2.2](image2.png)

![Figure 3.2.2.3](image3.png)

![Figure 3.2.2.4](image4.png)
3.2.3 E-K Time-Complexity Analysis

The algorithm performs a breadth first search for each augmenting path in the graph. A single breadth first search takes $O(m)$ time. Every time the algorithm finds an augmenting path, it does a push along it. There must be at least one edge $(u, v)$ on this path that is saturated, namely the edge with the minimum capacity. For this edge to be in the path, the distance from $s$ to $u$ must be less than the distance from $s$ to $v$. After the edge has been saturated, it can not be used again before flow has been pushed the opposite way, which requires that the distance from $s$ to $v$ becomes less than the distance from $s$ to $u$. The distance from $s$ to any node can not be greater than $n$, and if the distance never decreases, so an edge can only be saturated $n$ times.

The only way we modify the distances is by pushing flow along the augmenting path. Saturated edges are effectively removed, and back edges are added back in if their residual capacity was zero. Removing an edge can not reduce the distance to a node. Adding an edge could, but the edges $(v_i, v_{i-1})$ we might add point the opposite way on the augmenting path which was found in a breadth first search. Adding $(v_i, v_{i-1})$ back in can not reduce the distance to $v_{i-1}$, because the distance to $v_i$ was already greater than the distance to $v_{i-1}$.

To summarize, there are $m$ edges that can be saturated $n$ times, each time requiring a breadth first search which takes time $O(m)$. This results in the running time of $O(nm^2)$.

- Use BFS Breadth_First_Search instead of DFS Depth_First_Search
- This variant of Ford-Fulkerson algorithm runs in $O(nm^2)$

Above is an implementation of the Ford-Fulkerson method for computing the maximum flow in a network in $O(nm^2)$ time. The algorithm is identical to the Ford-Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be a shortest path that has been available. This can be found by a breadth-first-search, as we let edges have unit length. The running time of $O(nm^2)$ is found by showing that each augmenting path can be found in $O(m)$ time, that every time at least one of the $m$ edges becomes saturated, that the distance from the saturated edge to source along the augmenting path must be longer than last time it was saturated, and that the length is at most $n$. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically.
3.3 Goldberg-Tarjan Algorithm

The push relabel algorithm of A. V. Goldberg and R. E. Tarjan works by manipulating the preflow in a graph. First step is saturating all the edges exiting the source. Next step is moving the excess into nodes that are estimated to be closer to the target. If at some point the excess of a node can not reach the target, the excess is moved back into the source. In the end, the preflow is therefore an actual flow and in fact it is the maximum flow.

3.3.1 Preflow and Labels

Remember that a valid flow is an assignment to each edge such that the flow of an edge never exceeds the capacity of that edge and the incoming flow of any vertex except the source or sink is equal to the outgoing flow. A valid preflow is similar, but allows the incoming flow to be greater than or equal to the outgoing flow.

The push-relabel method maintains a preflow, rather than an actual flow, during execution of the algorithm. While there is a valid preflow to a graph, there exists a set of vertices whose incoming flow is greater than the outgoing flow. This is the set of active vertices, and it is on these nodes the push and relabel functions are applied. When the set of active vertices is empty, the preflow is a valid flow.

The push-relabel method also assigns each vertex v a label \( d(v) \) which restricts where excess flow can be pushed and it will be modified by the relabel function. The value of a label can be interpreted as the last known distance to the sink. Thus, at the start of most algorithms using the push-relabel method, a breadth-first search is performed from the sink, labeling each vertex with the distance to the sink.

3.3.2 Push

The push function moves excess flow from an active vertex v to an adjacent vertex w and is only allowed if \( d(v) = d(w) + 1 \). This increases the flow of the edge \((v,w)\) by P and therefore decreases the excess flow of v and increases the excess flow of w by P. To guarantee that a valid preflow is maintained, P cannot be bigger than the excess flow of v or the residual capacity of \((v,w)\).

3.3.3 Relabel

The relabel function changes the label of an active vertex v and is only allowed if v has no adjacent vertices that can be pushed to. The label of v is set to the minimum required for a push action to be available. Note that the label of any vertex v never decreases in value (Goldberg and Tarjan, 1988).
3.3.4 The Algorithm

This is the generic algorithm, as presented in Goldberg and Tarjan (1988).

**Push(v, w).**
Applicability: \( v \) is active, \( rG(v, w) > 0 \) and \( d(v) = d(w) + 1 \).
Action: Send \( P = \min(e(v), rG(v, w)) \) units of flow from \( v \) to \( w \) as follows:
\[
\begin{align*}
& f(v, w) \leftarrow f(v, w) + P; \\
& f(w, v) \leftarrow f(w, v) - P; \\
& e(v) \leftarrow e(v) - P; \\
& e(w) \leftarrow e(w) + P
\end{align*}
\]

**Relabel(v).**
Applicability: \( v \) is active and \( \forall \ w \in V, rG(v, w) > 0 \implies d(v) \leq d(w) \).
Action: \( d(v) \leftarrow \min\{d(w) + 1 | (v, w) \in rG\} \).
(if this minimum is over an empty set, \( d(v) \leftarrow \infty \))

**Procedure Max-Flow(V, E, s, t, c);**
<<initialization>>
<<initialize preflow>>
\[
\forall (v, w) \in (V - \{s\}) \times (V - \{s\}) \text{ do begin} \\
& f(v, w) \leftarrow 0; f(w, v) \leftarrow 0; \\
\text{end;}
\]
\[
\forall v \in V \text{ do begin} \\
& f(s, v) \leftarrow c(s, v); \\
& f(v, s) \leftarrow -c(s, v); \\
\text{end;}
\]
<<initialize labels and excesses>>
\[
& d(s) \leftarrow n; \\
& \forall v \in V - \{s\} \text{ do begin} \\
& & d(v) \leftarrow 0; \\
& & e(v) \leftarrow f(s, v); \\
\text{end;}
\]
<<loop>>
while \( \exists \) a basic operation that applies do
select a basic operation and apply it;
return(f);
end.
3.3.6 Example

Figure 3.3.5.1

Figure 3.3.5.2

Figure 3.3.5.3

Figure 3.3.5.4

Figure 3.3.5.5

Figure 3.3.5.6

Figure 3.3.5.7

Figure 3.3.5.8

Figure 3.3.5.9

Figure 3.3.5.10

Figure 3.3.5.11

Figure 3.3.5.12
3.3.6 G-T Time Complexity Analysis

The generic algorithm terminates after $O(n^2m)$ basic operations. This is derived from the sum of all possible relabel operations, all possible saturating pushes and all possible nonsaturating pushes. The main parts of the proof in (Goldberg and Tarjan, 1998) is presented below.

Relabeling operations only apply to vertices $v \in V - \{s,t\}$. A relabeling of $v$ increases $d(v)$. The label $d(v)$ is zero initially, and the label can grow to at most $2n-1$. Therefore, there are at most $2n-1$ relabelings of each vertex in $V - \{s,t\}$, and the total number of relabelings is at most $(2n-1)(n-2) < 2n^2$.

For any pair of vertices $v$ and $w$, consider the saturating pushes from $v$ to $w$ and from $w$ to $v$. If there are any such pushes, then either $(v,w) \in E$ or $(w,v) \in E$. Consider a saturating push from $v$ to $w$. In order to push flow from $v$ to $w$ again, the algorithm must first push flow from $w$ to $v$, which cannot happen until $d(w)$ increases by at least 2. Similarly, $d(v)$ must increase by at least 2 between saturating pushes from $w$ to $v$. Since $d(v) + d(w) \geq 1$ when the first push between $v$ and $w$ occurs and $d(v) + d(w) \leq 4n-3$ when the last such push occurs, the total number of saturating pushes between $v$ and $w$ is at most $2n-1$. Thus the total number of saturating pushes is at most $2n-1$ per edge, for a total over all edges of at most $(2n-1)m < 2nm$.

Let $\Phi = \sum_{\{v \mid v \text{ is active}} d(v)$. Each nonsaturating push from a vertex $v$ to another vertex $w$ causes $\Phi$ to decrease by at least 1, since the push makes $v$ inactive and $d(w) = d(v) - 1$. A saturating pushing operation causes $\Phi$ to increase by at most $2n-1$. The total increase in $\Phi$ over the entire algorithm due to relabeling operations is at most $(2n-1)(n-2)$. Before the first push or relabel operation $\Phi$ is zero, and at the end of the algorithm $\Phi$ is also zero. Thus the total decrease in $\Phi$, and hence the total number of nonsaturating pushing operations, is equal to the total increase in $\Phi$, which is at most $(2n-1)2nm + (2n-1)(n-2) \leq 4n^2m$ (recall the assumption $m \geq n-1$). Thus, the total number of operations is $O(n^2m)$ as proposed.

3.3.7 The Global Relabel Heuristic

Goldberg and Tarjan proposed a heuristic to partially prevent unessesary operations: “The heuristic periodically updates the distance labels by performing breadth-first searches backward from the sink and source in the residual graph.” (Goldberg and Tarjan, 1998, p. 938). This allows the program to skip a number of unessessary relabel operations and detect any nodes that cannot reach the sink. The push-relabel algorithm using this heuristic performed after every $n$ pushes was implemented for testing alongside the above three algorithms.
3.4 Implementation

This subsection gives information that allows the reader to replicate the circumstances in which the results were achieved.

3.4.1 Code Details

Four algorithms was implemented in C++ as functions in the same file. The fourth one being Goldberg-Tarjan with the global relabel heuristic. The main-function, when called, sequentially read each of the previously generated files containing the graphs into a capacity-matrix that was then used as a parameter to each of the four algorithms. The time it took for the algorithm to return was measured and printed into a log-file for later review.

As hinted to above, all algorithms was implemented using capacity matrices, not with lists. This theoretically increases the time complexity of the algorithms, the difference is smaller in practice as the sizes of the lists representing the residual graph would be under constant change. The matrix was used due to easier implementation and because no argument stating that any of the algorithms would be given an advantage due to this was found.

Both versions of Goldberg-Tarjan used a simple queue (first-in first-out) to decide which active node to push and/or relabel.

3.4.2 Test Environment

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Windows 7 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7-2600 CPU @ 3.40GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>16384 MB RAM</td>
</tr>
<tr>
<td>Language</td>
<td>C++11 / C</td>
</tr>
<tr>
<td>Compiler</td>
<td>TDM-GCC 4.8.1 64-bit Release</td>
</tr>
</tbody>
</table>
4. Results

This section compares the performance of the three algorithms with respect to the size, density and the maximum capacity of the edges in the graph. The size is \( n = |V| \) and the graph density is defined as \( D = \frac{m}{n(n-1)} \), where \( m = |E| \).

4.1 Literature Findings

Ford-Fulkerson can outperform Edmond-Karp in specific cases, notably in grid graphs as depth-first-search typically is faster on those. This was statistically proven by Laube and Nebel (2013).

The paper *An experimental comparison of max flow algorithms* (Friis, and Olesen, 2014) contains many pieces of information related to this. Edmond-Karp did generally underperform and some Goldberg-Tarjan with heuristics proved to be fastest on every test.
4.2 Performance Test

Each cell in the presented tables represents the average running time of ten network graphs generated with the given parameters. Each chart is presented in both a linear and a logarithmic time scale for readability. In the tables and charts, global relabel is named mass relabel.
550 Vertices
Capacities from 1 to 100

<table>
<thead>
<tr>
<th>Graph Density</th>
<th>Ford-Fulkerson</th>
<th>Edmond-Karp</th>
<th>Goldberg-Tarjan</th>
<th>Mass Relabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>327.919</td>
<td>86.6051</td>
<td>761.143</td>
<td>1.90048</td>
</tr>
<tr>
<td>0.5</td>
<td>1476.08</td>
<td>360.021</td>
<td>1378.78</td>
<td>6.50043</td>
</tr>
<tr>
<td>0.9</td>
<td>2076.82</td>
<td>595.834</td>
<td>604.734</td>
<td>9.30919</td>
</tr>
</tbody>
</table>

(milliseconds)

Chart 4.2.2
550 Vertices
Graph Density 0.5

<table>
<thead>
<tr>
<th>Maximum Capacity</th>
<th>Ford-Fulkerson</th>
<th>Edmond-Karp</th>
<th>Goldberg-Tarjan</th>
<th>Mass Relabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.3022</td>
<td>156.009</td>
<td>1303.388</td>
<td>4.59995</td>
</tr>
<tr>
<td>100</td>
<td>1476.08</td>
<td>360.021</td>
<td>1378.78</td>
<td>6.50043</td>
</tr>
<tr>
<td>10000</td>
<td>125104</td>
<td>385.322</td>
<td>1385.78</td>
<td>6.50101</td>
</tr>
</tbody>
</table>

(ms)
5. Discussion

The first and most apparent thing to note is the dominance of Goldberg-Tarjan with the global relabeling heuristic. It wasn’t only faster than the other algorithms in all the test cases, the difference became more pronounced as the graph complexity increased.

The same algorithm without the heuristic had its average running time severely lengthened by a fairly common edge case. The reason for this is most likely that during the execution, the minimum cut (the set of edges that separate the source from the sink with the minimum sum of capacities) became saturated with several active vertices on the source side of the cut. This leads to an unproportionally high number of pushes and relabels of all vertices on the source side of the minimum cut. This is the same phenomenon observed by Friis and Olesen (2014) mentioned in section 4.1.

However, because the average running time of that algorithm was mostly dependent on the frequency of such cases, only having ten test-cases of a given parameter set are not enough to make the numbers trustworthy, which is readily apparent in chart 4.2.3.

Ford-Fulkerson performed admirably in very specific cases, most notably when all edge capacities were equal to 1. However, it is apparent why other algorithms have been developed to solve the maximum flow problem more efficiently. In particular, its performance is severely affected by the size of the edge capacities, while there is no notable change in the running time of the others, except when moving away from boolean edge capacities.

Although most experienced programmers would agree that implementing Ford-Fulkerson or Edmond-Karp is rather easy, Goldberg-Tarjan not that much further on the difficulty scale, as we perceived it. Adding the global-relabel heuristic was also easy, as it simply required a call to the initial backwards breath-first-search from the sink at regular intervals.
6. Conclusion

A naive Ford-Fulkerson can in specific cases be faster than a naive Edmond-Karp, but neither of them will outperform a Goldberg-Tarjan algorithm with appropriate heuristics in any sort of randomized graph. Ford-Fulkerson is especially inefficient when the capacities of the edges vary greatly.

Since the difficulty to implement Goldberg-Tarjan with at least the global relabel heuristic is very comparable to the difficulty to implement the aforementioned two algorithms, it is highly recommended for use in solving the maximum flow problem in any situation where higher performance is advantageous.

For future studies, other heuristics can be applied to Goldberg-Tarjan for comparison. Slightly different implementations can also be used, for example replacing the capacity matrix with capacity lists. Other types of graphs can also be investigated; it is recommended to read Friis and Olesen (2014) for inspiration.
7. **Bibliography**


